University of Thessaly

Department of Computer & Communication Engineering

Program of Graduate Studies
«Computer & Communication Science and Technology»

Spring Semester

Embedded Systems

- Teachning notes -

Labros Bisdounis, Ph.D.

Volos 2006

Selected books

- F. Vahid, T. Givargis, "Embedded system design: A unified hardware-software introduction", Wiley, 2002.
- W. Wolf, "Computers as components: Principles of embedded computer systems design", Morgan Kaufmann, 2001.
- P. Marwedel, "Embedded systems design", Kluwer Academic, 2006.
- T. Noergaard, "Embedded systems architecture: A comprehensive guide for engineers and programmers", Elsevier, 2005.
- S. Heath, "Embedded systems design", Elsevier, 2002.
- J. Hennessy, D. Patterson, "Computer organization and design", Elsevier, 2003 (μετάφραση: Οργάνωση και σχεδίαση υπολογιστών, Εκδόσεις Κλειδάριθμος).
- C. Hammacher, Z. Vranesic, S. Zaky, “Computer organization", McGraw Hill, 2002 (μετάφραση: Οργάνωση και αρχιτεκτονική ηλεκτρονικών υπολογιστών, Εκδόσεις Επίκεντρο).

Selected books (cont’d)

- S. Furber, "ARM system-on-chip architecture", Addison-Wesley, 2000.
- A. Jerraya, W. Wolf, "Multiprocessor systems-on-chips", Morgan Kaufmann, 2004.
- M. Keating, P. Bricaud, "Reuse methodology manual for system-on-a-chip designs", Kluwer Academic, 2002.
- N. Voros, K. Masselos, "System-level design of reconfigurable systems-on-chips", Springer, 2005.
- W. Chen, "The VLSI handbook", CRC Press, 2000.

Contents of the course

- Introduction to embedded systems.
- Specification and modeling of embedded systems.
- Embedded systems design flows.
- Embedded systems synthesis (system-level synthesis, hardware synthesis, software generation) and estimation.
- Power optimization in embedded systems.
- Verification and co-simulation of embedded systems.
- Reduced Instruction Set Computing (RISC) machines.
- The ARM processor.
- Application Specific Instruction-set Processors (ASIP) design.
- Very Large Instruction Word (VLIW) processors.
- System-on-chip design and prototyping platforms.
- Reconfigurable systems.
- Communication in embedded systems.

1. Introduction to Embedded Systems

University of Thessaly

Department of Computer and Communication Engineering

Basic idea

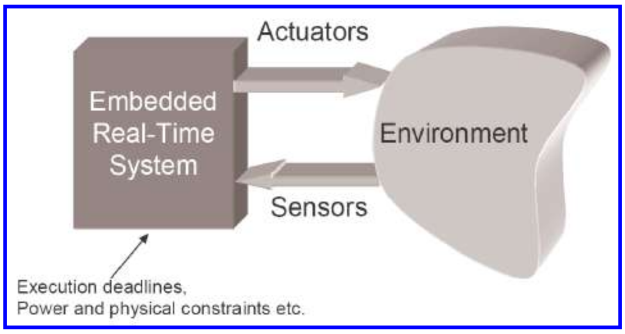
- The heterogeneity of today's embedded systems faces developers and engineers with new problems when it comes to specifying, simulating, designing and optimising such complex systems.
- Implementations are typically comprised of programmable components, dedicated hardware components, communication and memory subsystems.
- The design of embedded systems is driven by cost vs. performance trade-offs. The optimization involves the simultaneous consideration of several incomparable and often competing objectives, such as cost, power consumption, reliability etc.
- As a consequence, much effort and automated design tools are necessary in order to handle the complexity of today's embedded systems, and find the trade-off which is the most suitable for the market requirements.

What is an embedded system ?

- An embedded system is any device which includes a programmable component but itself is not intended to be a general-purpose computer.
- An embedded system:
  - ✓ is a collection of programmable components surrounded by application-specific hardware components and other peripherals.
  - ✓ and interacts continuously with its environment through sensors (with the general meaning).

Main characteristics of an embedded system

- Dedicated and application-specific (not general purpose).
- Contains at least one programmable component.
- Interacts continuously with the environment.
- It is real-time: must meet external timing constraints (deadlines).
- Must meet other constraints: power consumption, physical constraints, cost, reliability, safety.

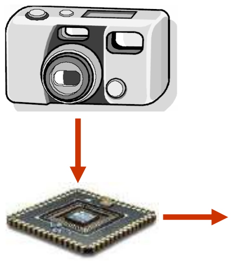

Actuators
Embedded Real-Time System
Environment
Sensors
Execution deadlines,
Power and physical constraints etc.

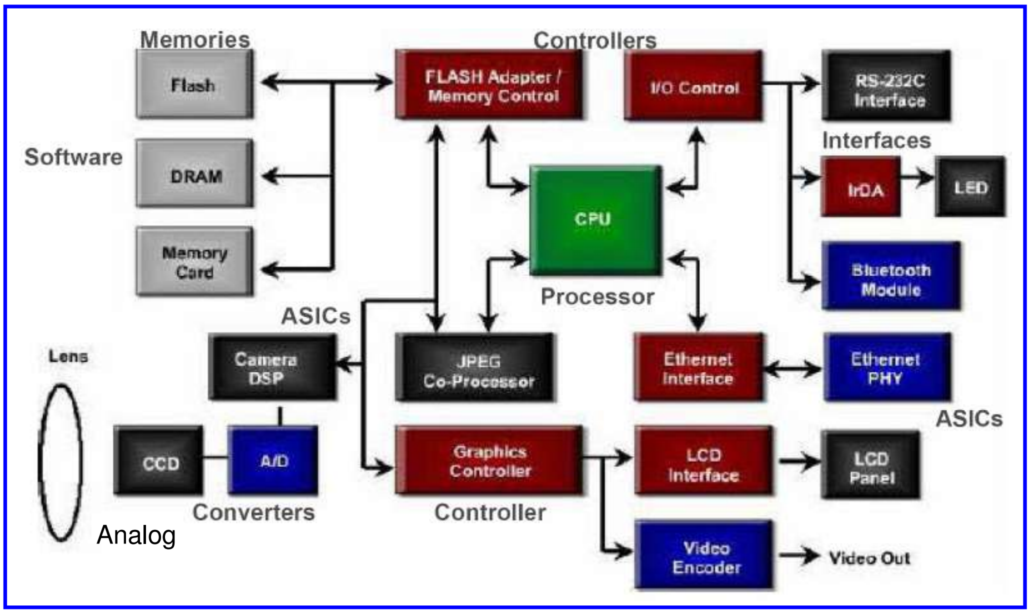
Present and future of computing !

Components of an embedded system

- Digital components: processors, memories, controllers, buses, application specific circuits, peripherals (interface circuits).
- Embedded software.
- Analog components: sensors, actuators.
- Converters: A/D and D/A.

Example: Digital camera





Embedded software and digital components

- Embedded software: software running on embedded processors:
  - ✓ Application programs.
  - ✓ Real-time operating system.
  - ✓ Peripheral’s drivers.

- Digital (hardware) components:
  - ✓ Programmable processors.
  - ✓ Dedicated hardware implementing critical and demanding tasks or tasks that are not suitable for software implementation.
  - ✓ Reconfigurable hardware (e.g. FPGAs).

Computing elements in embedded systems

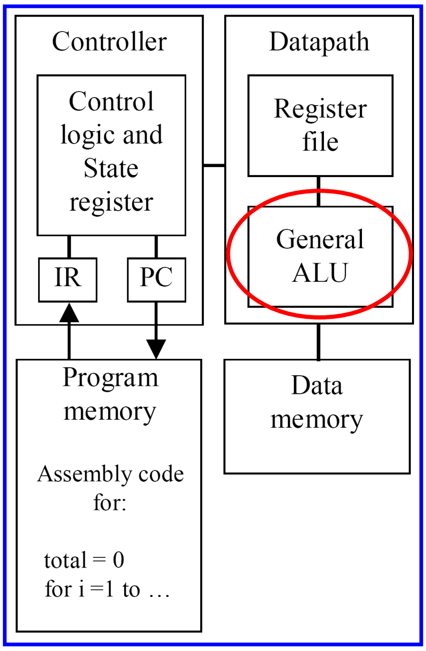
- Processing:
  - ✓ Used to transform data.
  - ✓ Implemented using programmable processors and custom hardware.
- Storage:
  - ✓ Used to maintain data.
  - ✓ Implemented using memory modules.
- Communication:
  - ✓ Used to transfer data between processors, custom hardware blocks, peripherals and memories within a system.
  - ✓ Implemented using buses in most cases.
- Peripherals (interfaces) and controllers.

Processors in embedded systems

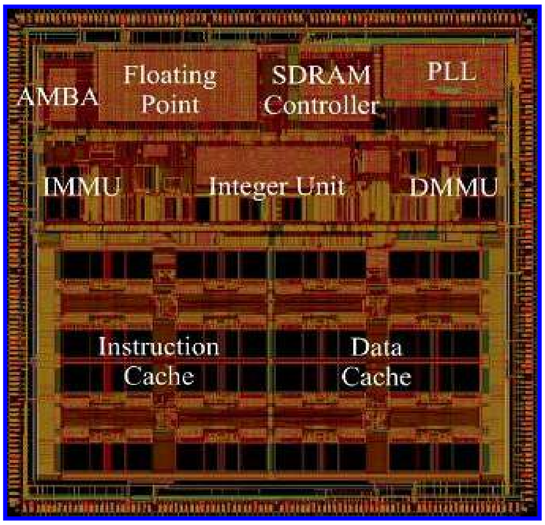
- Need for energy, code size and run time efficiency.
- In general, a processor is the device that runs a number of algorithms and contains control and datapath units.
- General purpose processors (GP):
  - ✓ Perform a variety of computational tasks.
  - ✓ Flexibility and low cost.
  - ✓ Slow and power hungry.
- Application-specific processors (ASIPs):
  - ✓ Tuned for application domain, but programmable.
  - ✓ Fast and power efficient (compared to GP).
- Application-specific circuits (ASICs):
  - ✓ Customized hardware components for specific task.
  - ✓ Fast, power efficient, minimal area.
  - ✓ Inflexibility and high cost.

General-purpose processors

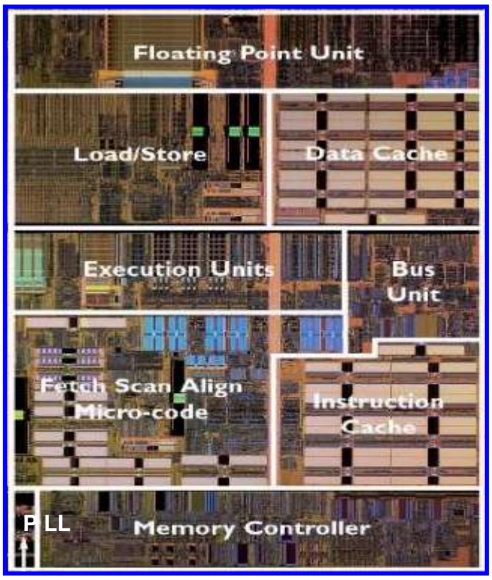
- Programmable devices used in a variety of applications.
- Contain program memory, general datapath with large register file and general ALU.
- Low time-to-market and NRE (non-recurring cost).
- High flexibility.



General-purpose processor examples



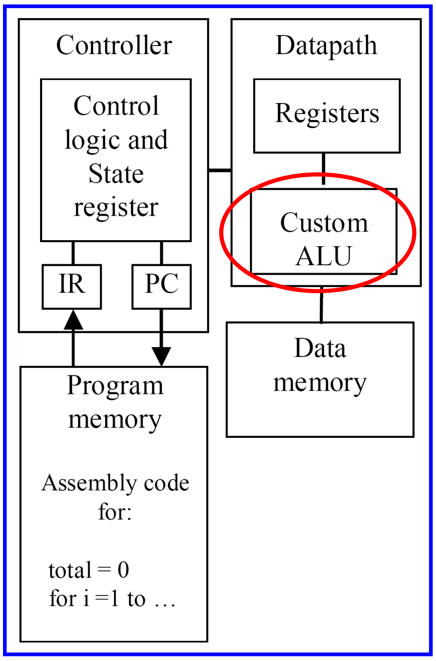
ARM10
core processor



AMD Athlon 64
core processor

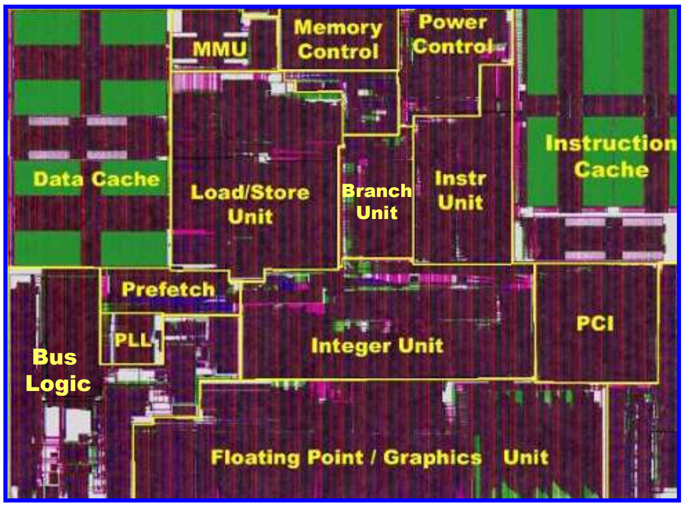
Application-specific processors (ASIPs)

- Programmable devices optimized for a particular application or family of applications having common characteristics.
- Contain program memory, optimized datapath and specific functional units.
- Use specific instruction set.
- High performance, small size and low power consumption.
- Usually they exhibit small flexibility.
- DSP, VLIW, network processors, etc.



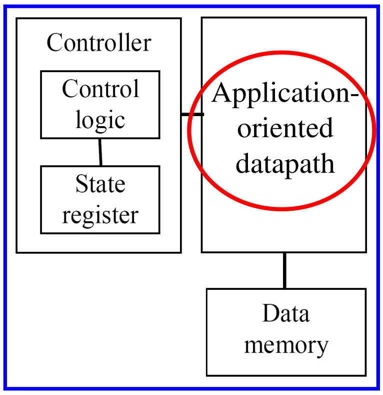
ASIP example: Transmeta Crusoe processor

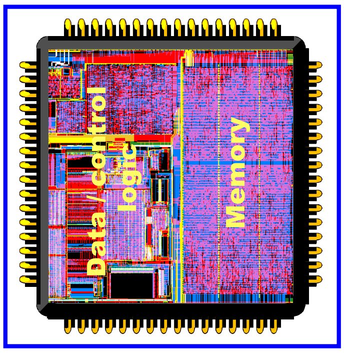
- VLIW architecture (instruction-level parallelism fixed at compile-time) that contains a specialized floating-point unit for graphics applications.



Application-specific circuits

- Digital circuits designed to implement exactly one algorithm (application or part of application).
- Custom-designed circuits that are necessary if ultimate speed or energy efficiency is the goal (known as coprocessors or hardware accelerators).
- Contain only the components needed for the execution of a specific algorithm (no program memory is needed).
- Fast, low power consumption, small size, high cost for low volume.

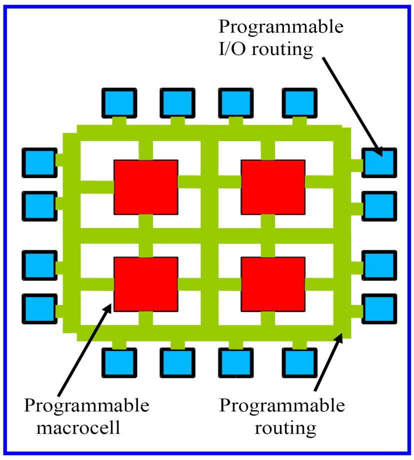


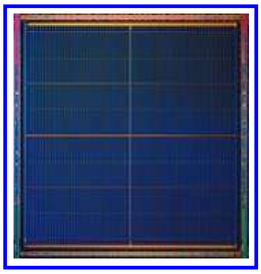


Audio processing ASIC

Programmable devices

- Prefabricated digital devices that can be purchased and programmed by the designer/user.
- Programmable arrays of generic logic modules, programmed by the designer/user and not by the semiconductor foundry.
- PLDs, FPGAs.
- Alternative to ASICs with low NRE cost, and fast availability.
- Penalty on area, performance and power consumption.





Xilinx FPGA

Memory in embedded systems

- Memory is used in embedded systems for storage purposes and offers access capabilities (read and/or write).

- Main characteristics:
  - ✓ Storage permanence: ability of memory to hold stored bits after they are written.
  - ✓ Write ability: manner and speed a memory can be written.

- There are many different types of memories:
  - ✓ SRAM, DRAM.
  - ✓ ROM, PROM.
  - ✓ EPROM, EEPROM, Flash.
  - ✓ NVRAM.

Memory in embedded systems (cont'd)

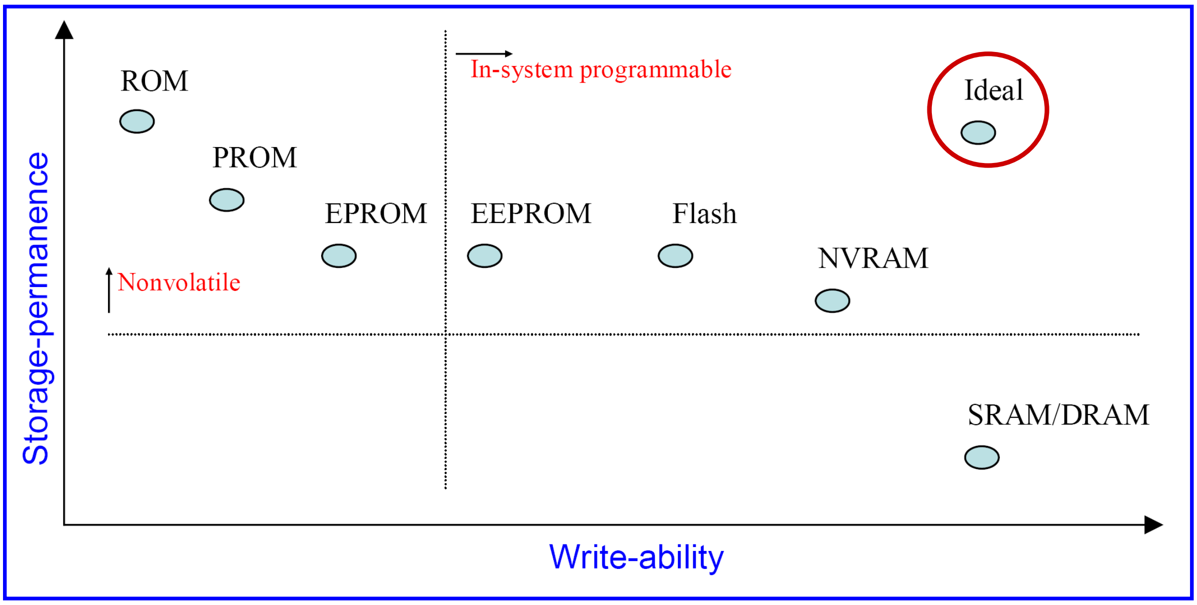
- SRAM, DRAM:
  - ✓ Processor writes to memory simply and fast.
  - ✓ Holds bits as long as power supplied to memory (DRAM begin to lose bits almost immediately after written – refreshing is needed).
  - ✓ SRAM is more expensive, faster (access time 10ns vs 60ns) and more reliable than DRAM

- ROM, PROM:
  - ✓ Once data has been written onto a ROM it cannot be removed and can only be read (mask programmed: data are written during fabrication).
  - ✓ Never loses its data.
  - ✓ PROM is a variation of ROM manufactured as blank memory on which data can be written with a special device (programmer).

Memory in embedded systems (cont'd)

- EPROM:
  - ✓ Retains contents until it is exposed to ultraviolet light. The light clears the contents, making possible to reprogram the memory.
  - ✓ To write the memory, a special device (programmer) is needed.
- EEPROM, Flash:
  - ✓ EEPROM is a special type of PROM that can be erased by exposing it to an electrical charge.
  - ✓ Retains its contents even when the power is turned off.
  - ✓ Processor can write to memory, but slower than RAM.
  - ✓ The principal difference between EEPROM and Flash is that EEPROM requires data to be written or erased one byte at a time whereas Flash allows data to be written or erased in blocks (faster).
- NVRAM:
  - ✓ Retains contents when power is turned off. It is an SRAM that is made non-volatile through a connection with a power source (battery).
  - ✓ Often it is a combination of SRAM and EEPROM.

Memory in embedded systems (cont'd)



Communication in embedded systems

- Communication in an embedded system accounts for the transfer of data between processors, custom hardware blocks, peripherals and memories.
- Implemented using buses.
- Example: Common forms of communication are when a processor read or writes a memory or when a processor reads or writes a peripheral's register.
- Connectivity schemes:
  - ✓ Serial communication (USB, RS232 etc.): use single wire, high throughput for long distance communication, low cost).
  - ✓ Parallel communication (PCI, AMBA etc.): use multiple wires, high throughput for short distance communication, high cost).
  - ✓ Wireless communication (Infrared, RF).
- Each connectivity scheme has an associated protocol describing the rules for transferring data over it.

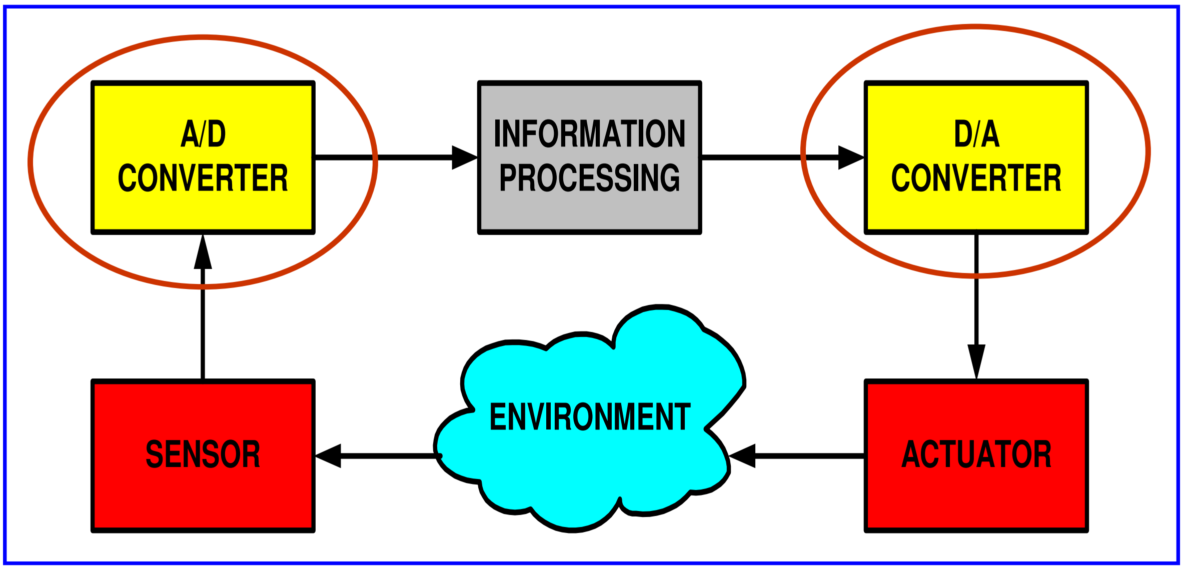
Communication in embedded systems (cont'd)

- The main issues regarding the communication of a processor with the peripherals through a system bus are:
  - ✓ The addressing procedure: how the system address map is used in order the processor to communicate with the memory and the peripherals.
  - ✓ The interrupt-driven communication: the processor accepts interrupt signals in order to read and process data from a peripheral.
  - ✓ The direct memory access (DMA) for transferring data between memories and peripherals, without going through the processor.
  - ✓ Arbitration: how to handle simultaneous servicing requests of peripherals.

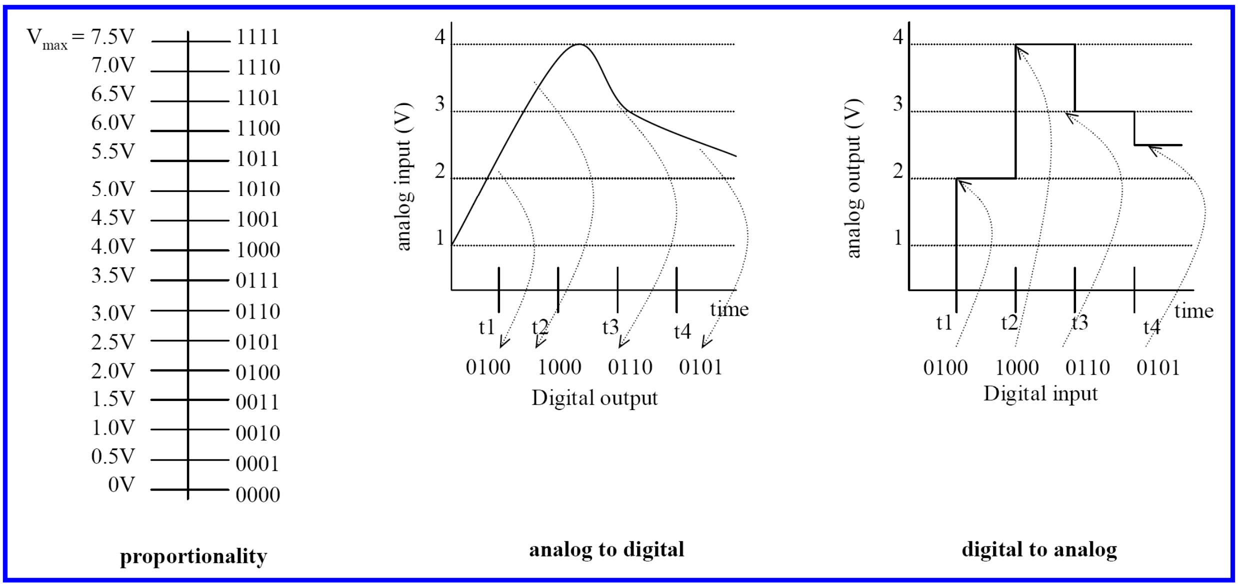
Peripherals and controllers in embedded systems

- Peripherals and controllers perform specific computation tasks.
- Custom single-purpose processing blocks:
  - ✓ Designed by us for a unique task.
  - ✓ Predesigned (by others) for a common task.
- Examples:
  - ✓ Timers, counters: to measure timed events or indicate that a maximum count reached.
  - ✓ UART: universal asynchronous receiver transmitter that takes parallel data and transmits serially, receives serial data and converts to parallel.
  - ✓ LCD interface: interface the system to a liquid crystal display.
  - ✓ External memory controller, PCI controller, USB interface, Ethernet or other network type interface.

Converters in embedded systems



Converters in embedded systems (cont'd)



Main issues:

- Sampling: how often is the signal converted.
- Quantization: how many bits used to represent a sample.

Analog components in embedded systems

- Sensors
  - ✓ Capture physical stimulus (heat, light, sound, pressure, magnetism, mechanical motion).
  - ✓ Typically, they generate a proportional electrical current.
- Actuators
  - ✓ Convert a command to a physical stimulus (heat, light, sound, pressure, magnetism, mechanical motion).

Microphone

Megaphone

Laser diode, transistor

Antenna

DC motor

Accelerometer

Sensors and actuators in embedded systems

- Sensors can be designed for virtually every physical stimulus. Firstly, they capture the physical data and then they process them.
- Many physical effects are used for constructing sensors: generation of voltages in an electric field (law of induction), light-electric effect etc.
- Examples: heart monitoring sensors, car sensors (rain sensors for wiper control, proximity sensors), pressure sensors (touch pads and screens), audio sensors, motion sensors, thermal sensors (SARS detection through high fever) etc.
- Actuators produce output physical stimulus for various environments: motor control actuators (industrial applications), optical actuators (IR), thermal actuators, MEMS devices (Micro-Electro-Mechanical Systems) etc.
- MEMS technology regards the integration of mechanical elements and electronics on a common silicon substrate. The electronics are fabricated using silicon processes, and the micromechanical components are fabricated using micromachining processes that selectively etch away parts of the silicon wafer or add new structural layers to form the mechanical devices.
- Applications: biotechnology (DNA identification), communications (RF-MEMS), accelerometers (air-bags).

Embedded applications

- Automotive electronics (airbag control, dashboard info, ABS, consumption control etc.).
- Aircraft electronics (guidance, flight control, air quality control, pressure control etc.).
- Telecommunication systems (mobile phones and network cards, mobile base stations etc.).
- Medical systems (diagnostic and monitoring systems, radiation systems).
- Defence systems (radars and safety communication systems, navigation systems like GPS etc.).
- Consumer multimedia electronics (cameras, game machines etc.).
- Industrial process control systems.
- Robotics (electro-mechanical systems).

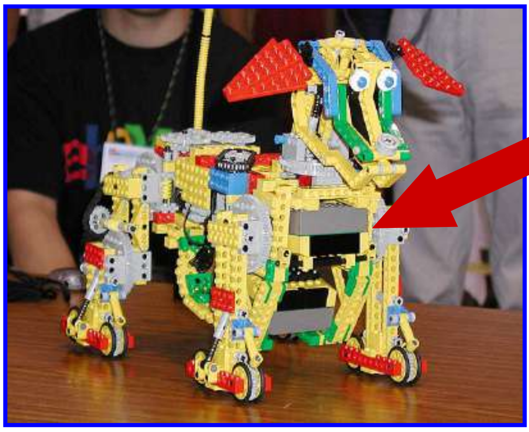
Embedded applications in every-day life

- Mobile phone:
  - ✓ Multiprocessor system (8-32 bit processor for user interface, DSP, 32-bit processor for IR and Bluetooth ports)
  - ✓ 8-100 MB memory, custom chips, integrated camera, megaphone, speaker etc.

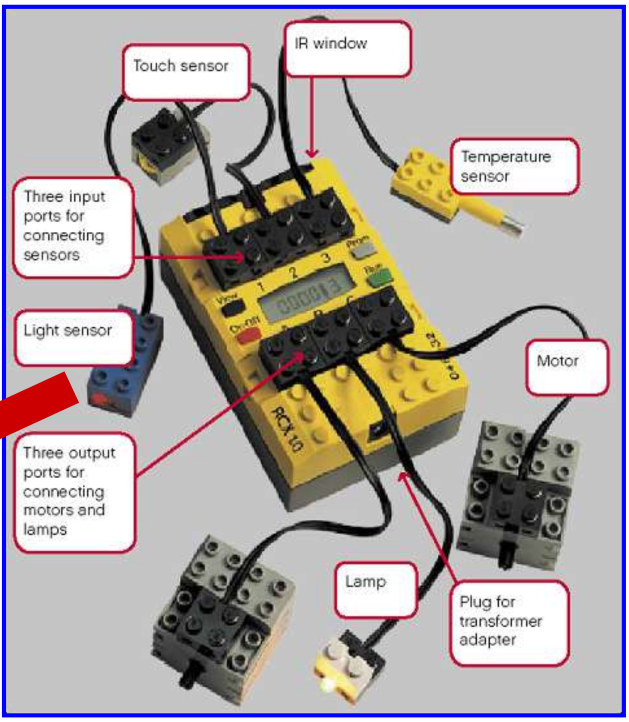
- Smart beer glass:
  - ✓ Combines a fluid-level sensor with a simple 8-bit processor and an RF system with internal antenna. The system checks the fluid level & alerts the servers when close to empty.
  - ✓ Integrates several technologies such as: radio transmission, sensor engineering, computer monitoring.

Embedded applications for kids

- Lego mindstorms robotics kit: combines an 8- bit controller with 64 kB memory.
- Electronic circuits to interface the processor with the various sensors and motors.
- Good way to start learning embedded systems…

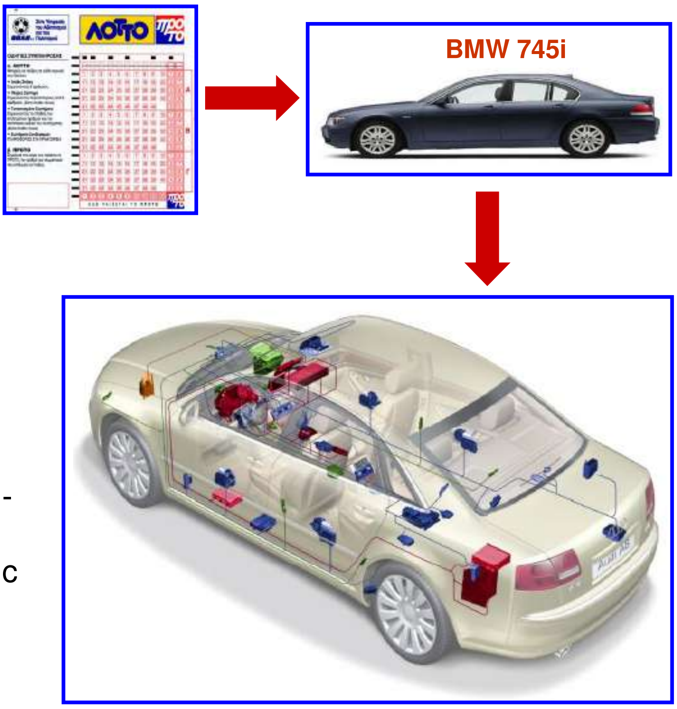


Control unit


Touch sensor
IR window
Temperature sensor
Three input ports for connecting sensors
Light sensor
Motor
Three output ports for connecting motors and lamps
Lamp
Plug for transformer adapter
RCX 1.0

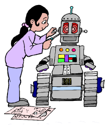
Embedded applications for lotto winners

- 53 8-bit, 11 32-bit and 7 16-bit microprocessors (71 in total!).
- Multiple networks.
- Sensors and actuators distributed all over the vehicle.
- Windows CE operating system.
- Engine management: consumption, ignition, emission control etc.
- Instrumentation: data acquisition, display and processing.
- Safety and stability: airbags, ABS (anti-lock braking system), ESP (electronic stability control), efficient and automatic gearboxes etc.
- Entertainment and comfort: Radio-CD, A/C, television, GPS etc.

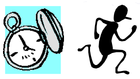


Constraints of embedded applications

- Reactivity requirement.



- Timing constraints.

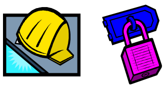


- Power dissipation constraints.

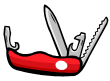
- Size and weight constraints.

- Cost constraints.

- Safety and security constraints.



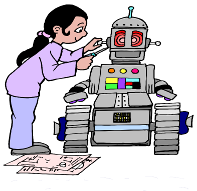
- Reliability constraints.



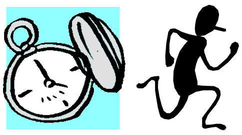
- Time-to-market constraints.

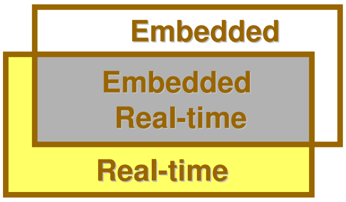
Reactivity and timing constraints

- Reactivity requirement: embedded systems are in continual interaction with its physical environment through sensors and actuators, and execute at a rate determined by the environment.



- Timing constraints:
  - ✓ Most of the embedded systems have to perform in real-time which means that if data is not ready by a certain deadline (i.e. reaction of the system within a certain time interval dictated by the environment), the system fails.
  - ✓ A real-time constraint (deadline) is called hard, if not meeting that constraint could result in a failed operation of the system. All other time-constraints are called soft (if not meeting, the operation of the system will be degraded, but the system will not fail).
  - ✓ Embedded & real-time terms are almost synonymous.
  - ✓ Most embedded systems are real-time and most real-time systems are embedded.





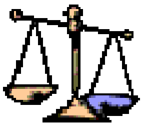
Embedded

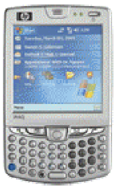
Embedded
Real-time

Real-time

Power, size and weight constraints

- Power consumption constraints:
  - ✓ High power dissipation needs strong power supply and expensive cooling system.
  - ✓ High power consumption leads to short battery life time (very critical issue in mobile/portable applications).
- Size and weight constraints:
  - ✓ Critical for mobile, portable devices (e.g. PDAs, mobile phones, cameras).
  - ✓ Very critical for specific medical applications (e.g. pills with integrated camera and data acquisition system).





11 x 26 mm

Cost constraints

- Cost constraints: embedded systems are very often mass products in highly competitive markets and have to be shipped at a low cost (e.g. mobile phones market).

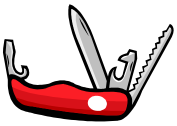
- We are mainly interesting in manufacturing cost and design cost.

- Main cost factors: design time and effort, type of used components (processors, memory, I/Os), technology (board-based, system-on-chip, type of manufacturing processes), testing time, power consumption.

- Non-recurring engineering (NRE) costs (design cost and prototypes development) are becoming very high, and because of that:
  - ✓ It is difficult to come out with low quantity products.
  - ✓ Hardware and software platforms are introduced, which are used for products of similar type.

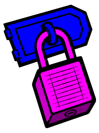
Reliability, safety and security constraints

- Reliability constraints:
  - ✓ Reliability is the probability of an embedded system working correctly provided that it was working at t = 0.
  - ✓ Even perfectly designed systems can fail if the operation assumptions (workload, possible errors) turn out to be wrong. So, we have to be very carefully when define the operation assumptions for a specific application in a given environment.



- Safety constraints: embedded systems are often used in life critical applications (automotive electronics, nuclear plants, medical applications, defence applications etc.).

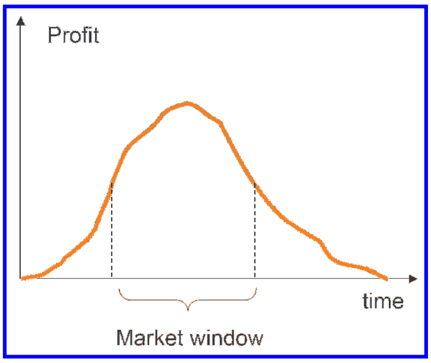
- Security constraints: embedded systems for communication applications must often support confidentiality and authenticity.



- In order to guarantee the above constraints during the design, exhaustive verification of the certain properties of the designed system must be performed, as well as synthesis and design based on automatic design tools.

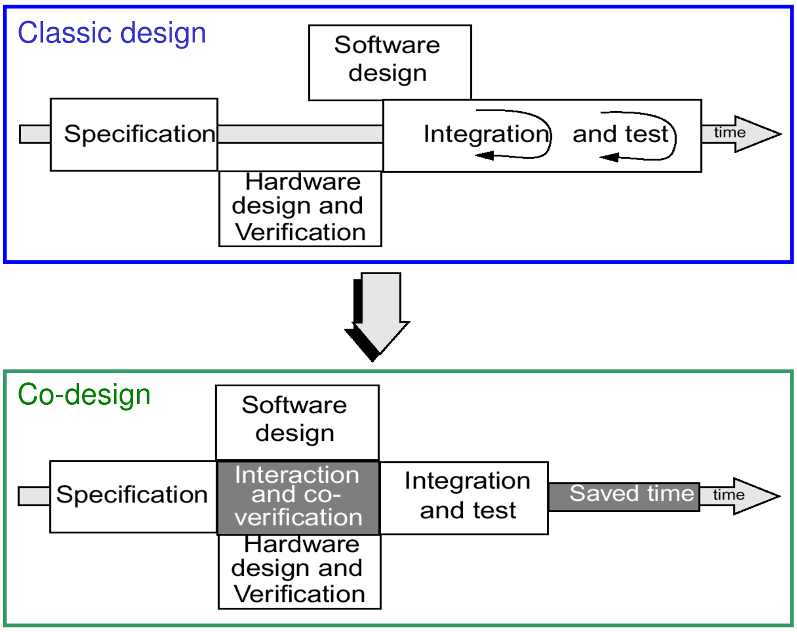
Time-to-market constraints

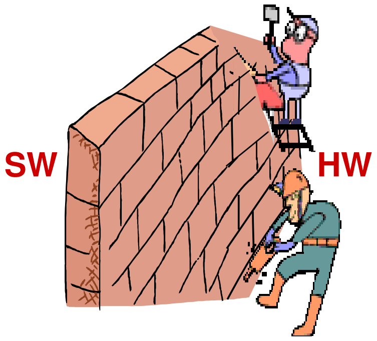
- Time-to-market constraints: in highly competitive markets, it is critical to catch the market window: a short delay may have catastrophic financial consequences, even if the quality of the product is excellent.
- Development time has to be reduced and some ways to achieve that are:
  - ✓ Efficient design methodologies.
  - ✓ Efficient design tools.
  - ✓ Reuse of previously designed and verified parts (hardware and software).
  - ✓ Use of existing hardware-software prototyping platforms.
  - ✓ Design team understanding both software and hardware.



What is hardware-software co-design ?

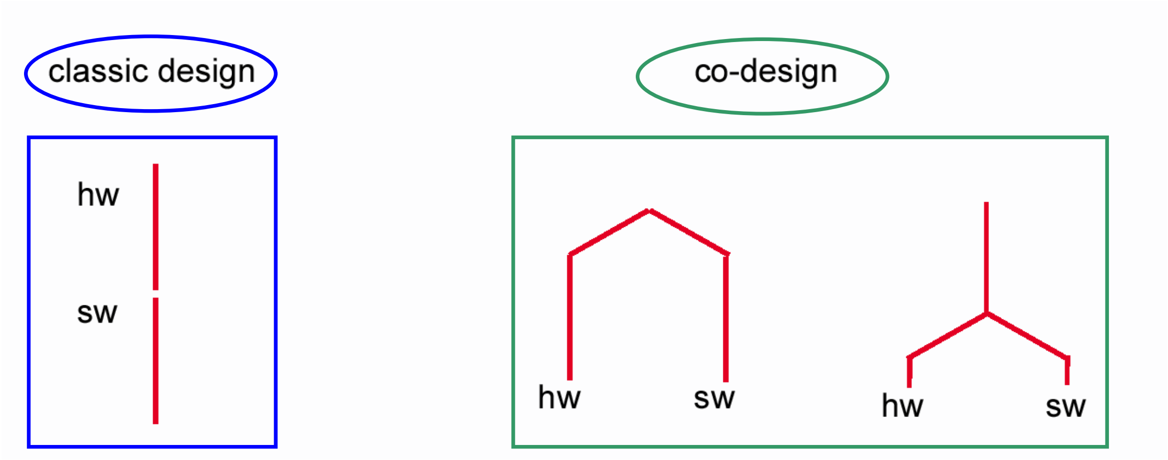
Co-design is the concurrent design of hardware and software components of a digital system.





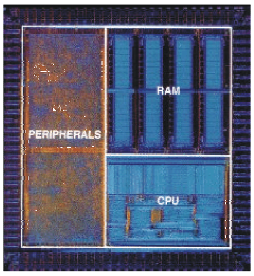
The wall between hardware and software must be torn down !

What is hardware-software co-design ? (cont’d)



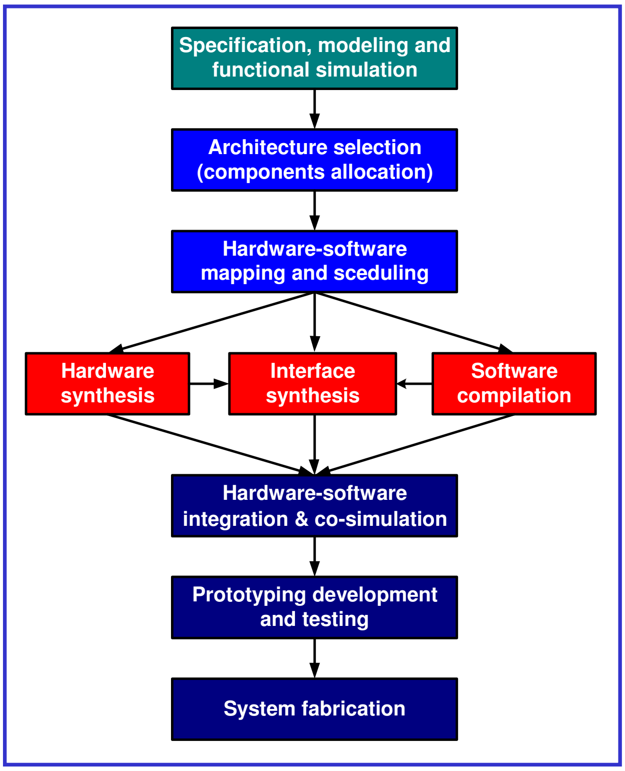
Difficulties in hardware-software co-design

- The difficulties in designing embedded systems are due to the fact that such systems has high complexity, are dedicated towards a certain application, and must be efficient in what concerns:
  - ✓ Run-time.
  - ✓ Power consumption.
  - ✓ Code-size (low memory requirements).
  - ✓ Cost (minimization of hardware resources).
  - ✓ Development time (time to market).
  - ✓ Size and weight.
- In order to achieve all the above, embedded systems have to be highly optimized.
- Both hardware and software aspects have to be considered simultaneously (co-design) in order to achieve:
  - ✓ A good solution by balancing hardware and software resources (flexibility).
  - ✓ Exploration of more design alternatives.
  - ✓ Design of systems-on-chip (optimized complex systems).

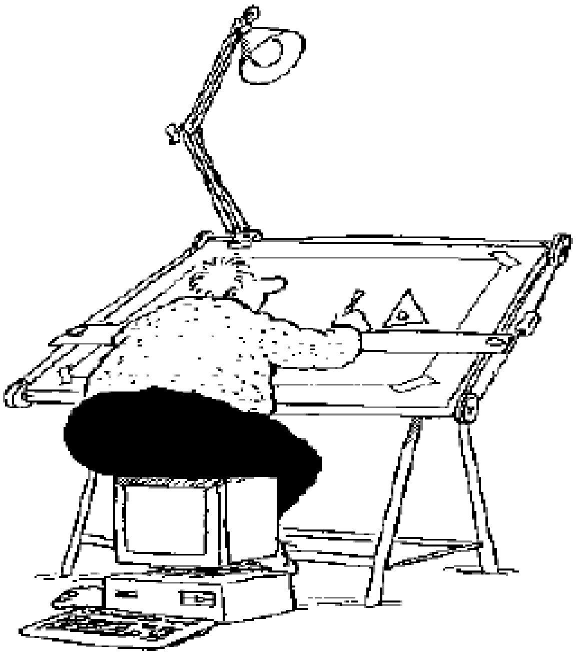


Hardware-software co-design flow

- Specification and modeling (co-specification, functional co-simulation).
- High-level co-synthesis:
  - ✓ Architecture selection.
  - ✓ Components allocation (processing elements, storing elements and communication elements).
  - ✓ Tasks hardware-software mapping.
  - ✓ Scheduling of the several tasks.
- Low-level co-synthesis:
  - ✓ Hardware synthesis.
  - ✓ Software compilation and code generation.
  - ✓ Interface synthesis.
- Integration, simulation, prototyping, fabrication.
- All steps are supported by CAD tools.


Specification, modeling and functional simulation
Architecture selection (components allocation)
Hardware-software mapping and sceduling
Hardware synthesis
Interface synthesis
Software compilation
Hardware-software integration & co-simulation
Prototyping development and testing
System fabrication

Computer-aided design (CAD)



COMPUTER AIDED DESIGN

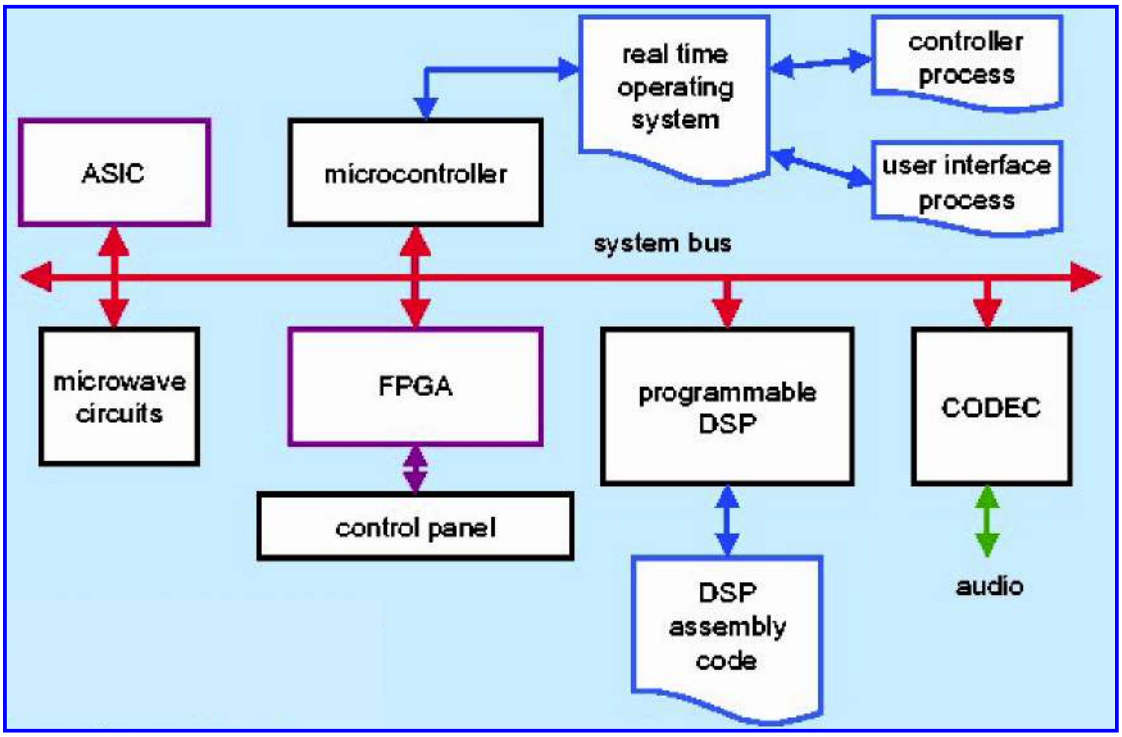
Specification and modeling

- For the specification the heterogeneous nature of embedded systems, has to be taken into account.
- There are no global modeling methods, except for restricted domains:
  - ✓ Dataflow models.
  - ✓ Finite state machines.
  - ✓ Petri nets, etc.
- Specific optimizations are often required (data flow graph, FSM reduction etc.).
- Several specification languages (SDL, UML, Matlab, C) can be used, based on several specification models.
- Executable co-specification for early validation of the system.

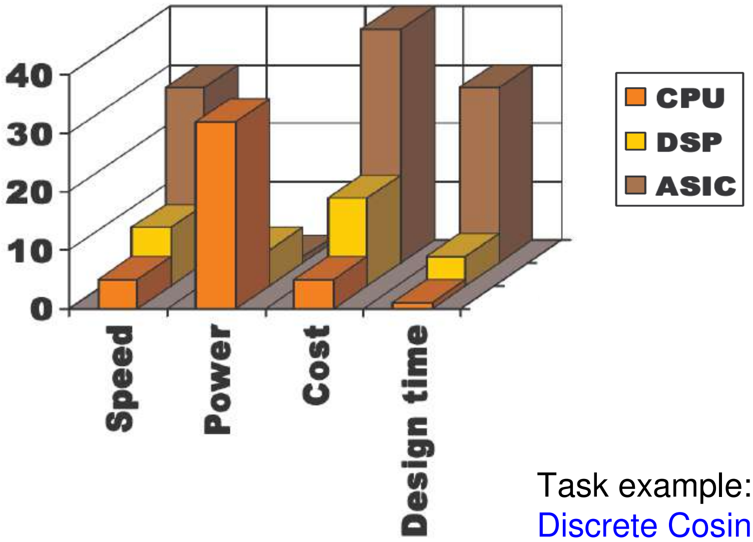
Synthesis of embedded systems

- Co-synthesis is a complex task:
  - ✓ Components selection (architecture definition) and hardware-software mapping.
  - ✓ Tasks scheduling.
  - ✓ Software synthesis and code generation.
  - ✓ Hardware synthesis.
  - ✓ Interface and communication synthesis
- Optimizations:
  - ✓ With respect to the design objectives and constraints.
  - ✓ Often the objects are conflicting.
- Design space exploration in order to generate alternative implementations.

Architecture of an embedded system

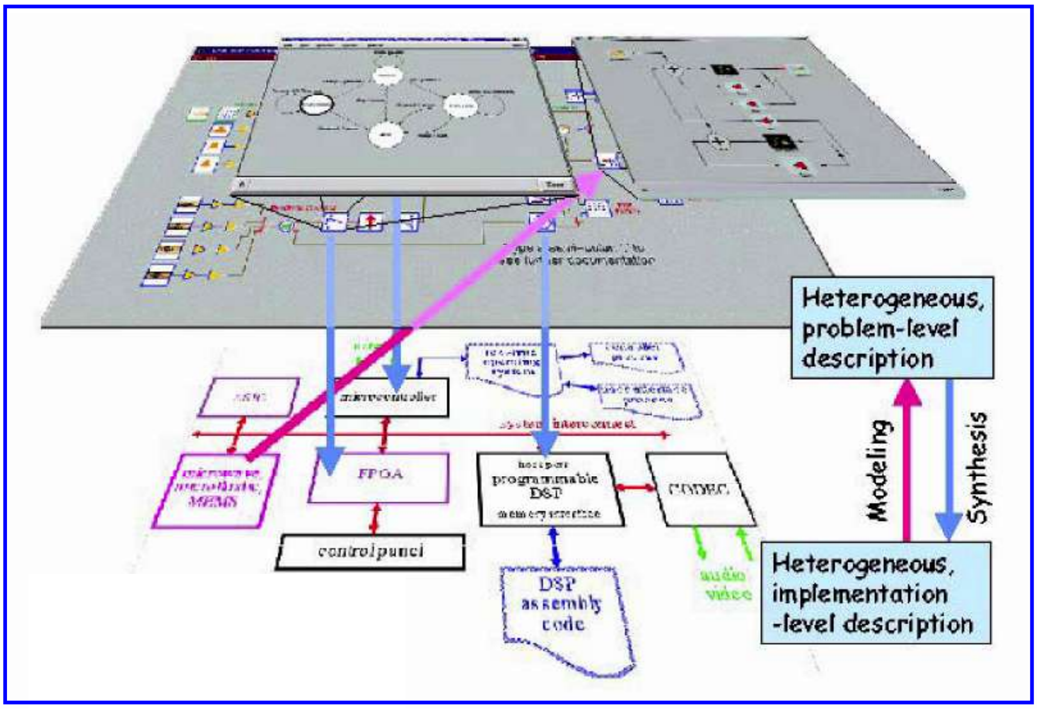


Design space exploration



Task example:
Discrete Cosine Transform

Hardware-software mapping



Hardware-software mapping (cont’d)

- Speed-up the software execution:
  - ✓ By migrating software functions to dedicated hardware.
- Reduce cost of hardware implementation:
  - ✓ By migrating hardware functions to software.
- Guarantee real-time constraints:
  - ✓ By migrating the timing-critical portion to ASIC.

Hardware-software co-simulation

- The basic problem of co-simulation is how to simulate hardware and software together so that simulation to be fast and accurate.
- The existing simulation techniques (hardware and software) have to be extended to combine simulation of hardware and software components.
- Different simulation platforms can be used.
- Software runs fast while hardware simulation is relatively slow. So, the problem is how to run the system simulation as fast as possible and keep the two domains synchronized.
- Slow models provide full details and produce accurate results while fast models do not produce enough timing information and the simulation is not accurate.

Conclusions

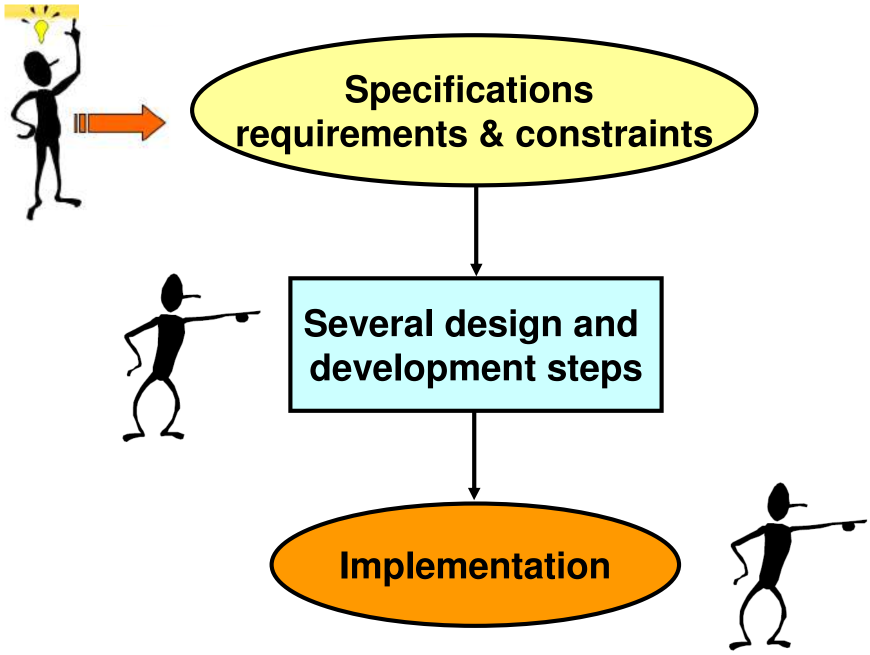
- Embedded systems are dedicated and application-specific, contains at least one programmable component, requires continuous interaction with the environment in real time and must meet several constraints.
- This nature of today’s embedded systems faces developers and engineers with new problems when it comes to specifying, simulating, designing and optimising such complex systems.
- Implementations are typically comprised of general purpose or application-specific programmable components, dedicated processing components, communication and memory modules.
- The design of embedded systems is driven by several constraints: performance, power consumption, size and weight, cost, safety and security, reliability, time to market.
- Optimization involves the simultaneous consideration of these incomparable and often competing objectives.
- As a consequence, much design effort and advanced CAD tools are necessary in order to handle the complexity of today’s embedded systems & applications.

2. System specification and modeling

University of Thessaly

Department of Computer and Communication Engineering

Specifications in embedded systems



Specifications requirements & constraints

Several design and development steps

Implementation

Specifications in embedded systems (cont'd)

- Requirements and constraints: informal description of what the customer wants.
- Specification: precise description of what design team should deliver.
- Requirements and constraints analysis phase links the customer with the designers.

Types of specifications, requirements & constraints

- Functional: input-output relationships.
- Non-functional:
  - ✓ Timing.
  - ✓ Power consumption.
  - ✓ Production cost.
  - ✓ Physical size, weight.
  - ✓ Time-to-market.
  - ✓ Safety requirements.
  - ✓ Environmental aspects, reliability.

Proper specifications, requirements & constraints

- Correct.
- Unambiguous.
- Complete.
- Verifiable: we will be capable to check if the specification, requirement or constraint is satisfied in the final system.
- Consistent: specification, requirements or constraints do not contradict each other.
- Modifiable: can be updated easily.
- Reasonable: know why each specification/constraint exists.

Setting requirements and constraints

- Customer interviews.
- Comparisons with competitors.
- Feedback from sales and marketing departments.
- Experience from prototypes and similar products.
- Design a product for someone like you….

Setting specifications

- A complete specification captures non-functional requirements (speed, power, cost, size) and the behavior of the system by providing:
  - ✓ Relation between inputs and outputs.
  - ✓ Possibly internal states.
  - ✓ Algorithm for the system functionality.
- The design team must have the capability to verify the correctness of the specification and to compare the specification with the implementation.
- Basic specification styles:
  - ✓ Textual
  - ✓ Graphical
  - ✓ Mixed

System specifications properties

- Specifications can be formulated in:
  - ✓ Natural language (informal).
  - ✓ Specification languages or models (more detailed).
- A specification language or model have to be:
  - ✓ able to express the basic properties and basic aspects of the system behavior in a clear manner.
  - ✓ able to check the system requirements and to ensure the synthesis of an efficient system implementation.
- Depending on the particularities of the system or parts of the system, adequate languages or models have to be chosen.
- The specification language or model has to contain the appropriate constructs (textual or graphical) in order to express the system's functionality and requirements.

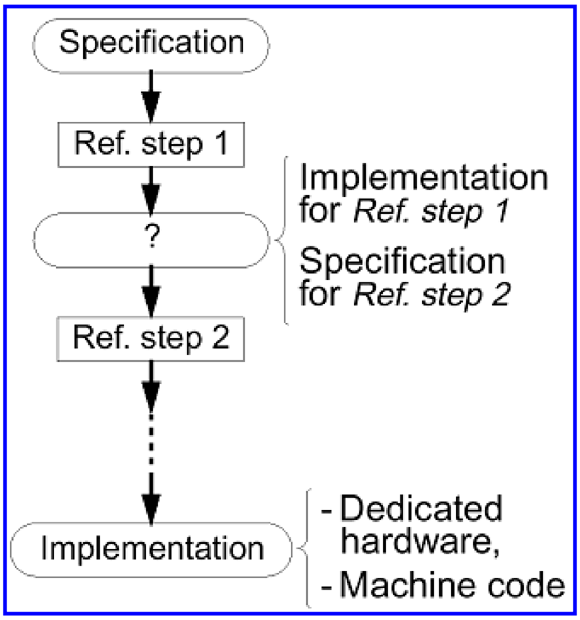
Specifications and refinement

- The design process consists of a sequence of steps: each step performs a transformation from a more abstract description to a more detailed one.

- A design step takes a specification (model, code etc.) of the design at a level of abstraction and refines it to a lower one.

From specification … to … implementation

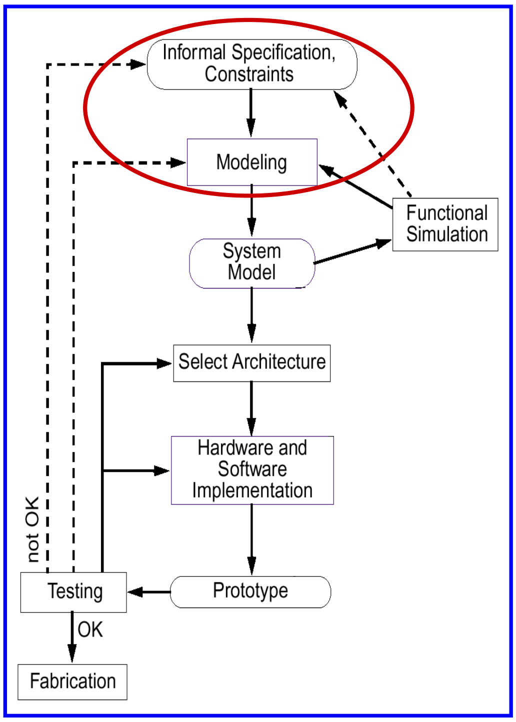
The designer gets a specification (behavior description and other properties) as an input and finally has to produce an implementation, after a sequence of refinement steps.



Simplified design flow

1. Start from some informal specification of functionality and a set of constraints (time and power constraints, cost limits, etc.)
2. Generate a more formal specification of the functionality, based on some modeling concept (e.g. finite state machines).

   This model can be in Matlab, C, UML.
3. Simulate the model in order to check the functionality. If needed make adjustments.
4. Choose an architecture (µprocessor, buses, etc.) such that the cost limits are satisfied and, you hope, that time and power constraints will be fulfilled.
5. Build a prototype and implement the system.
6. Verify the system: time, power constraints satisfied ?
   - Go back to 4 and choose another architecture and start a new implementation.
   - Or negotiate with the customer on the constraints.

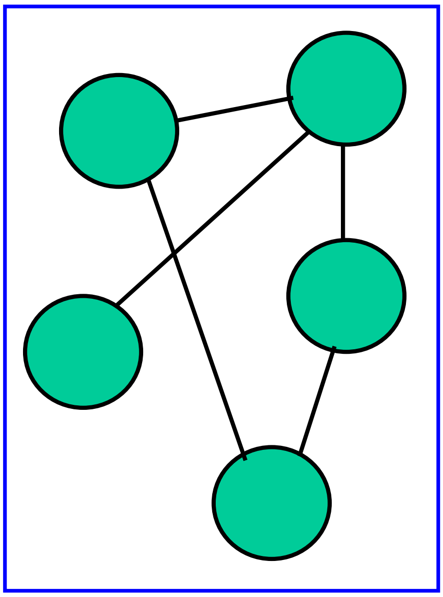


System modeling: use of computation models

- A computation model assists the designer to understand and describe the behavior of a system by providing a “vehicle” to compose the system’s behavior from simpler objects
- A computation model provides a set of objects and rules for composing those objects in order then to be able to formally represent (model) the behavior of our system.
- A system is represented as a set of components, which can be considered as isolated modules (often called processes or tasks), interacting each other and with the environment.
- Usually computation models are based on some kind of graph representation.
- The computation models define the behavior and interaction mechanisms of the system modules.
- The computation models help the designer to formally analyse, estimate some useful parameters, verify (at the high level) the system by using the proper tools.

System modeling: use of computation models (cont'd)

- Thus, computation models usually refer to:
  - ✓ How each module (process or task) performs internal computation.
  - ✓ How the modules transfer information between them (communication).
  - ✓ How they are related in terms of execution order and synchronization.
- Some computation models do not refer to aspects related to the internal computation of the modules, but only to modules interaction.



Order of execution

- Two different approaches for ordering the execution of tasks in computation models.
  - ✓ Data-driven
  - ✓ Control-driven

Data-driven order of execution

The system is specified as a set of processes without any *explicit* specification of the ordering of executions.

↓

The execution order of processes (and the possible parallelism) is determined solely by data dependencies

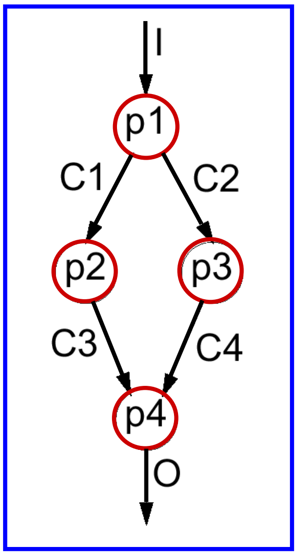
- Typical for many DSP applications

Data-driven order of execution (cont'd)

- Processes communicate by passing data through FIFO channels.
- Each process is blocked until there is sufficient data in the channel

↓

A process that tries to read from a channel waits until data is available.



```
Process p1( in a, out x, out y) {...} ;
Process p2( in a, out x) {...} ;
Process p3( in a, out x) {...} ;
Process p4( in a, in b, out x) {...} ;
channels   I, O, C1, C2, C3, C4 ;
```

```
p1(I, C1, C2);
p2(C1, C3);
p3(C2, C4);
p4(C3, C4, O);
```

It doesn’t matter in which order they are expressed

Control-driven order of execution

- The execution order of processes is given explicitly in the system specification.
- Explicit constructs are used to specify sequential execution and concurrency.

```
module p1:
..........
end module

module p2:
..........
end module

module p3:
..........
end module

module p4:
..........
end module
```

```
run p1;
[ run p2 || run p3];
run p4
```

Here, the order in which they are expressed is essential!

- p1 is started first and has to finish before the starting of p2 and p3;
- p2 and p3 are started in parallel;
- both p2 and p3 have to finish before p4 is started.

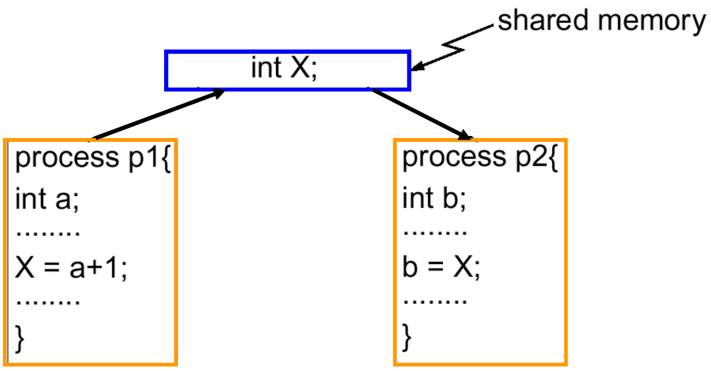
Communication and synchronization

- Processes have to communicate in order to exchange information. Various communication mechanisms are used in the different computation models.
  - shared memory
  - message passing (blocking and non-blocking)

- Synchronization cannot be separated from communication. Any interaction between processes implies a certain degree of communication and synchronization.
- Synchronization: One process is suspended until another one reaches a certain point in its execution.

Shared-memory communication

- Each sending process writes to shared variables which can be read by a receiving process.



```
process p1{
int a;
........
X = a+1;
........
}
```

```
process p2{
int b;
........
b = X;
........
}
```

Private variables:

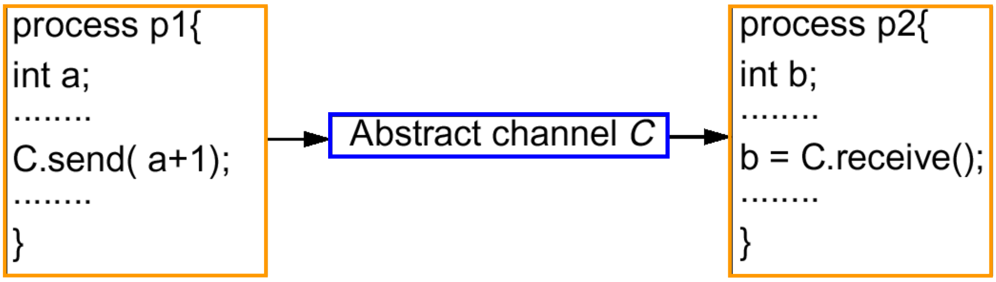
- *a*: local to p1
- *b*: local to p2

Shared variable:

- *X*

Message-passing communication

- Data (messages) are passed over an abstract communication medium called *channel*.



```
process p1{
int a;
........
C.send( a+1);
........
}
```

```
process p2{
int b;
........
b = C.receive();
........
}
```

- This communication model is adequate for specification of distributed systems.

Message-passing communication (cont'd)

Blocking communication

A process which communicates over the channel *blocks* itself (suspends) until the other process is ready for the data transfer.

↓

The two processes have to synchronize before data transfer can be initiated.

Message-passing communication (cont'd)

Non-blocking communication

- Processes do not have to synchronize for communication, **but** additional buffer has to be associated with the channel no messages are to be lost.

- The sending process places the message into the buffer and continues execution.
  The receiving process reads the message from the channel whenever it is ready to do it.

Common computation models

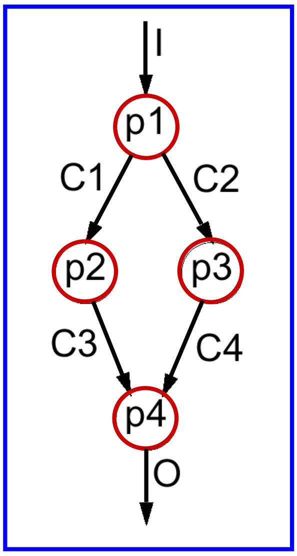
- Different computation models provide different properties.
- We have to choose the right computation model for a particular application domain.
- Computation models commonly used to describe embedded systems:
  - ✓ Dataflow models
  - ✓ Finite state machines
  - ✓ Petri nets

Common computation models (cont'd)

- Most applications are implemented by using control-dominated and data-dominated systems.
- A control-dominated system is one whose behavior consists mostly of monitoring control inputs and reacting by setting control outputs.
- A data-dominated system behavior consists mostly of transforming streams of input data into streams of output data.

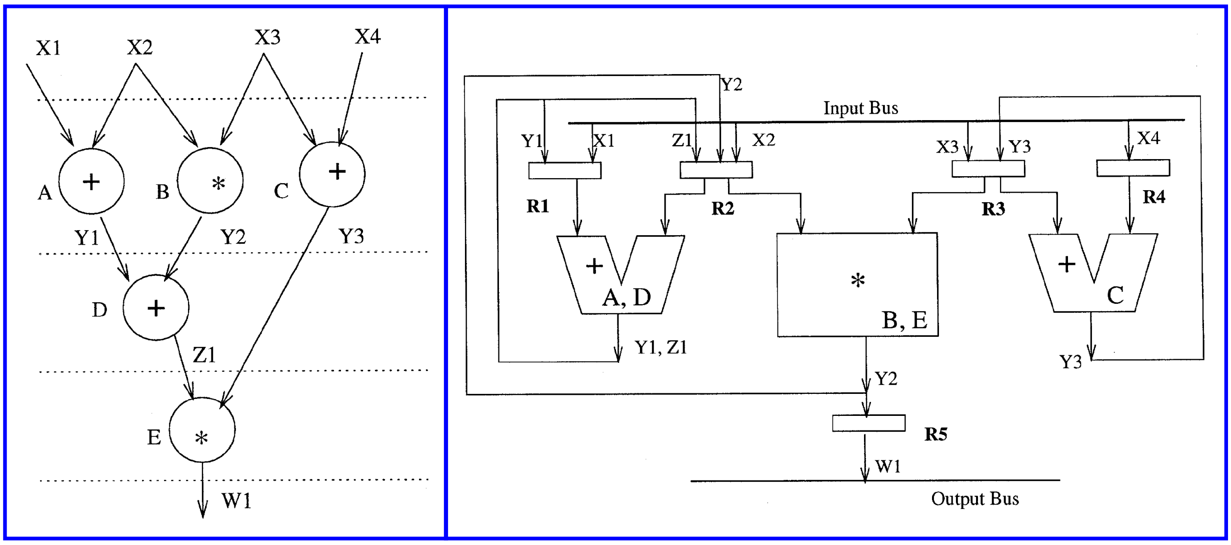
Dataflow models

- Systems are specified as directed graphs where:
  - ✓ Nodes represent computations (processes).
  - ✓ Arcs represent sequences (streams) of data.
- Suitable for signal processing algorithms (encoders, decoders, compressors) that are expressed as block diagrams.
- Typical case of data-driven execution order.
- Commercial tools using dataflow models are: COSSAP (Synopsys), DSP Station (Mentor Graphics), Matlab.



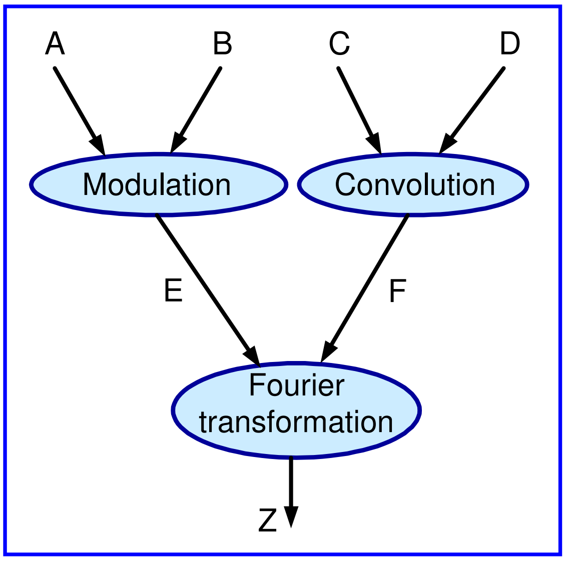
Dataflow models (cont'd)

Example showing a dataflow model of an algorithm and the resulting implementation after synthesis using a CAD tool.



Dataflow models (cont'd)

- Nodes in dataflow models may represent more complex transformations than single arithmetic operations.
- Each node may also include an individual dataflow model.



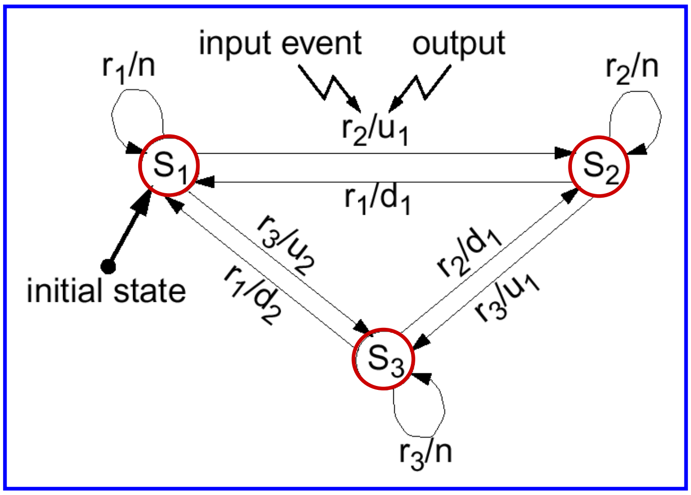
Finite State Machines (FSMs)

- The system is specified by representing its states as well as the transitions from one state to another.
- One particular state is specified as the initial one.
- Finite number of states and transitions.
- Transitions are triggered by input events.
- Transitions generate outputs.
- FSMs are suitable for modeling control-dominated reactive systems, i.e. react on inputs with specific outputs; with not much computation (e.g. communication protocols). There is no possibility to specify computations.
- Commercial tools using FSMs are based on graphical programming: Rational Rose framework (UML-based system), Telelogic SDL Suite (SDL-based tool).

Finite State Machines (cont'd)

Example 1: Elevator controller

- Input events $\{r_1, r_2, r_3\}$
  - r_i: request from floor i.
- Outputs $\{d_2, d_1, n, u_1, u_2\}$
  - d_i: go down *i* floors
  - u_i: go up *i* floors
  - *n*: stay idle
- States $\{S_1, S_2, S_3\}$
  - S_i: elevator is at floor *i*.



Finite State Machines (cont'd)

Implementation in C of the elevator controller

```
#define S1 0
#define S2 1
#define S3 2
#define S4 3
{
int state = S1;
switch (state) {
    case S1: if (r1) { state = S1; n = TRUE; }
             if (r2) { state = S2; u1 = TRUE; }
             if (r3) { state = S3; u2 = TRUE; }
          break;
    case S2: if (r1) { state = S1; d1 = TRUE; }
             if (r2) { state = S2; n = TRUE; }
             if (r3) { state = S3; u1 = TRUE; }
          break;
    ...
```

Finite State Machines (cont'd)

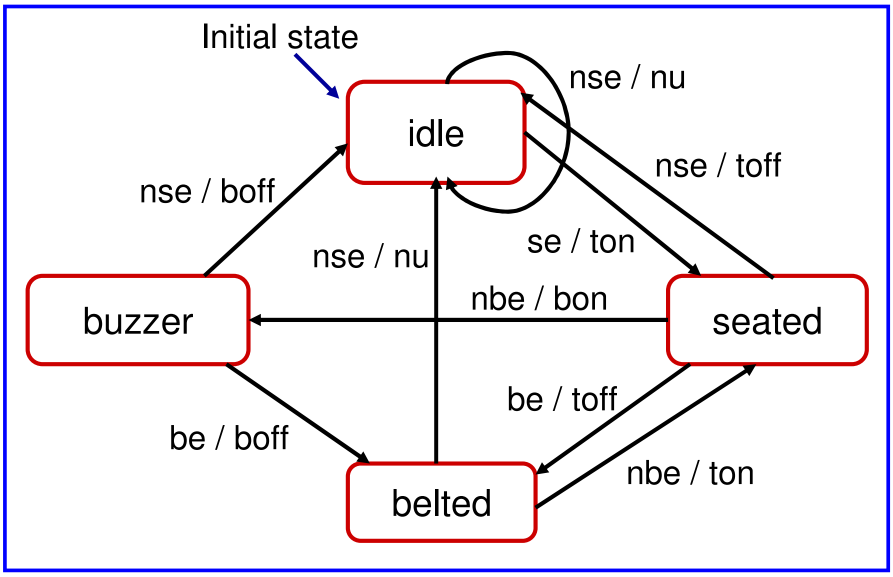
Example 2: **Car belt controller**

Inputs:

nse (no seat)
se (seat)
be (belt)
nbe (no belt)

Outputs:

nu (null)
toff (timer off)
ton (timer on)
bon (buzzer on)
boff (buzzer off)



Finite State Machines (cont'd)

Implementation in C of the car belt controller

```
#define IDLE 0
#define SEATED 1
#define BELTED 2
#define BUZZER 3
{
int state = IDLE;
switch (state) {
    case IDLE: if (no seat) { state = IDLE; }
                 if (seat) { state = SEATED; timer on = TRUE; }
           break;
    case SEATED: if (belt) { state = BELTED; }
                      if (no belt) { state = BUZZER; buzzer on = TRUE; }
                      if (no seat) { sate = IDLE; timer off = TRUE;}
           break;    …………
```

Finite State Machines (cont'd)

- Complex systems tend to have very large number of states. This particularly is the case in the presence of concurrency. This is called state explosion.
- Expressing such a system as a single FSM is very difficult.
- There are two important mechanisms that reduce the size of an FSM model:
  - ✓ Hierarchy
  - ✓ Concurrency
- Using hierarchy and concurrency we only reduce the size of the graphical model; the intrinsic complexity (the number of states of the actual system) cannot be reduced.
- However, the difficulty of realising the model can be drastically reduced.
- In the case of use of the above two mechanisms, we refer to the model as a hierarchical/concurrent FSM or HCFSM.

Finite State Machines (cont'd)

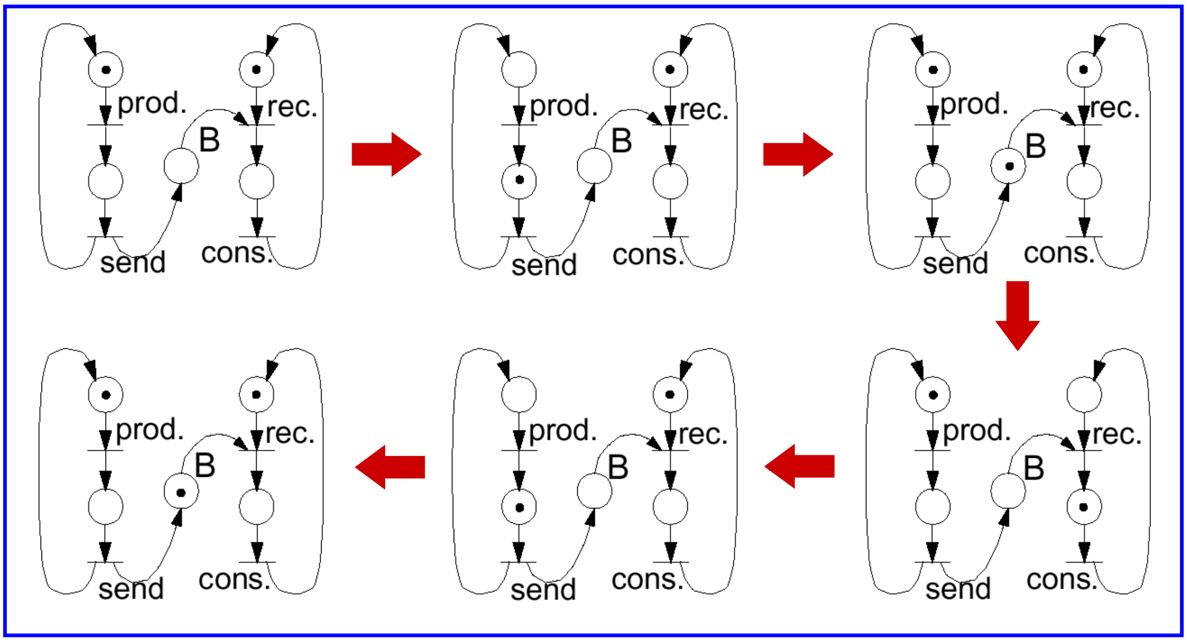
- Hierarchy:
  - ✓ A single state S can represent an enclosed state machine F.
  - ✓ Being in state s means that state machine F is active, and then the system is in one of the states machine F.
- Concurrency:
  - ✓ Two ore more state machines are viewed as being simultaneously active, and then the system is in one state of each parallel state machines simultaneously.
- Another option is the program state machine (PSM) model that extends FSMs to allow use of sequential program code in order to define a state’s actions.

Petri nets

- The system is specified as a sequence of directed graphs.
- There are two kinds of symbols in each graph:
  - ✓ Places: they hold the distributed state of the system expressed by the presence or absence of streams of data in the places.
  - ✓ Transitions: denote the activity of the system.
- The state of the system captured by the marking of the places in which data is present.
- A transition may fire whenever its predecessor places are marked.
- If a transition fires, it removes the data from each predecessor place and adds a mark (data) to each successor place.

Petri nets (cont'd)

Example: A **producer** and a **consumer process** communicating through a buffer, and each process provides acknowledgment for “send” and “receive” actions.



Petri nets (cont'd)

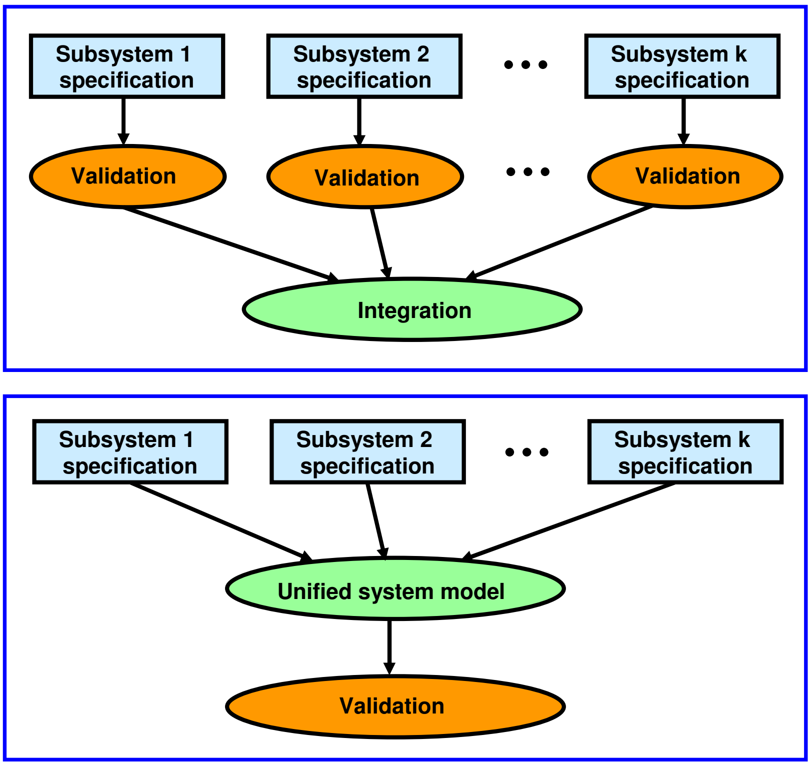
Since petri nets have a limited expressive power we use extended petri nets.

- Timed Petri Nets
  - Transitions have associated times (time intervals).
  - Data streams are carrying time stamps.
  - With timed petri nets we can model the timing aspects.
- Coloured Petri Nets
  - Data steams have associated values.
  - Transitions have associated functions.
  - Coloured petri nets are similar to data flow models.
- In general, petri nets are a mixture of data flow and state-based models.
- Applications of Petri nets: distributed computing, communication networks.

Computation models & specification languages

- A single specification language can be used for the specification of the overall system.
- This does not mean necessarily that we have a homogeneous specification (in terms of computation models).
- It is possible for example to specify in the same programming or HDL language parts of the system as FSM and other parts according to another data-flow model.
- Several languages can be used for system specification:
  - ✓ Specific languages for the hardware and software parts (C and VHDL, for example).
  - ✓ Different languages can be used inside the software and hardware domain, depending on the selected computation model.
  - ✓ Without having decided on the future implementation of a part of our system (hardware or software), different languages can be used depending on the computation model that fits each part of the system.

Multi-language specification


Subsystem 1 specification
Subsystem 2 specification
• • •
Subsystem k specification
Validation
Validation
• • •
Validation
Integration
Subsystem 1 specification
Subsystem 2 specification
• • •
Subsystem k specification
Unified system model
Validation

Specification and validation are supported by specific tools in both cases

Specification languages

- General purpose programming languages (C, C++, Java) or hardware description languages (VHDL, Verilog, SystemC). They do not support, by definition, a certain model of computation.

- Synchronous languages (FSM-based): Esterel.

- Languages for description of networks of communicating processes: UML, SDL (graphical programming).

- Data-flow languages: Silage, Matlab.

- Example: combining SystemC (for subsystems suitable for hardware implementation) and C++ (for subsystems suitable for software implementation) we can use a unified validation environment.

Conclusions

- The design procedure can be viewed as a sequence of refinement steps leading from specification to implementation.
- Specifications are formulated using specification languages and are based on the used computation models.
- Specification languages and computation models are based on specific set of rules and syntactical constructs.
- The chosen language and computation model have to contain the appropriate rules and constructs in order to express the system's functionality and requirements, and to ensure efficient system implementation.
- Different specification languages and computation models can be used depending on the nature of each part of the system or each application domain.
- Interesting aspects of a computation model are: execution order, communication and synchronization.

3. Detailed design flows, synthesis and estimation of embedded systems

University of Thessaly

Department of Computer and Communication Engineering

3a. System-level synthesis of embedded systems

3b. System estimation

3c. Low-level synthesis: Hardware synthesis

3d. Low-level synthesis: Software generation

3e. Power optimization in embedded systems

3a. System-level synthesis

Labros Bisdounis, Ph.D.

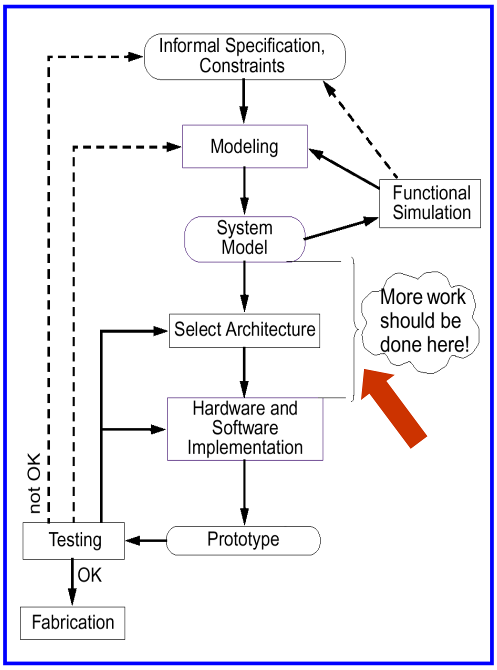
University of Thessaly

Department of Computer and Communication Engineering

Simplified (traditional) design flow

1. Start from some informal specification of functionality and a set of constraints (time and power constraints, cost limits, etc.)
2. Generate a more formal specification of the functionality, based on some modeling concept (e.g. finite state machines).

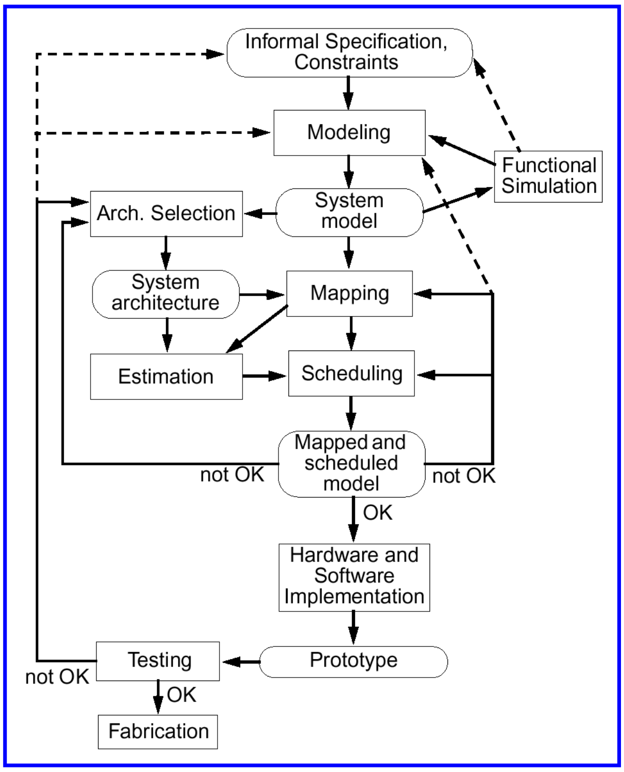
   This model can be in Matlab, C, UML.
3. Simulate the model in order to check the functionality. If needed make adjustments.
4. Choose an architecture (μprocessor, buses, etc.) such that the cost limits are satisfied and, you hope, that time and power constraints will be fulfilled.
5. Build a prototype and implement the system.
6. Verify the system: time, power constraints satisfied ?
   - Go back to 4 and choose another architecture and start a new implementation.
   - Or negotiate with the customer on the constraints.


Informal Specification, Constraints
Modeling
Functional Simulation
System Model
Select Architecture
More work should be done here!
Hardware and Software Implementation
not OK
Testing
Prototype
OK
Fabrication

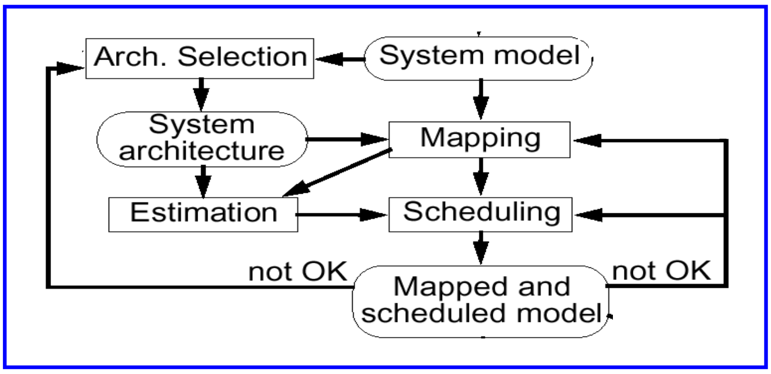
Modified design flow

What is the essential difference:

- It is the inner loop which is performed before the effective hardware/software implementation. This loop can be performed several times (*design space exploration*). Different architectures, mappings, schedules explored, before the prototyping/implementation.
- We get highly optimised good quality solutions in short time. We have a good chance that the outer loop, including prototyping, is not repeated.



System-level synthesis



- Architecture selection: defines a system-level view of the architecture by selecting a set of processing, storage & communication elements along with the system topology.

- Mapping (partitioning): distributes the functionality captured by the specification among the allocated system components.

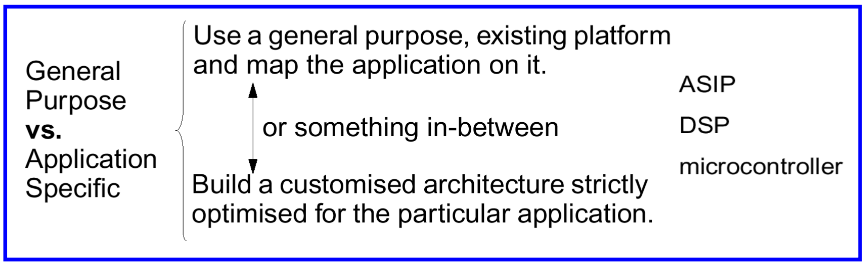
- Scheduling: determines activation times and priorities for processes, so that the constraints are satisfied.

Architecture selection

- Architecture selection decides on the kind and number of components used for implementation of the system, and on their interconnection topology.

- Three types of components are allocated:
  - ✓ Processing elements: microprocessors, microcontrollers, ASIPs, ASICs, FPGAs.
  - ✓ Storing elements: memories, registers.
  - ✓ Communication elements: buses.

Architecture selection (cont'd)



General Purpose **vs.** Application Specific

Use a general purpose, existing platform and map the application on it.

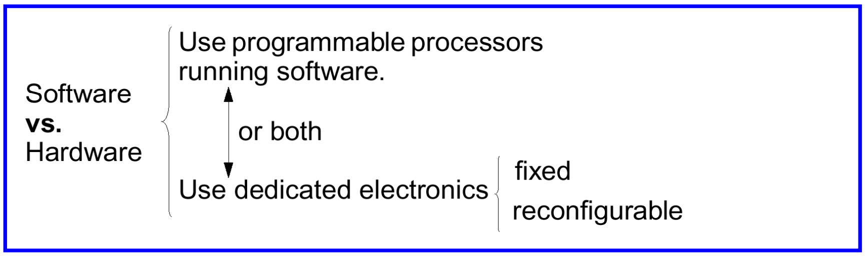
or something in-between

Build a customised architecture strictly optimised for the particular application.

ASIP

DSP

microcontroller



Software **vs.** Hardware

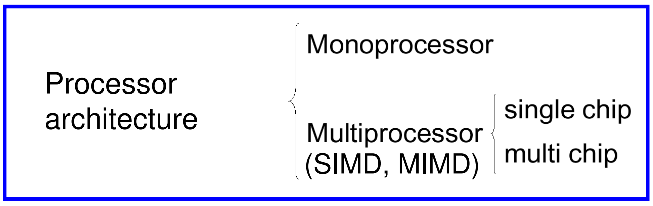
Use programmable processors running software.

or both

Use dedicated electronics

fixed

reconfigurable



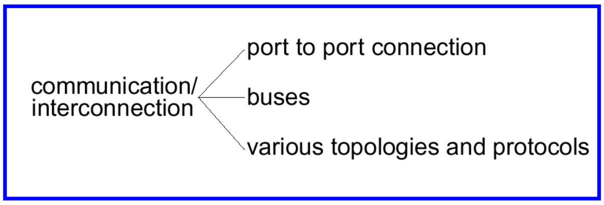
Processor architecture

Monoprocessor

Multiprocessor (SIMD, MIMD)

single chip

multi chip



communication/ interconnection

port to port connection

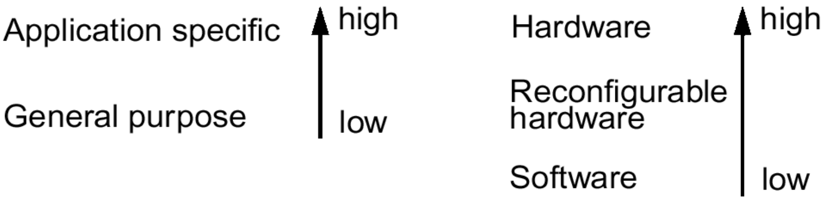
buses

various topologies and protocols

Architecture selection (cont'd)

The trade-offs:

- Performance (high speed, low power consumption)



Application specific

General purpose

high

low

Hardware

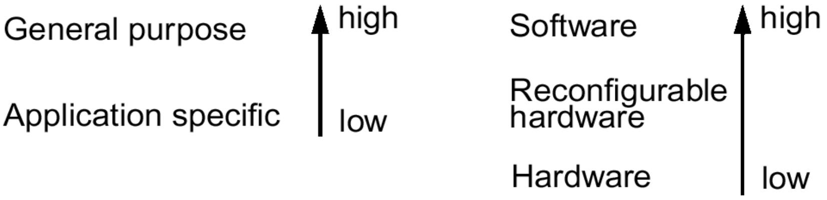
Reconfigurable hardware

Software

high

low

- Flexibility (how easy it is to upgrade or modify)



General purpose

Application specific

high

low

Software

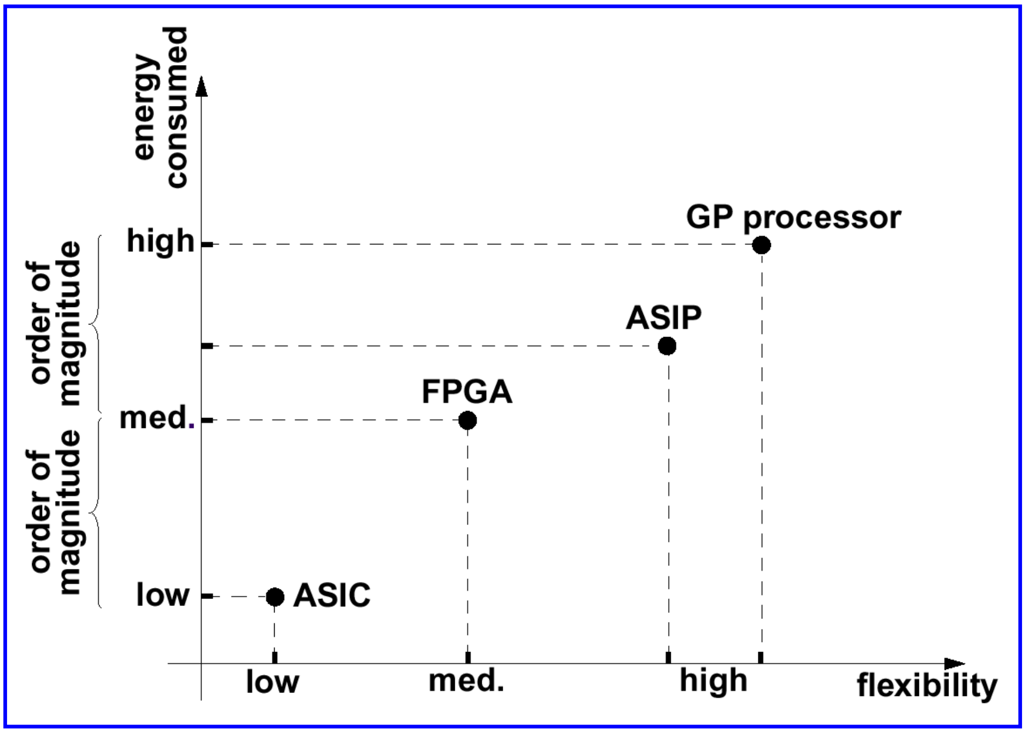
Reconfigurable hardware

Hardware

high

low

Architecture selection (cont’d)



Architecture selection (cont’d)

- General purpose processors: neither instruction set nor microarchitecture or memory system are customized for a particular application or family of applications

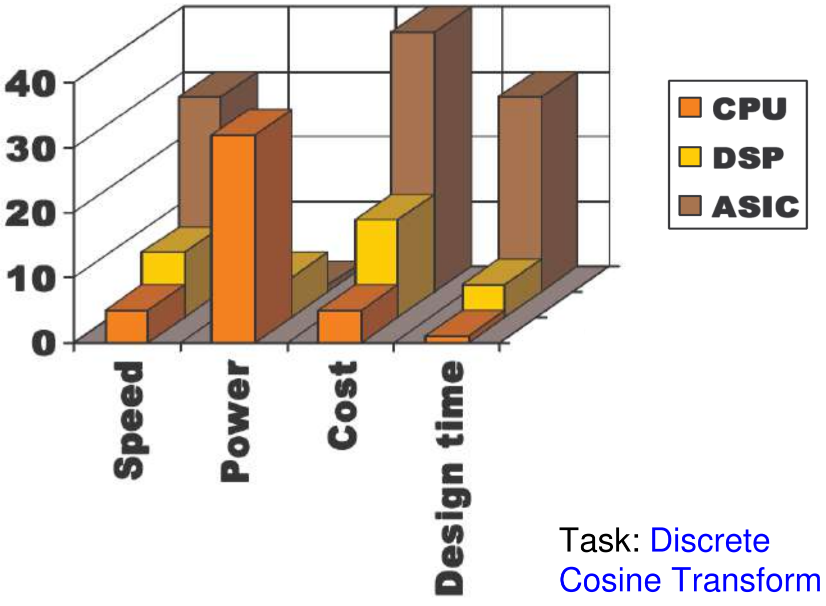
- ASIPs (Application Specific Instruction-set Processors):
  - ✓ Instruction set, microarchitecture and/or memory system are customized for an application of family of applications.
  - ✓ What results is better performance and reduced power consumption.

Architecture selection (cont'd)

- Skills and experience are very important at architecture selection (it is not a straightforward procedure).
- Designer interaction and automatic design space exploration should work together.
- Design space exploration and estimation are based on:
  - ✓ An initial architecture model which should capture the essential features of the class of architectures which have to be taken into consideration.
  - ✓ A component library containing processors, buses, memory modules, and also models corresponding to peripherals (reusable designs).

Architecture selection (cont'd)

Design space exploration example



Task: Discrete Cosine Transform

Hardware-software partitioning

- During the partitioning step we decide what will be executed on programmable processors (software components) and what will be implemented in hardware (ASICs, FPGAs), and we distribute the functionality captured by the specification among the allocated system components.

- Partitioning is strongly related to architecture selection and contains two main steps:
  - ✓ Final allocation of the selected components for the implementation of the system.
  - ✓ Binding: assignment of functions to components (divide the behavior of the system between allocated components) .

Hardware-software partitioning (cont'd)

- The set of factors that have to be taken into account during the hardware-software partitioning of a system are closely tied to the design goals.
- Some of the considerations that can be taken into account are:
  - ✓ Performance constraints (speed and power): functions that have a great impact on the overall performance of the system may need to be implemented in hardware.
  - ✓ Implementation cost: if hardware resources can be shared among functions, it may also be necessary how the partitioning impacts that sharing.
  - ✓ Flexibility: sometimes a software implementation is desired so that the function or algorithm can be easily changed.
  - ✓ Nature of computation: a function may be suitable for either hardware or software implementation. For example computations which can achieve a high degree of parallelism may be better suited for hardware.
  - ✓ Communication: the overhead of communication between hardware and software components has a significant impact to the overall performance.

Hardware-software partitioning (cont'd)

- The hardware-software partitioning is based on metric values derived from profiling, static analysis of the specification and cost estimation.
- Profiling of the executable specification is used to obtain information such as number of branches, loop counts, instruction frequencies, and thus to find how critical is each process in terms of time and energy.
- The time-consuming processes are identified in terms of instruction cycles if an instruction-set simulator is used, and the energy-consuming applications are identified by using proper energy models.
- The hardware-software partitioning is performed through exact methods (e.g. enumeration of the solutions) or through iterative heuristic methods, in which the goal is to find a partitioning solution that reduces several cost functions (e.g. cost, latency, power consumption, memory requirements etc.).

Hardware-software partitioning (cont'd)

- Basic heuristics start from extreme initial solutions: put all processes into the processor (software implementation) which is minimal cost but probably does not meet performance requirements, or put all processes to ASIC (hardware implementation), which gives a maximal performance but also maximal cost.
- Given one of these initial solutions, heuristics select which process to move to other side of the partition to either reduce hardware cost or increase performance, as desired.

Hardware-software partitioning (cont'd)

- More sophisticated heuristics try to construct a solution by estimating how critical a process will be to the overall system performance and choosing a software or hardware implementation accordingly.

- Iterative improvement strategies may move components across the partition boundary to improve the design.

- The whole procedure is guided by cost function(s) that reflects the global quality of the partitioning and a starting solution is modified iteratively, by passing from one candidate solution to another based on evaluations of the cost function(s).

Partitioning example 1

Example for allocating part of the tasks to software and the rest to hardware, based on a cost function computed for each task:

- System constraints (Ci): maximum execution time (T), maximum energy consumption (E) and maximum memory space (M) allowed for each task.

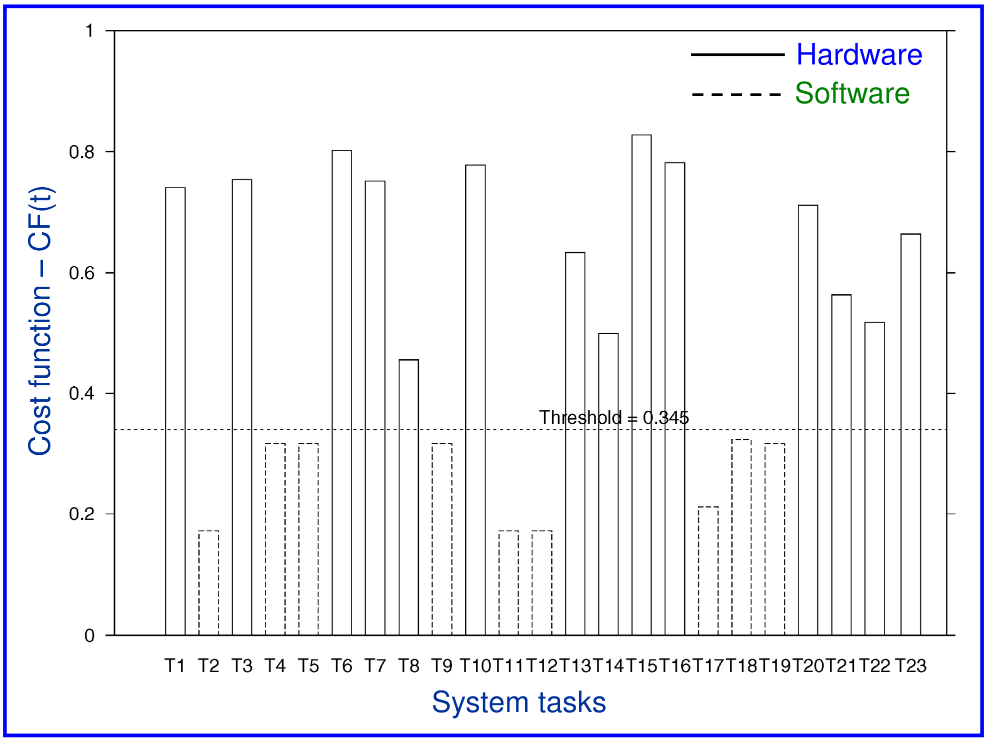
- Estimated values of parameters for each task (Ci(t)): T(t), E(t), M(t).

- The cost function can have the following format:

$$\boxed{\mathrm{CF}(t) = \sum_i \underset{\substack{\uparrow \\ \text{weight factors}}}{k_i} \times \frac{C_i(t)}{C_i} + \sum_i m_i \times \underset{\substack{\uparrow \\ \text{correction terms}}}{f}(C_i, C_i(t))}$$

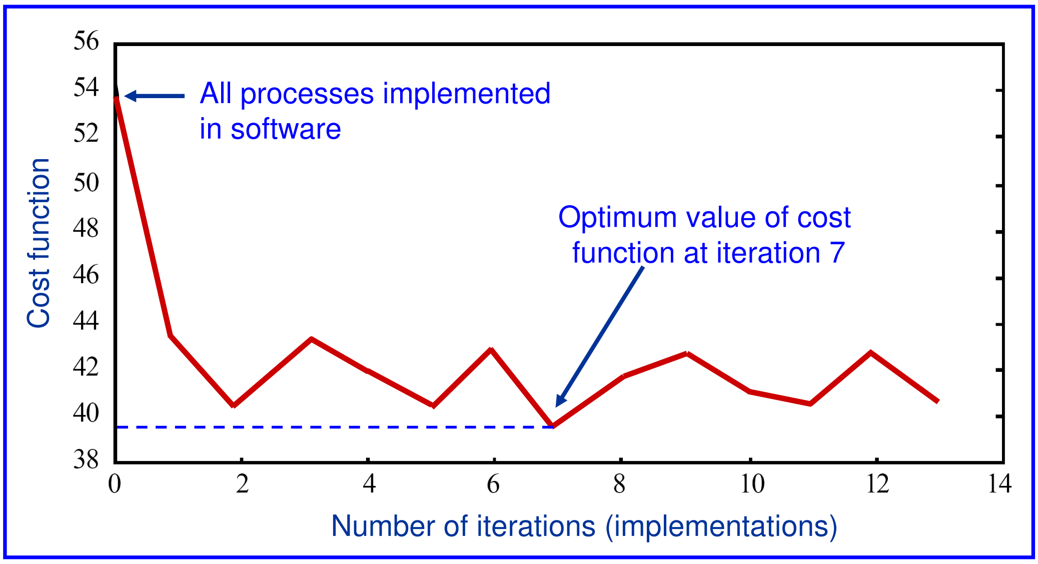
- The threshold value (hardware-software boundary) is obtained by the designer after taking into account how critical are the system constraints.

Partitioning example 1 (cont'd)



Partitioning example 2

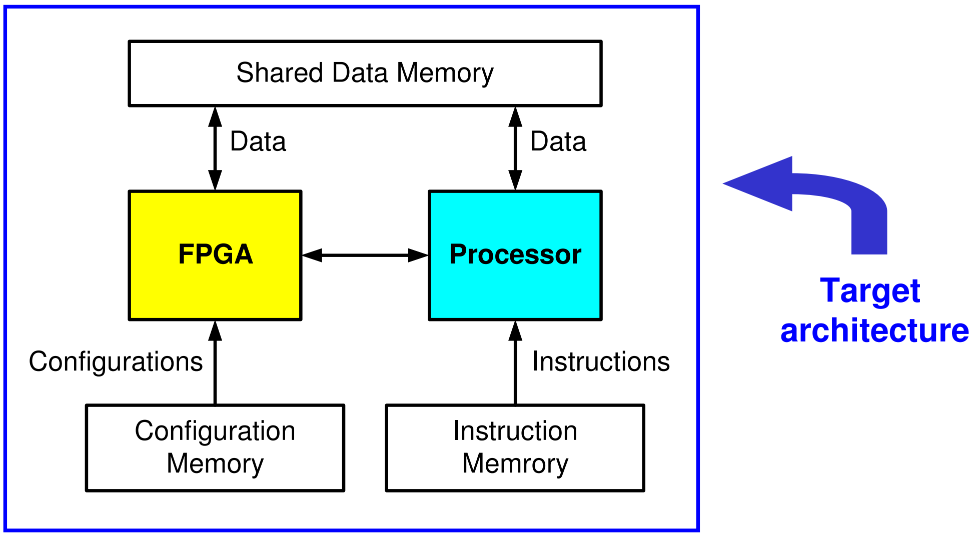
Example for finding the best implementation (partitioning solution) iteratively, based on a cost function that incorporates speed and power and computed for each different implementation:


All processes implemented in software

Optimum value of cost function at iteration 7

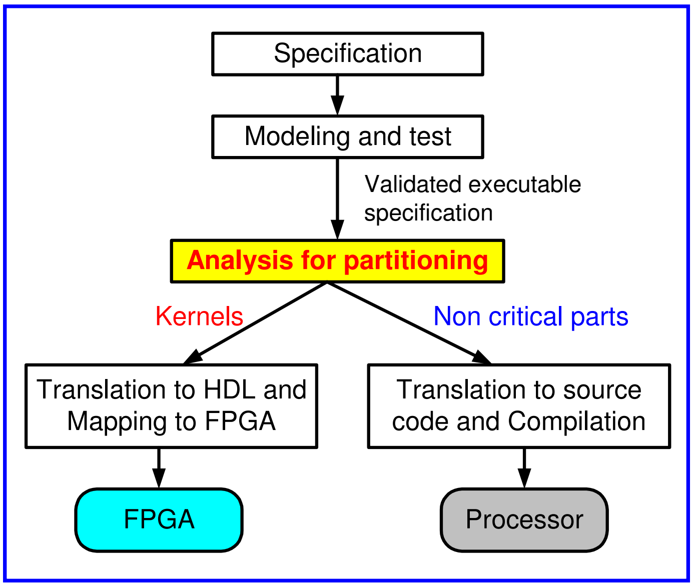
Partitioning example 3

Example for performing partitioning of system's tasks on a given architectural template in order to improve performance:



Partitioning example 3 (cont'd)

Partitioning flow


Specification
Modeling and test
Validated executable specification
Analysis for partitioning
Kernels
Non critical parts
Translation to HDL and Mapping to FPGA
Translation to source code and Compilation
FPGA
Processor

- The computational complexity of basic tasks (represented as the number of instructions executed when an application runs) is obtained by profiling and static analysis.
- Profiling is based on instruction-set simulation and reports the execution frequency of the basic tasks.
- Static analysis calculates the size of the basic tasks.
- We consider as kernels, the tasks which have an instruction count over a user-defined threshold (e.g. tasks contributing more than 10% in the total application instruction count).

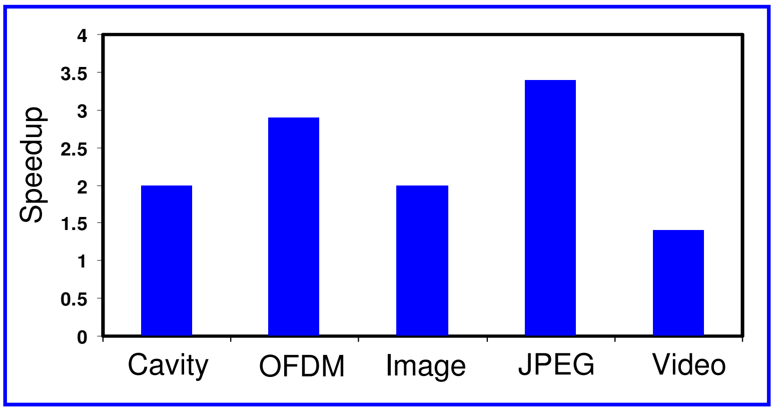
Partitioning example 3 (cont’d)

- Five applications were tested:
  - ✓ Cavity detector (medical image processing application)
  - ✓ OFDM transmitter (wireless communication systems)
  - ✓ Image compression technique
  - ✓ JPEG decoder
  - ✓ Video compression technique

Application	Total size	Kernels size	% size	% instructions	No of kernels
Cavity	12,039	910	7.6	79.8	4
OFDM	15,579	1,440	9.2	61.5	4
Image	12,835	602	4.7	78.8	4
JPEG	10,995	2,534	23.0	71.3	4
Video	24,767	2,477	10.0	51.0	3
Average	-	-	10.9	68.5	-

Partitioning example 3 (cont’d)

- Implementation:
  - ✓ Non-critical parts are executed in an ARM7 RISC processor.
  - ✓ Kernels are implemented on a XCV50 Virtex FPGA.
  - ✓ We achieve significant speedup after executing the kernels on the FPGA, in comparison with execution of the whole application on the processor.



A = $\text{Cycles}_{HW/SW}$ =

$\text{Cycles}_{SW} + \text{Cycles}_{FPGA} + \text{Cycles}_{COM}$

B = $\text{Cycles}_{SW\text{-}ALL}$

Speedup = B / A

Scheduling

- Most of the embedded systems are real-time systems in which the correctness of their behaviour depends not only on the logical results of the computations, but also on the time when the results are produced.

- These systems are time-critical, i.e. the failure to meet time constraints can lead to degradation of the provided service or to a system with wrong operation.

- In addition, embedded systems are made up of concurrent processes: the tasks share resources (e.g. processors) and communicate each other.

- This makes scheduling of tasks a central problem in the design of embedded systems.

Scheduling (cont'd)

- Time constraints are often expressed as deadlines at which tasks have to complete their execution.
  - ✓ Hard deadline: has to met strictly, and if not the system will not operate correctly.
  - ✓ Soft deadline: tasks can be finished after their deadline, although the value provided by completion may degrade with time.

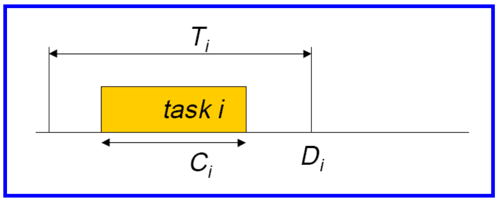
- So, an important property in real-time embedded systems is the predictability that means the possibility to guarantee that deadlines are met as imposed by requirements:
  - ✓ Hard deadlines are always fulfilled.
  - ✓ Soft deadlines are fulfilled to a degree sufficient for the imposed quality of service.

- The question now is: can the given tasks be scheduled on the available resources (processors, custom hardware etc.), so that deadlines are fulfilled ?

Scheduling (cont'd)

- The scheduling problem: Which task has to be executed at a certain moment on a given hardware resource (e.g processor), so that the time constraints are fulfilled.
- A set of tasks is schedulable if, given a certain scheduling policy, all constraints will be completed (which means that a solution to the scheduling problem can be found).
- At least for high-cost and high-demanding real-time embedded systems, it is needed to check off-line, in advance, if the system is schedulable.

Scheduling (cont'd)



- What do we need to know about a task:
  - ✓ Computation time (worst case).
  - ✓ Deadline of task completion.
  - ✓ Regularity of task arrival (periodic tasks with period T and non-periodic tasks with variable period of arrival).
  - ✓ Usually, it is assumed that T = D.
- In addition, we must take into account:
  - ✓ Precedence relations (due to application-dependent execution order or due to data dependencies).
  - ✓ Resource dependencies due to shared recourses (e.g. a shared buffer).

Scheduling policies

- There are three basic scheduling policies:
  - ✓ Static cyclic scheduling (off-line without priority): a table is generated off-line containing activation times for each task. The activation sequence of such table is repeated cyclically .
  - ✓ Time-driven scheduling (off-line without priority): it assigns time slices to processes independent of activation, execution times or data dependencies.
  - ✓ Priority-based scheduling: tasks are activated in response to a certain event (e.g. periodic signal or message for arrival of a new task). In case of conflict (several tasks ready to be executed on the same processor), priorities are considered. Priorities can be assigned statically (fixed, off-line and kept unchanged during the execution) or dynamically (changed during execution).
- The scheduling can be performed in an preemptive manner (a running task can be interrupted in order another task to be executed) or in a non preemptive manner (a task, ones started, may not be stopped).

Static scheduling

- In static cyclic scheduling, the activation times of all tasks and the execution order are generated off-line.
- These activation times determine the behaviour of the system over a period T_{total}. The sequence of activations is repeated in a cyclic manner.
- If all tasks have the same period T then $T_{total} = T$.
- If the tasks have different periods $T_1, T_2, \ldots, T_n$ then:

$$\boxed{T_{total} = \mathrm{LCM}\,(T_1, T_2, \ldots, T_n)}$$

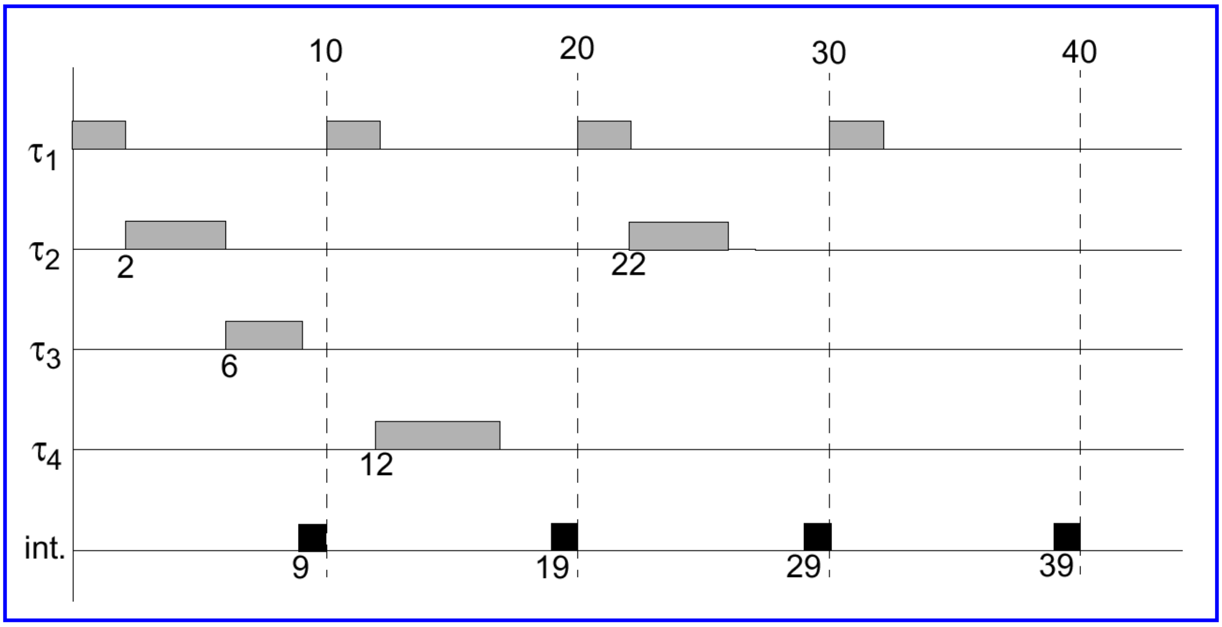
Static scheduling (cont'd)

Example of static cyclic scheduling with four independent tasks executed on the same processor.

	Period=deadline	Worst case comp. time
τ_1	10	2
τ_2	20	4
τ_3	40	3
τ_4	40	5
System management	10	1

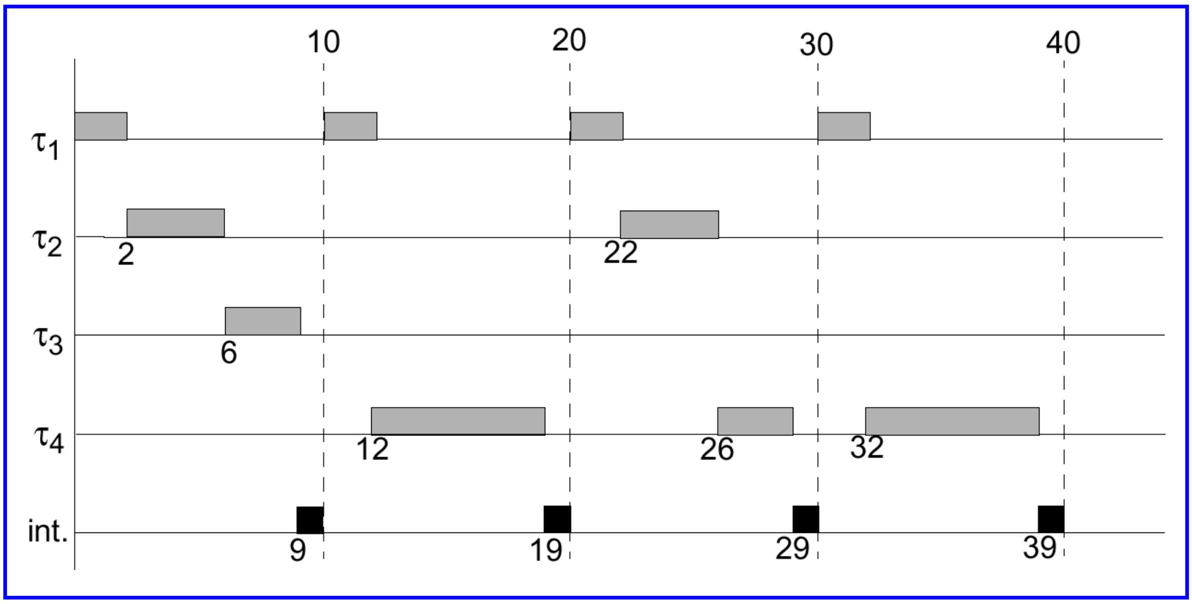
$$T_{total} = \text{LCM}(10, 20, 40) = 40$$

Static scheduling (cont'd)



Static scheduling (cont'd)

Consider the same example as before, but computation time for τ_4 is 17 ⇒ without pre-emption we cannot build a schedule!



Static scheduling (cont'd)

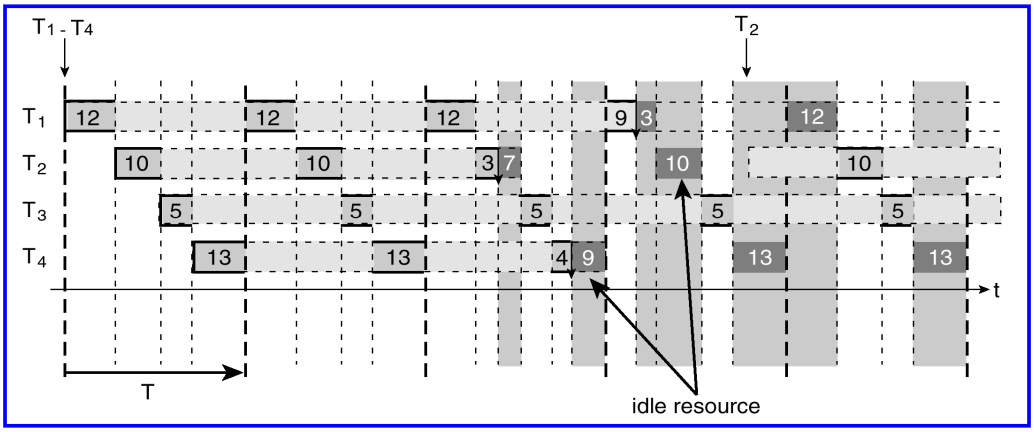
- Advantages of static cyclic scheduling:
  - ✓ High predictability.
  - ✓ Easiness of debugging.
  - ✓ Low execution time overhead (not much to do for the real-time kernel during execution time).
- Disadvantages of static cyclic scheduling:
  - ✓ Low flexibility: quality degrades rapidly if periods and execution times deviate from those predicted, and if new tasks are added a rescheduling is needed.
  - ✓ Sometimes lead to very long total periods.
  - ✓ Tasks have to be manually split in order to fit into available slots.

Time-driven scheduling

- In time-driven scheduling time slices are assigned to the tasks, independent of activation, execution times or data dependencies.
- One of the basic time-driven scheduling strategies is the Time Division Multiple Access (TDMA) strategy which keeps a fixed assignment of time slices to the tasks. This assignment is repeated periodically.
- The main advantages of TDMA are predictability and simplicity. The tasks can be merged in one resource without influencing each other.
- The main limitations of TDMA are efficiency and long total response times.
- It is applicable to communication and data processing.

Time-driven scheduling (cont'd)

- Example of TDMA with 4 independent tasks executed on the same processor.
  - ✓ Assignment of 12, 10, 5 and 13 time units (ms) to tasks T_1, T_2, T_3, T_4.
  - ✓ Total period is then: T = 40 ms.
  - ✓ T_1: execution time 45 ms, response time 129 ms, T_2: execution time 23 ms, response time 95 ms, T_3: execution time 54 ms, response time 426 ms, T_4: execution time 30 ms, response time 111 ms.
  - ✓ At t = 0 all tasks are activated, and at t = 150 ms T_2 is activated again and continues execution at t = 172 ms.

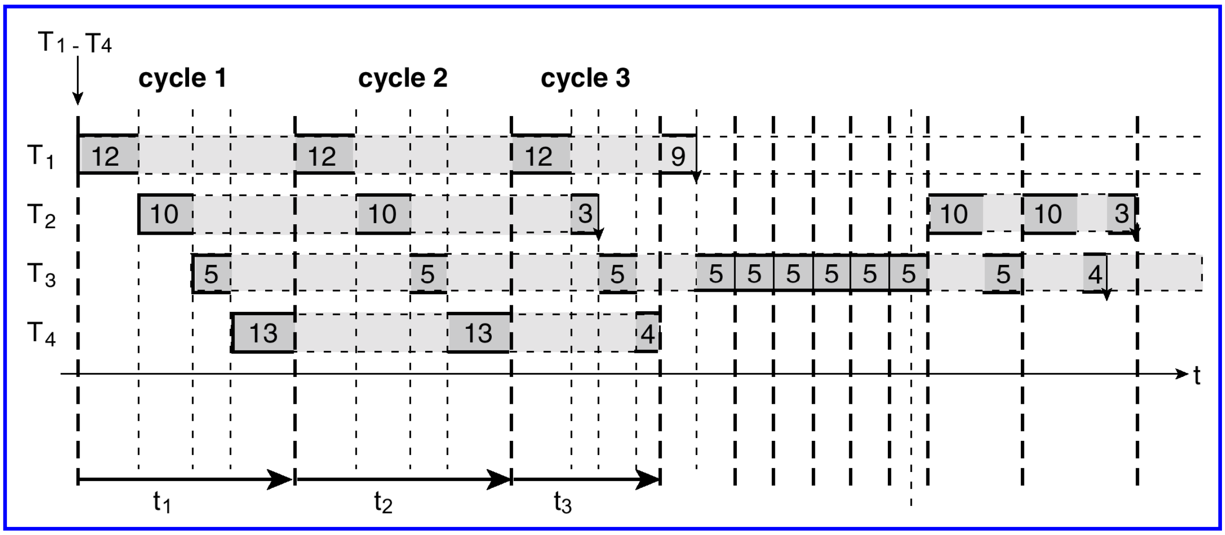


Time-driven scheduling (cont'd)

- A second basic time-driven scheduling strategy is the Round Robin which departs from the fixed time slot assignment and terminates a slot if the corresponding task ends.
- Therefore, slots are erased or shortened, and the cycle time (t_i) of the round robin schedule is time-variant (instead of the constant period in TDMA strategy).
- This strategy avoids the idle times of TDMA and reaches maximum resource utilization.
- On the other hand, the tasks execution is no longer independent (a task can finished earlier if the other tasks are not executed).
- It is also applicable to communication and data processing.

Time-driven scheduling (cont'd)

- Example for Round Robin scheduling: T_1 now ends at t = 113 ms, and more impressively T_3 has a response time of 179 ms.
- Note that T_3 is finished so quickly because the other processes were not executed.

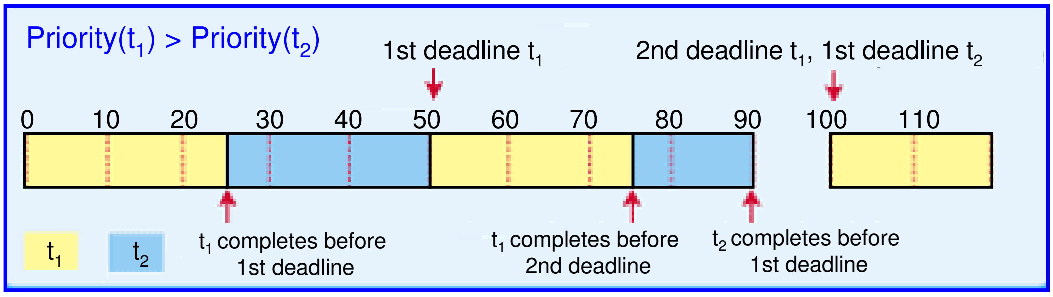


Priority-based scheduling

- In priority-based scheduling the tasks are activated in response to an input event (e.g. periodic signal or message for arrival of a new task). Priorities can be assigned statically or dynamically.
- In a static priority assignment model the tasks are activated by the arrival of a periodic input event.
- A first approach is to assign priority to the task with the shortest period. This is called Rate Monotonic Scheduling (RMS) and is very popular in embedded system design due to its simplicity and ease of analysis.
- A second approach is the Deadline Monotonic Scheduling (DMS) which is an extension of RMS for deadlines smaller than a period.

Priority-based scheduling (cont'd)

- Assume two periodic tasks with periods T_1 and T_2, and each has a deadline that is the beginning of its next cycle. Task t_1 has T_1 = 50ms, and a worst-case execution time of C1 = 25ms. Task t_2 has T_2 = 100ms and C_2 = 40ms.
- Rate Monotonic Scheduling (RMS):



Priority(t_1) > Priority(t_2)

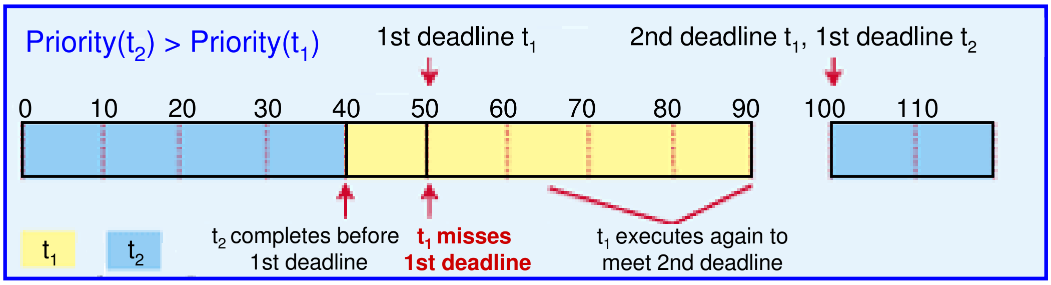
1st deadline t_1

2nd deadline t_1, 1st deadline t_2

t_1 completes before 1st deadline

t_1 completes before 2nd deadline

t_2 completes before 1st deadline



Priority(t_2) > Priority(t_1)

1st deadline t_1

2nd deadline t_1, 1st deadline t_2

t_2 completes before 1st deadline

t_1 misses 1st deadline

t_1 executes again to meet 2nd deadline

Priority-based scheduling (cont'd)

- Static priority algorithms cannot reach maximum resource utilization.
- For Rate Monotonic scheduling:

$$\boxed{\text{Utilization} = \sum_i \frac{C_i}{T_i}}$$

- All deadlines met when Utilization is less or equal to $n\,(2^{1/n} - 1)$

For n=2, $2*(2^{1/2}-1) = 0.83$,
For $n \to \infty$, $n(2^{1/n} - 1) \to \ln(2) = 0.69$

n: number of tasks

- Example: 0.753 < 0.779 (=$3*(2^{1/3} -1)$) => tasks are schedulable

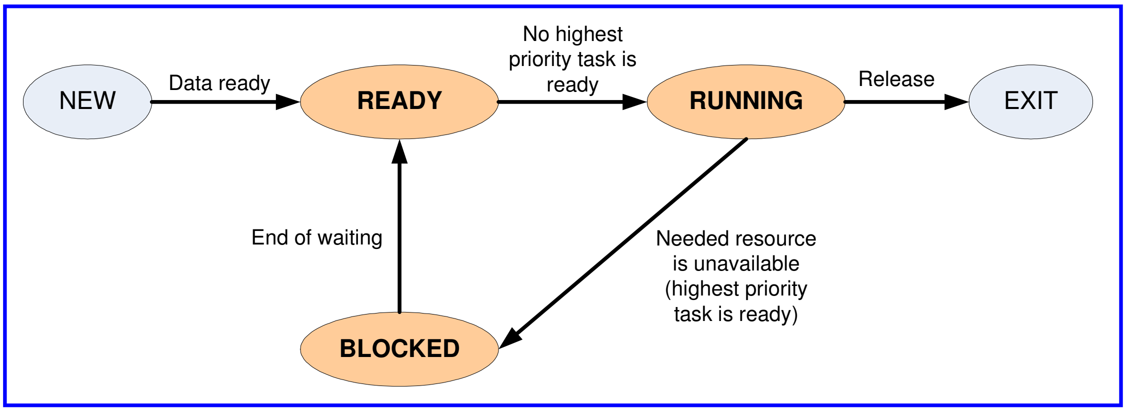
Task	Period (T)	Rate (1/T)	Exec Time (C)	Utilization(U)
1	100 ms	10 Hz	20 ms	0.2
2	150 ms	6.67 Hz	40 ms	0.267
3	350 ms	2.86 Hz	100 ms	0.286

Priority-based scheduling (cont'd)

- To reach a higher resource utilization, the priorities must be assigned dynamically at run time.
- The best dynamic priority assignment strategy is the one that gives the highest priority to the task with the earliest deadline.
- The advantage of the Earliest Deadline First (EDF) scheduling minimizes number of missing deadlines.
- Dynamic priority assignment requires a scheduler process running the assignment strategy and observing the system's state. This process adds overhead in terms of performance and power consumption.

Scheduling and RTOS

- The techniques that are used to schedule the tasks in an embedded system are usually integrated to an RTOS (Real Time Operating System).
- In general, an RTOS is a real-time software that manages the time of a microprocessor to ensure that all time critical events are processed as efficiently as possible.
- A task (process) may be in one of three basic states: running (currently executing), ready to execute or blocked (waiting). A task may not be able to execute until, for example, its data has arrived. Once its data arrives, it moves to the ready state.

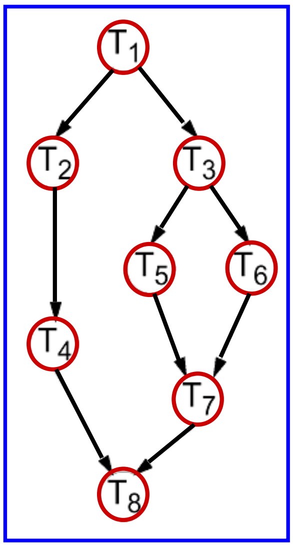

NEW
Data ready
READY
No highest priority task is ready
RUNNING
Release
EXIT
End of waiting
Needed resource is unavailable (highest priority task is ready)
BLOCKED

RTOS state diagram

Scheduling and RTOS (cont'd)

- The RTOS's scheduler chooses the highest priority ready process to run next, according to the applied scheduling policy.
- Unlike, general-purpose operating systems, RTOS generally allow to a process to run until it is pre-empted by a higher priority process.
- General-purpose operating systems often perform time-slicing operations to maintain fair access of all the users of a system, but time-slicing does not provide the control required for meeting strong deadlines.
- This is the main reason for which in real-time embedded systems we use RTOS or specific kernels to perform the scheduling process.

Simple example on system-level synthesis



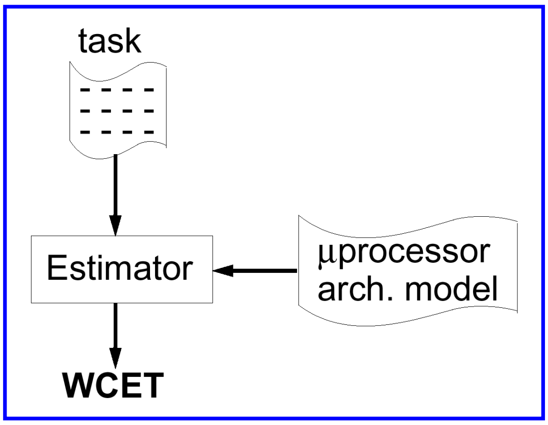
- The system to be implemented is modelled as a data flow graph:
  - ✓ A node represents a task (a unit of functionality activated as response to a certain input and which generates a certain output).
  - ✓ An edge represents a data dependency between two tasks.
- The order of tasks execution is data dependent and priorities between tasks are set by data dependencies.
- Constraints:
  - ✓ Period = 42 time units (the dataflow graph is activated every 42 time units, which means that each activation has to be terminated in time less than 42 units.
  - ✓ Cost limit = 8 (the total cost of the implemented system has to be less than 8).
  - ✓ Non-preemptive scheduling is only supported.

Simple example on system-level synthesis (cont’d)

☞ We decide on a certain μprocessor μp1, with cost 4.

☞ For each task the worst case execution time (WCET) when executed on μp1 is *estimated*.

Task	WCET
T_1	4
T_2	6
T_3	4
T_4	7
T_5	8
T_6	12
T_7	7
T_8	10



task

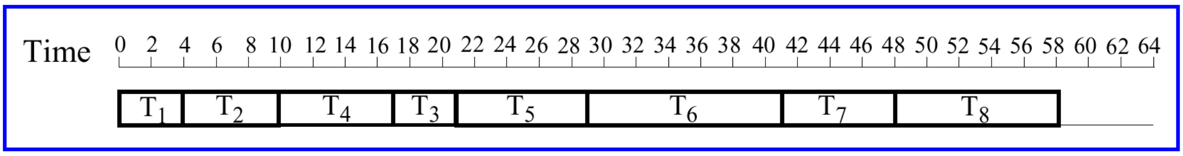
Estimator

μprocessor arch. model

WCET

Simple example on system-level synthesis (cont’d)

☞ A *schedule*:



Using this architecture we got a solution with:

- Execution time: 58 > 42
- Cost: 4 < 8

☞ We have to try with another architecture!

Simple example on system-level synthesis (cont’d)

☞ We look after a mprocessor which is fast enough with cost=15

☞ For each task the WCET, when executed on μp2, is estimated.

Using this architecture we got a solution with:

- Execution time: 28 < 42
- Cost: 15 > 8

☞ We have to try with another architecture!

Task	WCET
T_1	2
T_2	3
T_3	2
T_4	3
T_5	4
T_6	6
T_7	3
T_8	5

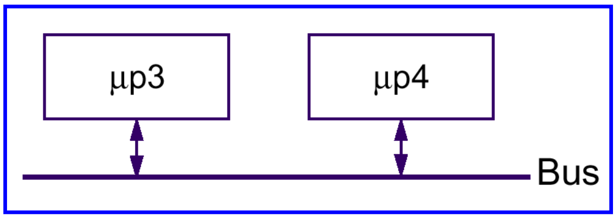
Simple example on system-level synthesis (cont'd)

- Now we have to look to a multiprocessor solution.
  In order to meet cost constraints we try two cheap (and slow) µps:

  µp3: cost 3
  µp4: cost 2
  interconnection bus: cost 1

- For each task the WCET, when executed on µp3 and µp4, is estimated.

Task	WCET	
	µp3	µp4
T_1	5	6
T_2	7	9
T_3	5	6
T_4	8	10
T_5	10	11
T_6	17	21
T_7	10	14
T_8	15	19



Simple example on system-level synthesis (cont'd)

- Now we have to *map* the tasks to processors.

  µp3: T_1, T_3, T_5, T_6, T_7, T_8.
  µp4: T_2, T_4.

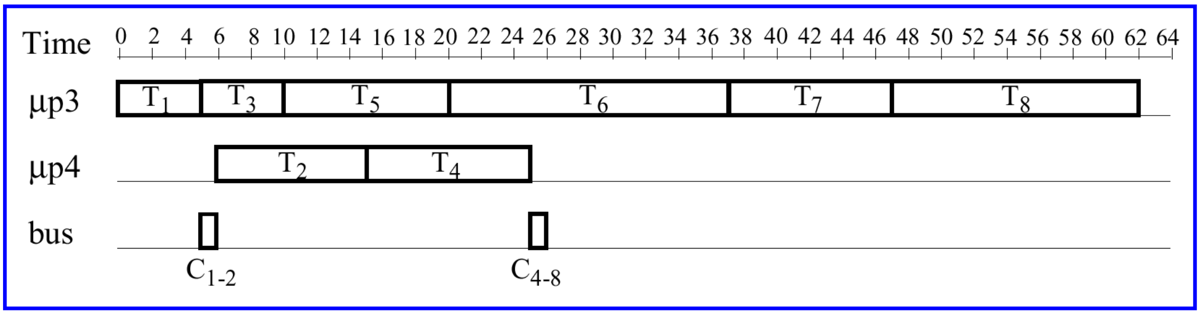
- If communicating tasks are mapped to different processors, they have to communicate over the bus.
  Communication time has to be estimated; it depends on the amount of bits transferred between the tasks and on the speed of the bus.

  Estimated communication times:

  $C_{1\text{-}2}$: 1
  $C_{4\text{-}8}$: 1

Simple example on system-level synthesis (cont’d)

- A *schedule*:



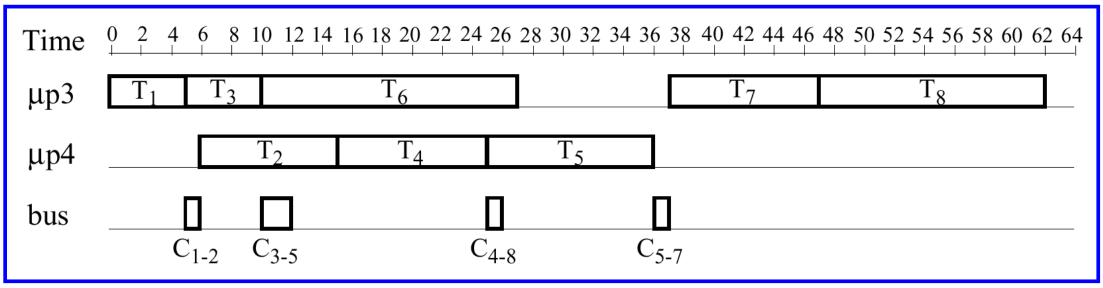
We have exceeded the allowed execution time (42)!

Simple example on system-level synthesis (cont’d)

- Try a new mapping; move T_5 to µp4, in order to increase parallelism.

  Two new communications are introduced, with estimated times:
  C_{3-5}: 2
  C_{5-7}: 1

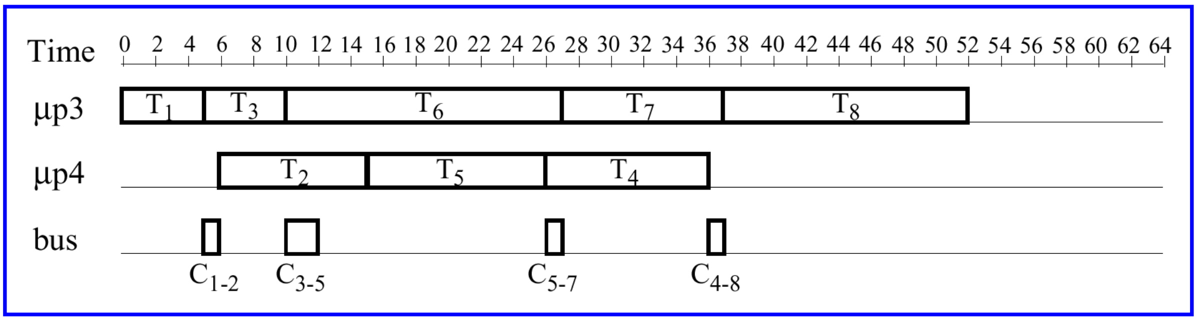
- A *schedule*:



The execution time is still 62, as before!

Simple example on system-level synthesis (cont'd)

☞ There exists a better schedule!



Using this architecture we got a solution with:

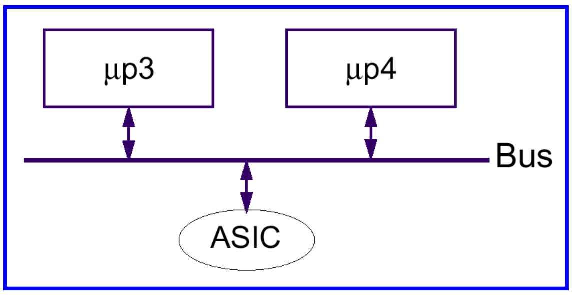
- Execution time: 52 > 42
- Cost: 6 < 8

Simple example on system-level synthesis (cont'd)

☞ Possible solutions:

- Change μprocessor μp3 with a faster one ⇒ cost limits exceeded
- Implement some part of the functionality in hardware as an ASIC

☞ New architecture
Cost of ASIC: 1

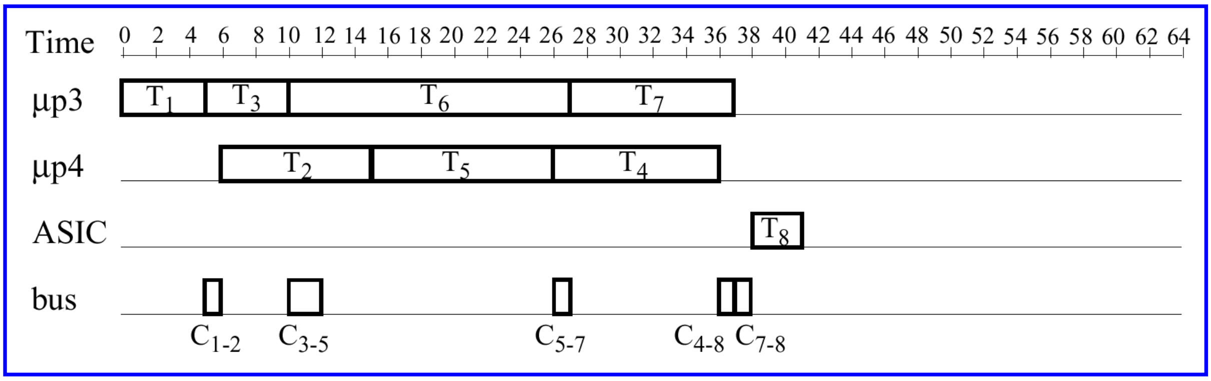


☞ Mapping
μp3: T_1, T_3, T_6, T_7.
μp4: T_2, T_4, T_5.
ASIC: T_8 with estimated WCET= 3

New communication, with estimated time:
$C_{7\text{-}8}$: 1

Simple example on system-level synthesis (cont'd)

☞ A *schedule*:



Using this architecture we got a solution with:

- Execution time: 41 < 42
- Cost: 7 < 8

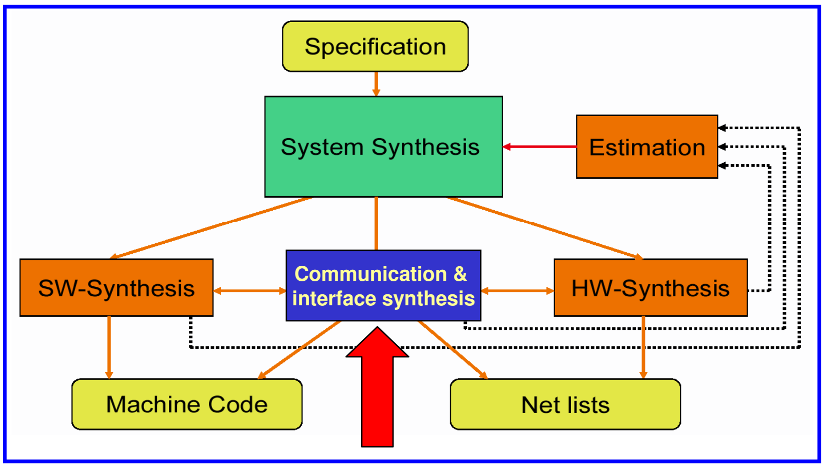
Achievements

- What did we achieve ?
  - ✓ We have selected an architecture.
  - ✓ We have mapped tasks to the processors and ASIC.
  - ✓ We have found a schedule.

- However, nothing has been implemented yet !
  The decisions to be implemented have to be based on simulation and estimation.

- After that we can perform the software and hardware implementation, with a degree of confidence that we will get a correct prototype.

Communication and interface synthesis



- After system partitioning and scheduling we got a set of processes assigned to system components (processors executing the software + hardware components).
- There is a communication between these processes, as well as an interaction of these processes with peripheral devices.
- Interface/communication synthesis has to generate the hardware and software, interconnecting the system components and enables processes to communicate with each other and with peripheral devices.

Communication and interface synthesis (cont'd)

- Communication synthesis, as a top-down design task, is performed in three main steps:

  1. Channel binding (allocation of processes communication to channels).
  2. Communication refinement.
  3. Interface generation.

- After communication synthesis, the initial system specification results into a specification which can be directly synthesized to a physical implementation.

Channel binding

- Communication channels have to be implemented using physical communication components:

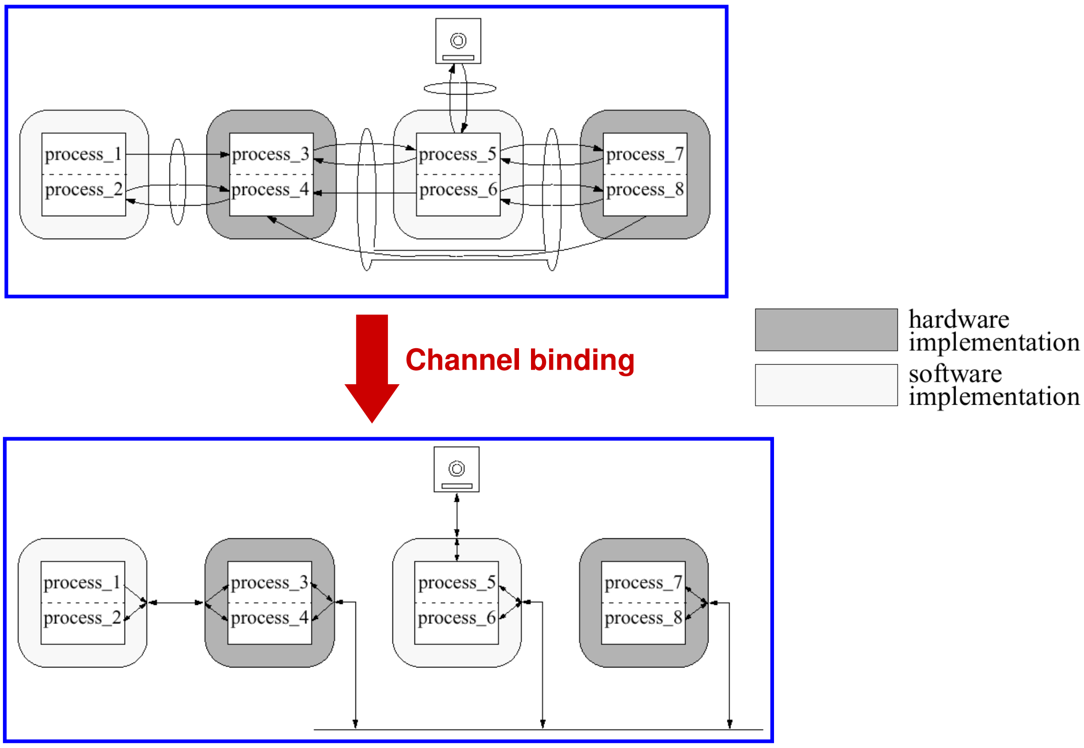
  1. Resources have to be allocated which support communication throughout the system.
  2. Channels have to be partitioned and the resulted groups are allocated to the communication resources.
  3. Messages corresponding to channels in one group are multiplexed on a shared communication component.

Channel binding (cont'd)

- The main criteria used for channel grouping is to avoid bus conflicts and to reduce the total number of connecting wires:

  - Group together channels which don't access concurrently the bus.
  - Group together channels which are accessed by the same processes.
  - Depending on its features, a communication unit can support a certain number of channels to be multiplexed on it, without reducing the communication rate below a required minimum.

Channel binding (cont'd)



Channel binding

▨ hardware implementation

□ software implementation

Communication refinement

- After channel binding the interconnection topology of the system is known and it is determined which channels have to be allocated to a given communication support.

- Communication is still described at the high (system) level, and the specification has to be refined with several details, going down towards the final implementation:
  - The width of the communication lines has to be determined, depending on constraints concerning data transfer rates, number of available pins, and cost.
  - If communication buses are shared, an adequate control strategy has to be decided.
  - A communication protocol has to be defined for each communication link.

Interface generation

In this step the interfaces needed for a correct functionality of the system can be generated (both software and hardware communication components):

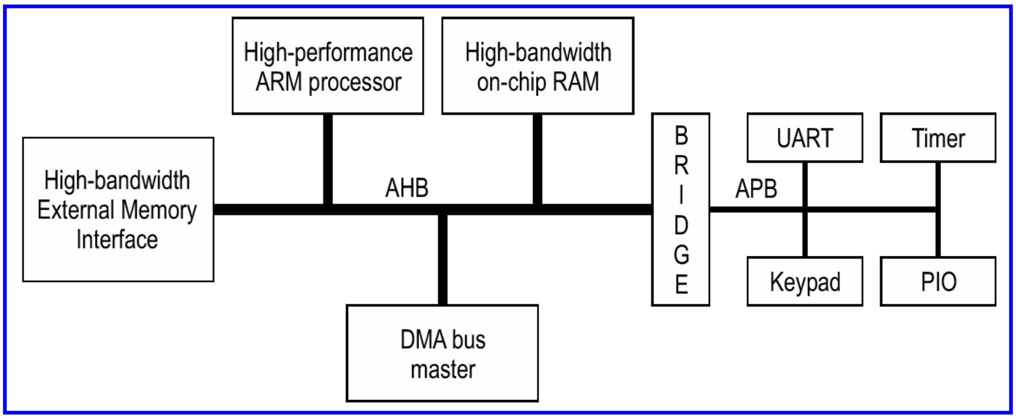
- Access routines inside the processes (in executable code for software processes or in hardware for processes implemented on hardware components).
- Controllers (buffers, FIFOs, arbitration logic) for implementing correct access to the communication support.
- Adapters needed to interface components which use incompatible protocols.
- Device drivers to support access to peripheral devices.
- Low-level support for communication-related tasks (interrupt controller, DMA etc.).

Interface generation (cont'd)

There are several approaches for the generation of interfaces:

- Manually: it is still one of the most applied technique due to many standards and proprietary connection schemes.
- Library-based: predefined and pre-tested device drivers and interface circuits (e.g. ARM AMBA bus specification kit).
- Template-based: predefined code fragments stored in a library that are used to compose a real interface during compilation. This approach provides more flexibility than the library-based approach, but more sources for errors.
- Pattern-based: patterns describe typical connection problems for components, but without predefined implementation. Provides efficient reuse of existing knowledge to create (manually) new device drivers and interface circuits.
- Generator-based: only few approaches exist due to the complexity and heterogeneity of components (e.g. the Cadence VCC tool is a mixed library- and generator-based approach.
- Component-based (IP): components with several standardized interfaces (disadvantage: incompatibility between different IP vendors).

An example: AMBA bus system


High-performance ARM processor
High-bandwidth on-chip RAM
High-bandwidth External Memory Interface
AHB
B R I D G E
APB
UART
Timer
Keypad
PIO
DMA bus master

- AMBA (by ARM) is an on-chip bus specification for system-on-chip designs.
- Provides two on-chip buses connected through a bridge:
  - ✓ A high-speed system bus (AHB) to connect processors, high-performance peripherals, DMA controller and on-chip memories.
  - ✓ A low-speed peripheral bus (APB) that follows a simpler protocol to connect timers, general–purpose (non-critical) peripherals, and serial interfaces.
- The implementation of communication strategies will be investigated later.

Conclusions

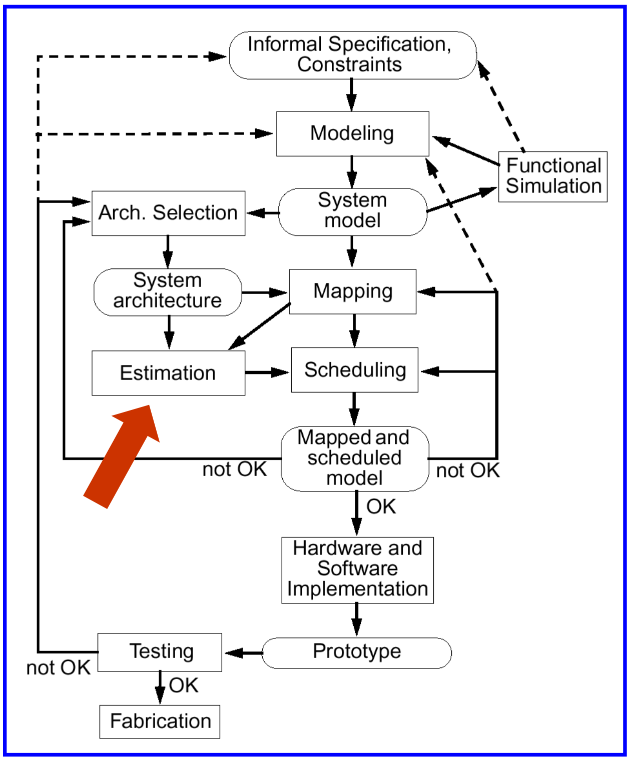
- System-level synthesis produces a system architecture and the behavioral models assigned to the components of this architecture.
- All steps of system synthesis from the design space exploration to the communication synthesis have to be supported by CAD tools.
- Architecture selection allocates the components for the implementation of the system and decides on the system topology.
- Hardware-software partitioning is an important aspect of the system-level synthesis that decides about the implementation nature of each system task.
- Timing behavior is critical for real-time embedded systems. So, scheduling is another quite important aspect of the system-level synthesis that decides about the order and the manner of execution of the system tasks.
- Communication synthesis produces the hardware and software which interconnects the system components and allows them to communicate.
- Design decisions are based on the estimation of design parameters (next lecture) that can be performed through system analysis, simulation or prototyping.

3b. System estimation

University of Thessaly

Department of Computer and Communication Engineering

Design flow



Informal Specification, Constraints

Modeling

Functional Simulation

Arch. Selection

System model

System architecture

Mapping

Estimation

Scheduling

Mapped and scheduled model

not OK

not OK

OK

Hardware and Software Implementation

Prototype

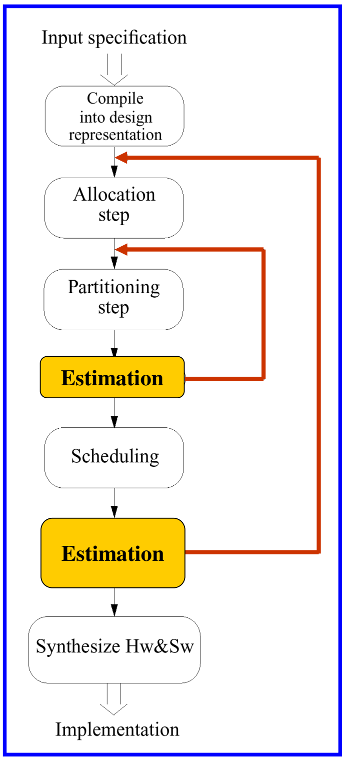
Testing

not OK

OK

Fabrication

System estimation



Input specification

Compile into design representation

Allocation step

Partitioning step

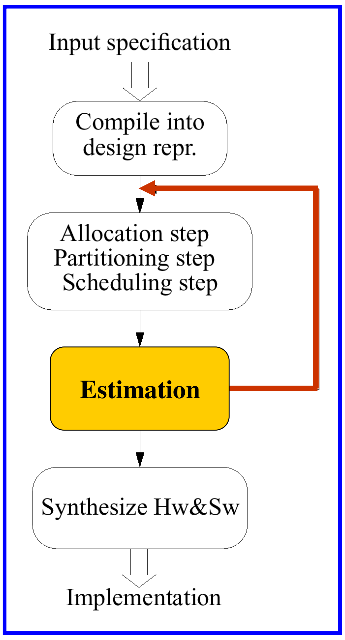
Estimation

Scheduling

Estimation

Synthesize Hw&Sw

Implementation



Input specification

Compile into design repr.

Allocation step
Partitioning step
Scheduling step

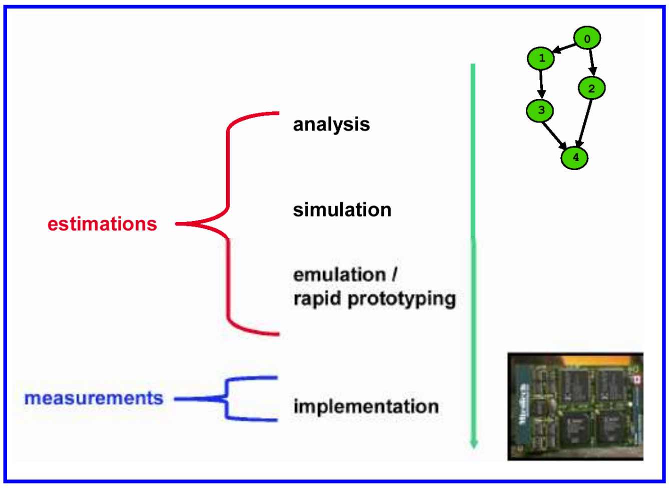
Estimation

Synthesize Hw&Sw

Implementation

System estimation (cont'd)

- Through estimation we get design parameters for the system without actually implementing it.
  - ✓ Supports design decisions.
  - ✓ Enables design space exploration.
  - ✓ Forms the basis for system optimizations.



estimations

analysis

simulation

emulation / rapid prototyping

measurements

implementation

Design parameters and metrics

- Performance:
  - ✓ Hardware: clock period, latency, execution time, throughput.
  - ✓ Software: execution time, worst-case execution time, throughput.
  - ✓ Communication: communication time, throughput.
- Cost: processing resources, silicon area, memory needs.
- Power consumption: in both hardware and software components.
- Others: time-to-market, size etc.

Hardware estimation - Performance metrics

- Clock period (T) depends on the used resources and technology, as well as on delay of the system's functional units.
- When a datapath contains N_k functional units with delays D_k, then the clock period is usually estimated by:

$$T = \max (D_1, D_2, \dots, D_k)$$

- Latency (L): number of clock periods needed for execution of a task in a given datapath.
- Execution time (T_{ex}) for a task: $T_{ex} = T \times L$.
- Throughput (R) for a task: $R = 1 / T_{ex}$.
- The estimation of the timing behavior of a system is based on estimations of the execution time at the process level and on the estimation of the communication times.
- It is strongly connected to the applied scheduling strategy and to the level of resource utilization.

Hardware estimation - Cost metrics

- Silicon area estimated in:
  - ✓ Actual (sq. mm) or parametrized (λ^2) units.
  - ✓ Number of transistors or equivalent gates.
  - ✓ Number of logic or functional blocks.
- Memory and register requirements.
- Package and number of input-output pins.

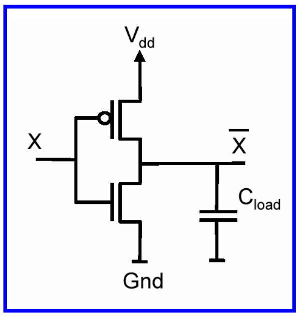
In CMOS technologies, λ is the half of the minimum allowed length (LMIN) of a single transistor's gate.

Year	LMIN (nm)	λ
2001	180	90
2003	130	65
2005	90	45
2007	65	33
2009	45	23

Hardware estimation - Power consumption

- Average power consumption in digital CMOS circuitry:

$$\boxed{P_{avg} = P_{dynamic} + P_{short-circuit} + P_{leakage} + P_{static}}$$

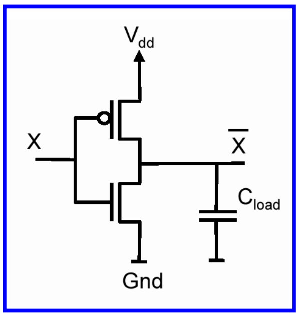


  - ✓ $P_{dynamic}$ is the power consumed due to charging and discharging of the capacitive loads, and is given by the product of the load capacitance, the square of the supply voltage and the frequency.
  - ✓ $P_{short\text{-}circuit}$ is the power consumed due to short-circuit currents between the supply rails during switching.
  - ✓ $P_{leakage}$ is the power consumed due to leakage currents.
  - ✓ P_{static} is the static power consumption occurred in some CMOS implementations.
- The first two components are strongly dependent on the transition activity of the circuitry.

Hardware estimation - Power consumption (cont'd)

$$P_{dynamic} = s\ C_L V_{DD}^2 \frac{1}{T} = s\ C_L V_{DD}^2\ f$$

$$T_d \propto \frac{C_L V_{DD}}{K(V_{DD} - V_{TH})^{\alpha}}$$



s: switching activity factor

- The power consumption is reduced by lowering the supply voltage. However, in this case the delay will be increased.
- The reduction of the load parasitic capacitance may lead to simultaneous improvement of power consumption and delay of a logic block. However, the driving capability of the subsequent block will be reduced, increasing its delay.
- Due to the fact that power and delay are conflicting metrics, it is better to use as metric for hardware components the power-delay product.

Software estimation - Performance metrics

- Execution time:

$$T = I_c \ x \ CPI \ x \ \tau = (I_c \ x \ CPI) / f$$

- I_c : instruction count for a program
- CPI : average cycles per instruction
- τ : clock period, f : clock frequency

Throughput metrics: amount of work that a processor can perform in a given time period (benchmarks are used)

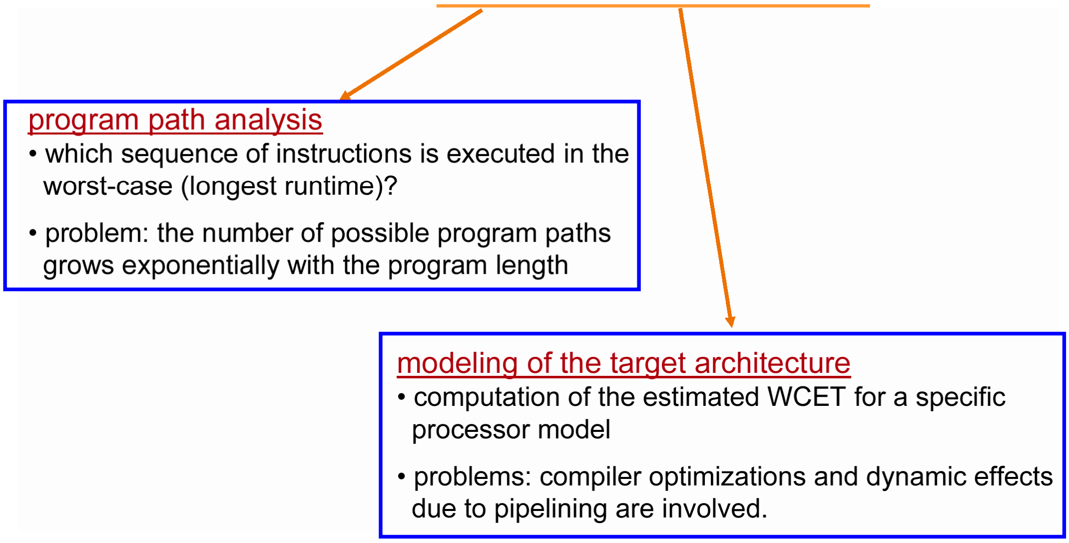
- MIPS rate (million instructions per second):

$$\text{MIPS} = I_c / (T \ x \ 10^6) = f / (CPI \ x \ 10^6)$$

- MFLOPS: million floating-point operations per sec.
- MACS: million multiply/accumulate operations per sec (important for DSPs).
- MOPS: million operations per second (concerns all operations).

Software estimation - Performance metrics (cont'd)

- An important performance metric for real-time systems with hard timing constraints is the Worst Case Execution Time (WCET).
- WCET cannot be estimated by profiling. Program analysis techniques are used.



program path analysis

- which sequence of instructions is executed in the worst-case (longest runtime)?
- problem: the number of possible program paths grows exponentially with the program length

modeling of the target architecture

- computation of the estimated WCET for a specific processor model
- problems: compiler optimizations and dynamic effects due to pipelining are involved.

Worst Case Execution Time (WCET)

- Definition: A program consists of N basic blocks, where each basic block B_i has a worst-case execution time c_i and is executed for x_i times. Then the worst-case execution time of the program is given by:

$$\boxed{WCET = \sum_{i=1}^{N} c_i \cdot x_i}$$

The c_i can be estimated because the sequence of instructions is known.

Software estimation - Cost metrics

- Cost related to processor type.

- Program memory requirements:
  - ***instr_size(j)*** is the memory requirement of a generic instruction ***j***

$$\boxed{progsize_{B_i} = \sum_{\boldsymbol{j} \in \boldsymbol{B}_i} instr\_size\,(\boldsymbol{j})}$$

- Data memory requirements:
  - a program has a set D of declarations
  - ***data_size(d)*** is the memory requirement of declaration ***d***

$$\boxed{data\_size = \sum_{\boldsymbol{d} \in \boldsymbol{D}} data\_size\,(\boldsymbol{d})}$$

Software estimation - Power consumption

- Power (in W) consumed during the execution of the application code to the processor (important for dimensioning of packaging, power supply, cooling).

- Power-delay product: P_{avg} x T_{exe} (in mW x sec). It is important for mobile devices with high performance requirements.

- For processor power characterization: power related to the clock cycle (in mW/MHz), and also power in mW/MIPS and in mW/SPEC.

- SPEC (Standard Performance Evaluation Corporation) is a speed metric for processors, which is based on performance measurements for standard compute-intensive workloads.

Mobile processors examples

Processor (Vendor)	Techn. [μ]	V_{DD} [V]	Clock [MHz]	Power [mW]	MIPS	MIPS/W
StrongARM (Intel)	0.35	2	230	360	268	744
ARM710 (VLSI)	0.8	3.3	25	120	30	250
ARM940T (VLSI)	0.35	3.3	150	675	160	237
MMC2001 (Motorola)	0.35	2	34	80	31	387
TR4102 (LSI)	0.25	1.8	80	40	90	2250
SH7708 (Hitachi)	0.5	3.3	25	95	25	263
SH7750 (Hitachi)	0.25	1.8	200	1600	300	188

Conclusions

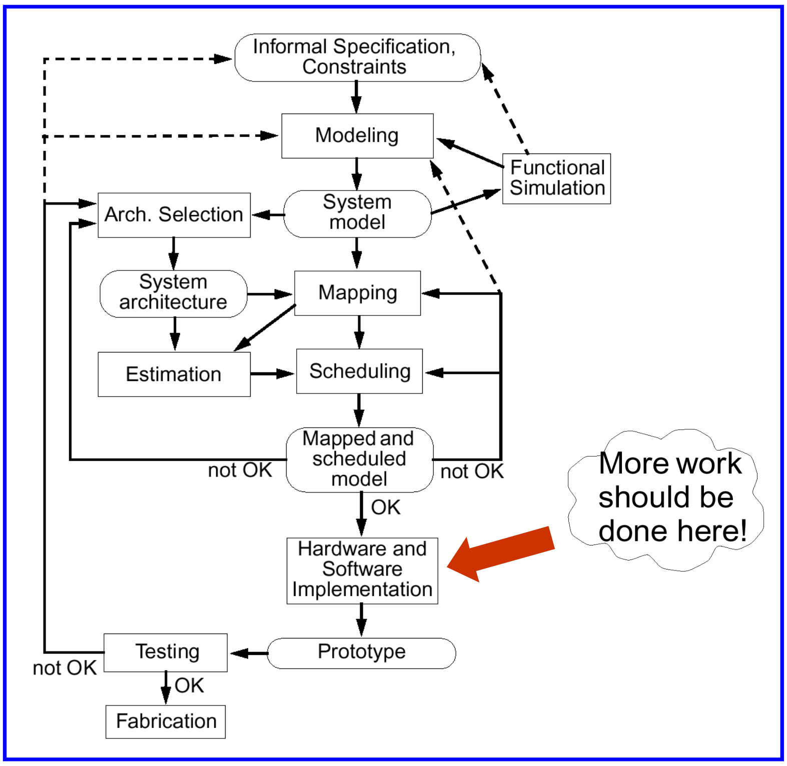
- A design environment should contain several estimation tools for the same parameter, which differ in accuracy and estimation time.
- They can be used at different stages of the synthesis process.
- Estimation of several parameters can be performed during the high-level analysis of the system, as well as during the simulation and the prototyping stages.
- At certain stages (levels) of the design procedure it is not needed that the estimation quantitatively reflects the value of the respective parameter.
- It is enough that it captures the evolution of the parameter from one design step to the other.

3c. Low-level synthesis: Hardware synthesis

University of Thessaly

Department of Computer and Communication Engineering

Design flow



Informal Specification, Constraints

Modeling

Functional Simulation

Arch. Selection

System model

System architecture

Mapping

Estimation

Scheduling

Mapped and scheduled model

not OK

not OK

OK

Hardware and Software Implementation

Prototype

Testing

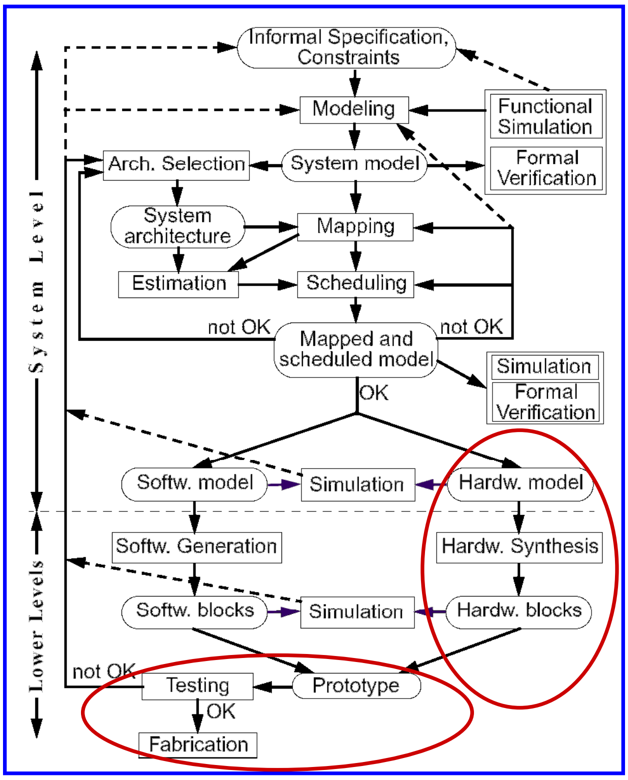
not OK

OK

Fabrication

More work should be done here!

Design flow (cont'd)



- During the implementation phase, hardware & software components have to be developed (implemented) in a coordinated way.

- Hardware-software co-simulation is important.

Hardware implementation

- Encoding in a hardware description language: VHDL, Verilog, SystemC.

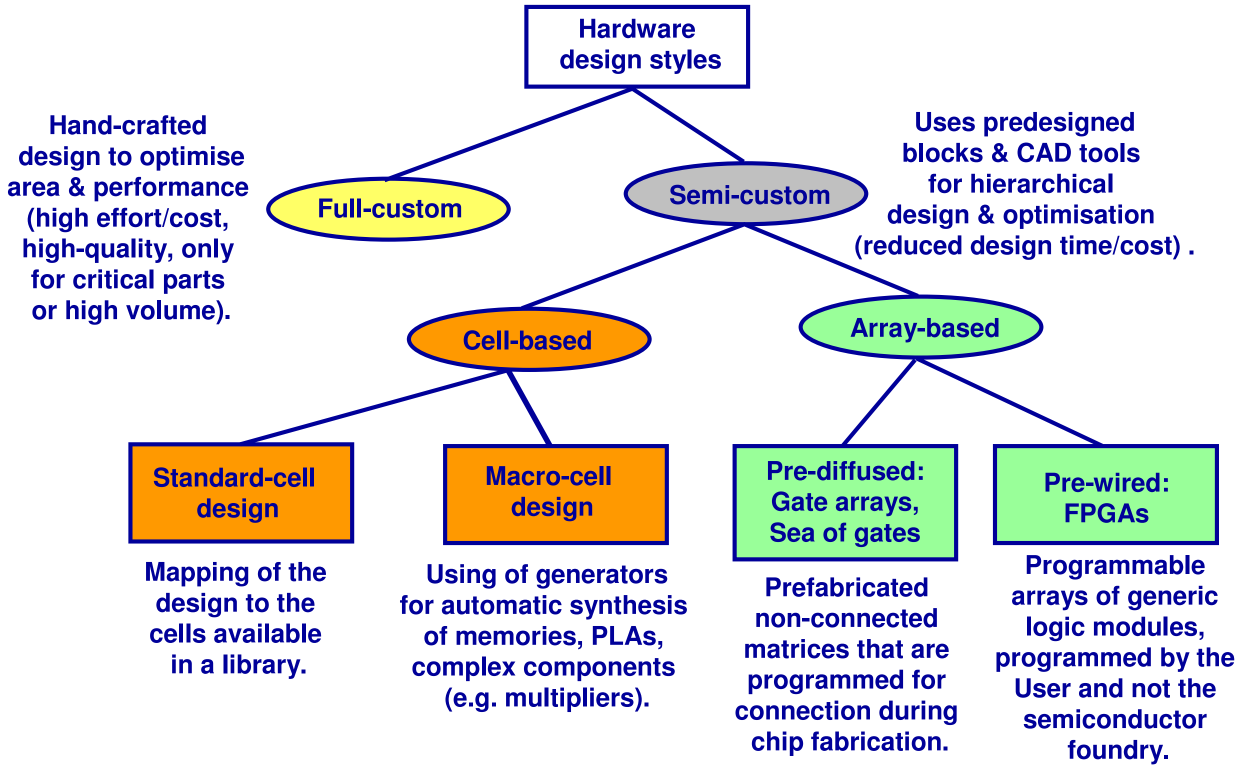
- Performing successive synthesis steps:
  - ✓ High-level synthesis.
  - ✓ Register transfer level synthesis.
  - ✓ Logic-level synthesis.

- Testing by simulation, co-simulation and prototyping platforms.

- During prototyping, a prototype of the hardware is constructed and the generated software is executed on the target architecture.

- Layout and physical implementation.

Hardware design styles



Hardware design styles

Full-custom

Hand-crafted design to optimise area & performance (high effort/cost, high-quality, only for critical parts or high volume).

Semi-custom

Uses predesigned blocks & CAD tools for hierarchical design & optimisation (reduced design time/cost) .

Cell-based

Array-based

Standard-cell design

Mapping of the design to the cells available in a library.

Macro-cell design

Using of generators for automatic synthesis of memories, PLAs, complex components (e.g. multipliers).

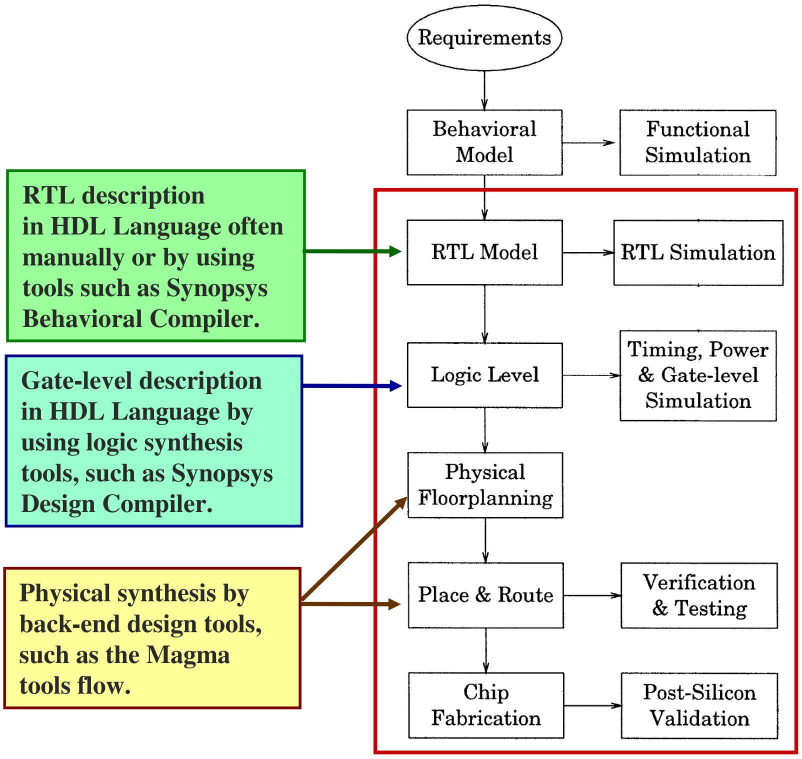
Pre-diffused: Gate arrays, Sea of gates

Prefabricated non-connected matrices that are programmed for connection during chip fabrication.

Pre-wired: FPGAs

Programmable arrays of generic logic modules, programmed by the User and not the semiconductor foundry.

Hardware design flow



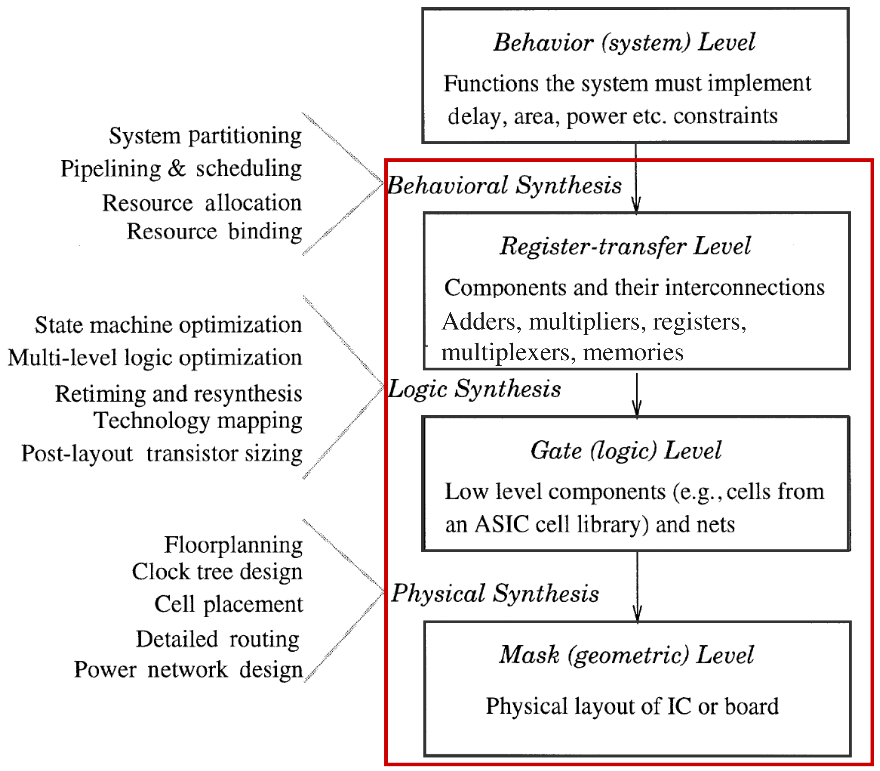
RTL description in HDL Language often manually or by using tools such as Synopsys Behavioral Compiler.

Gate-level description in HDL Language by using logic synthesis tools, such as Synopsys Design Compiler.

Physical synthesis by back-end design tools, such as the Magma tools flow.

- Hardware designs are specified using Boolean equations, schematic diagrams, state transition diagrams, finite state machine or HDL descriptions in a technology independent form.
- Logic synthesis tools are then used to synthesize the above specifications into functional and control units, respectively, and apply optimisations.

Hardware design flow (cont'd)



Behavior (system) Level

Functions the system must implement
delay, area, power etc. constraints

System partitioning
Pipelining & scheduling
Resource allocation
Resource binding

Behavioral Synthesis

Register-transfer Level

Components and their interconnections
Adders, multipliers, registers, multiplexers, memories

State machine optimization
Multi-level logic optimization
Retiming and resynthesis
Technology mapping
Post-layout transistor sizing

Logic Synthesis

Gate (logic) Level

Low level components (e.g., cells from an ASIC cell library) and nets

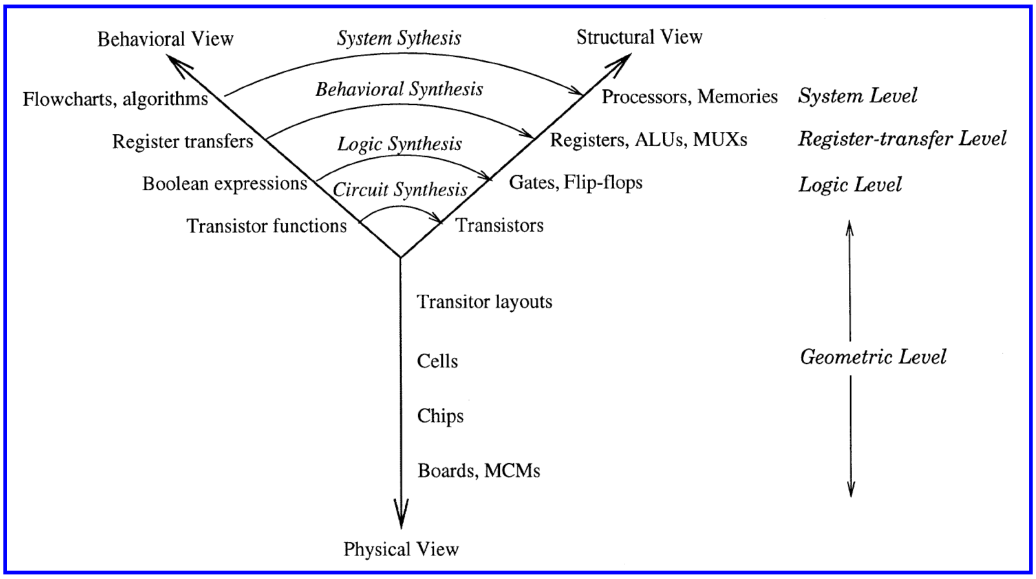
Floorplanning
Clock tree design
Cell placement
Detailed routing
Power network design

Physical Synthesis

Mask (geometric) Level

Physical layout of IC or board

Relations between abstraction levels: Y-chart



Behavioral View

System Sythesis

Structural View

Flowcharts, algorithms

Behavioral Synthesis

Processors, Memories

System Level

Register transfers

Logic Synthesis

Registers, ALUs, MUXs

Register-transfer Level

Boolean expressions

Circuit Synthesis

Gates, Flip-flops

Logic Level

Transistor functions

Transistors

Transitor layouts

Cells

Geometric Level

Chips

Boards, MCMs

Physical View

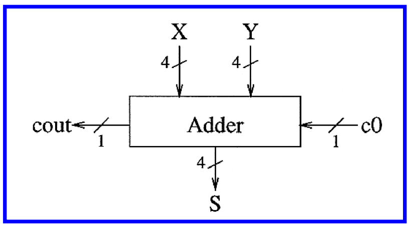
- At any stage of the design flow, the design refinement step is performed by using CAD tools (in conjunction with manual methods).
- In more recent years, some vendors are specialized in design of reusable blocks, which are sold as IP (Intellectual Property blocks) to other design houses, who then assemble these blocks together to create System-on-Chips (we will investigate this issue in a next lecture).

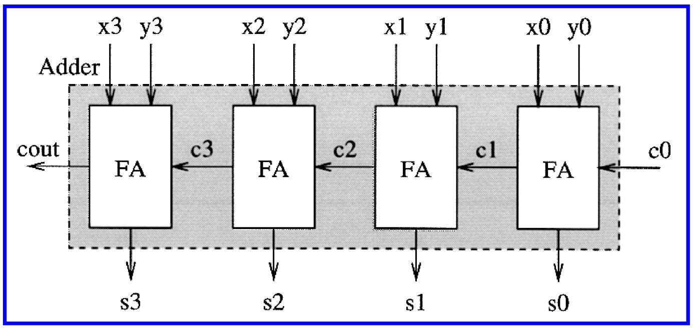
Hierarchical design

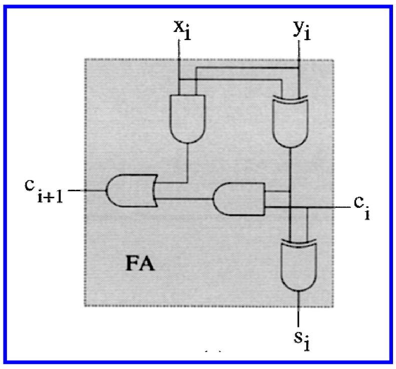
- Hierarchical decomposition of a complex system into simpler subsystems and further decomposition into subsystems of more simplicity.
- A system design can be recursively broken into components, each of which is composed of smaller components until the smallest components can be described in terms of gates and/or transistors.
- At any level of hierarchy, each component is treated as a black box with known input-output behavior, but the behavior description is unknown. Each black box is designed by building simpler black boxes until the lowest level of hierarchy (gate / transistor level).
- This is a hierarchical top-down design approach which helps to a step by step reduction of the system's complexity, and to an easier understanding of the functionality (without having to worry about low-level details).
- The bottom-up design approach starts by designing the lowest-level components and use these components to build components of more complexity until the final design requirements are met (if a low-level block turns out to be infeasible the whole process has to be repeated).
- Current design teams use a mixture of both approaches where critical low-level blocks are built concurrently with the system and blocks development.

Hierarchical design (cont'd)

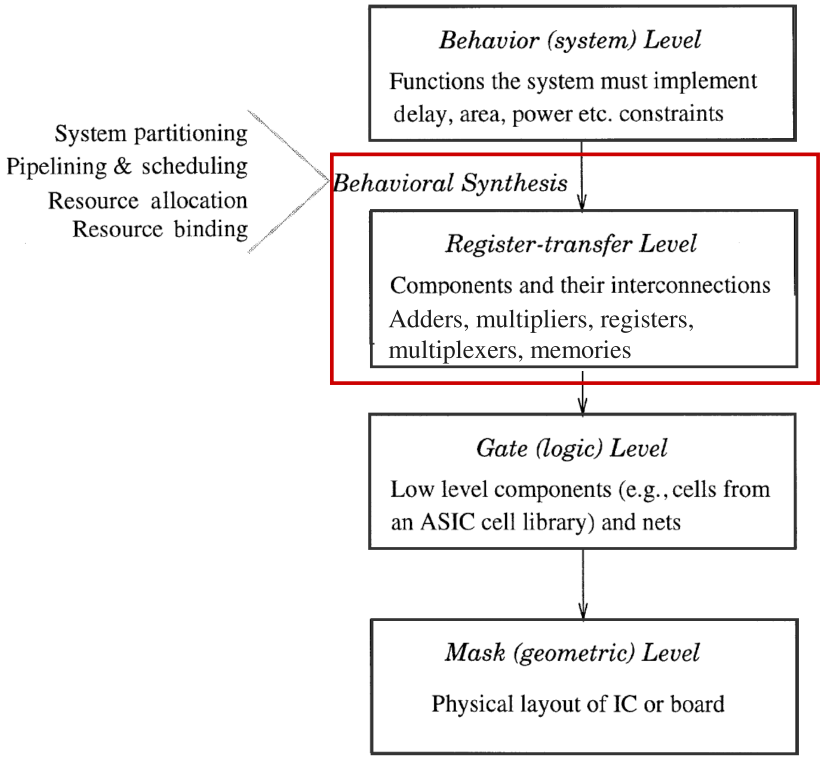
Example of top-down design approach:
4-bit ripple-carry adder







RTL synthesis

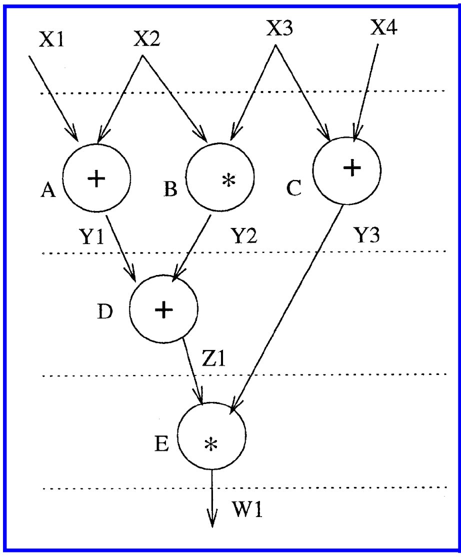

Behavior (system) Level
Functions the system must implement
delay, area, power etc. constraints
System partitioning
Pipelining & scheduling
Resource allocation
Resource binding
Behavioral Synthesis
Register-transfer Level
Components and their interconnections
Adders, multipliers, registers,
multiplexers, memories
Gate (logic) Level
Low level components (e.g., cells from an ASIC cell library) and nets
Mask (geometric) Level
Physical layout of IC or board

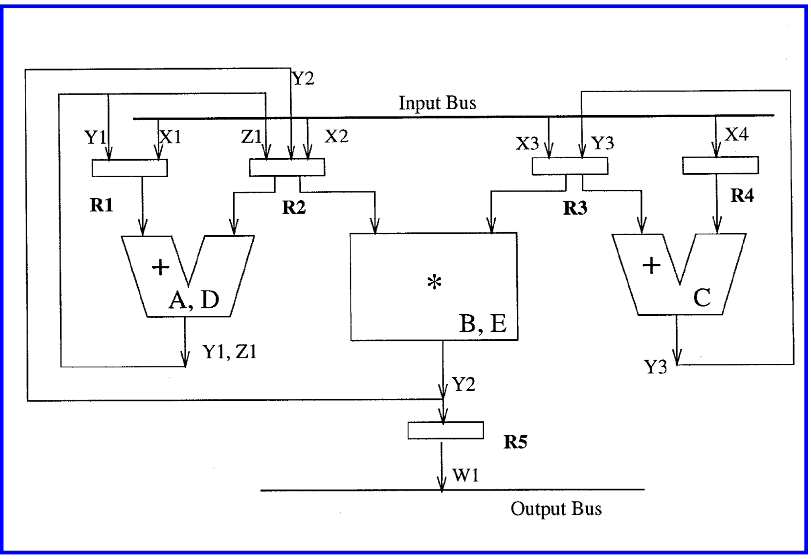
RTL synthesis (cont'd)

- The tasks in high-level synthesis are classified into allocation, binding (assignment of functionality to the resources) and scheduling.
- The components / resources of the allocation / binding task at the RTL are taken from a library of available modules, which includes components such as ALUs, adders, multipliers, registers, multiplexers, memories.
- Scheduling assigns each of the operation to time intervals (control steps).
- The data flows from one stage of registers to the next during each control step after the computation phase in a functional unit.
- The control steps are usually the length of the clock cycle.
- After the scheduling and the binding (assignment of operations to the functional units), the interconnects between the various units are also established.

RTL synthesis (cont'd)

Example showing the ability of CAD tools to synthesize a behavioral description into a data path



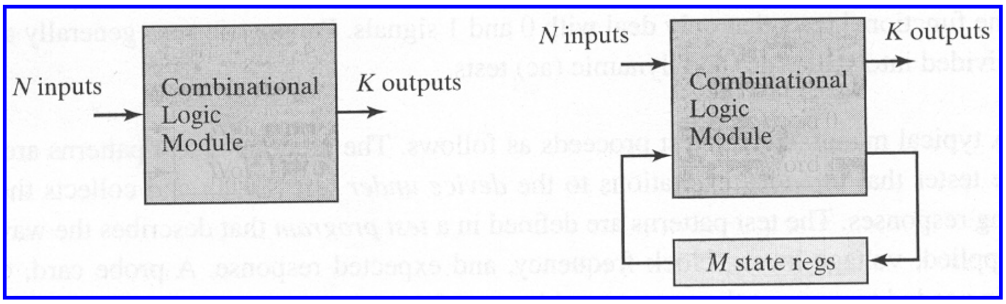


Data flow graph

Synthesized data path using a high-level synthesis system

Design for testability

- Some small modifications in a circuit can help make it easier to validate the absence of faults. This approach to design is called design for testability (DFT).
- DFT attempts to modify the circuit during the design phase without affecting its functionality so as to make it testable.

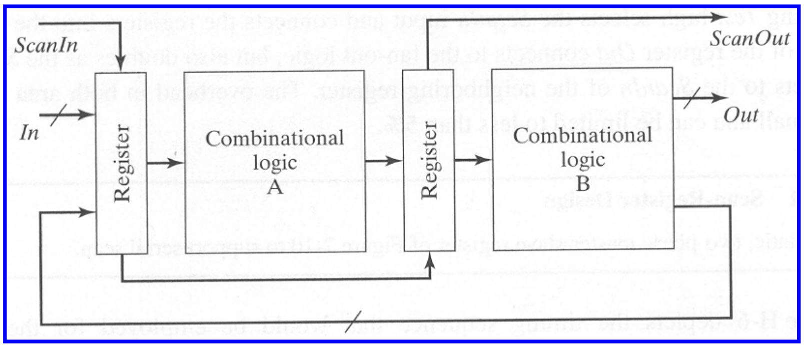


- In a combinational circuit, its correctness can be validated by exhaustively applying all possible input patterns (2^N) and observing the responses.
- The situation gets more dramatic in sequential modules, because the output depends not only upon the applied inputs, but also upon the value of the state (2^{N+M}).

Design for testability (cont'd)

- However, a single fault in a circuit is covered by a number of input patterns, and thus detection of that fault requires only one of these patterns, while the rest are not useful.
- A substantial reduction in the number of patterns can be obtained by relaxing the condition that all faults must be detected. Typical test procedures attempt a 95-99% fault coverage.
- Detecting the last single percentage of possible faults might require an exorbitant number of extra patterns, and the cost of detection might larger than the eventual replacement cost.
- Thus, it is possible to test most combinational modules with a limited set of input vectors.
- Testing a single fault in a sequential module requires a sequence of vectors, and this might make the process very expensive.
- One way to address this problem is to turn the sequential module into a combinational one by breaking the feedback loop during the test procedure. This is the key concept behind the scan-test approach.

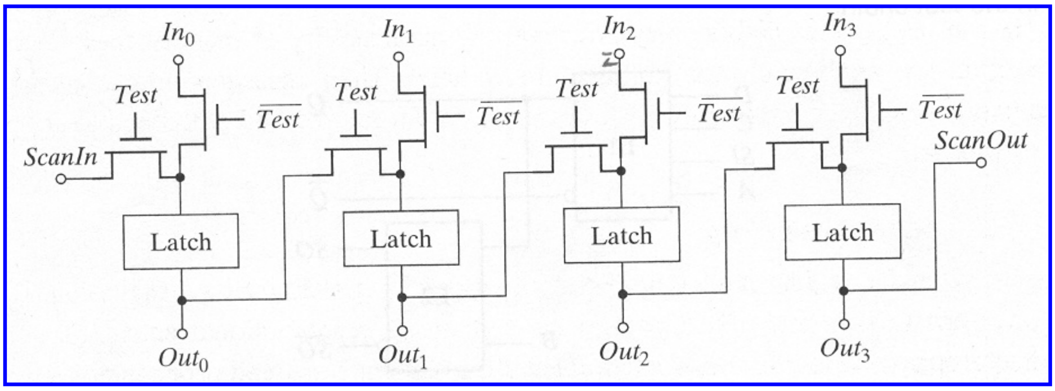
Design for testability (cont'd)



Modification of the registers is performed Automatically using tools such as Synopsys Tetra

- The registers have been modified to support two operation modes. In the normal mode they act as N-bit clocked registers. During the test mode the registers are chained together as a single serial shift register. All (full-scan) or some of the registers (partial scan) in a system can be connected in a test scan chain.
- An excitation that is applied to the logic module A, entered through *ScanIn* and shifted into the registers under control of a test clock.
- The excitation is applied to the logic, propagates to the output of the logic module, and the result is latched into the registers.
- The result is shifted out through the *ScanOut* & compared with the expected data.

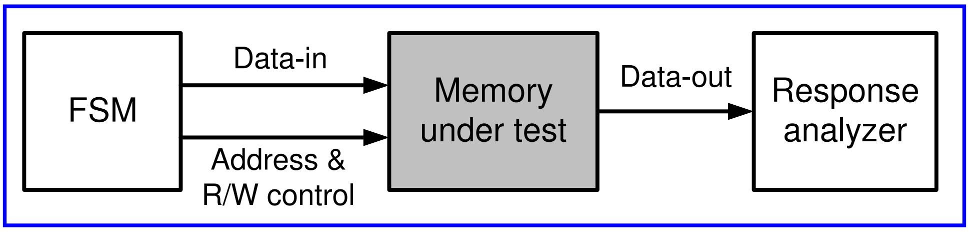
Design for testability (cont'd)



- This approach incurs only a minimal overhead.
- The serial nature of the scan chain reduces the routing overhead.
- Traditional registers can easily modified to support the serial scan technique.
- The only addition is an extra multiplexer at the input. When *Test* is low, the circuit is in normal operation, and when *Test* is high selects the *ScanIn* input and connects the registers into the scan chain.

Design for testability (cont'd)

- An alternative approach to testability is having the circuit itself generate the test patterns instead of requiring the application patterns.
- This requires the addition of extra circuitry for the generation and the analysis of the patterns.
- The self-test (or Built-in Self Test – BIST) technique is beneficial when testing regular structures such as memories.
- Memory tests include the reading and writing of a number of different patterns into and from the memory using several addressing sequences.
- The overhead is small compared to the size of the used memory modules.



FSM

Data-in

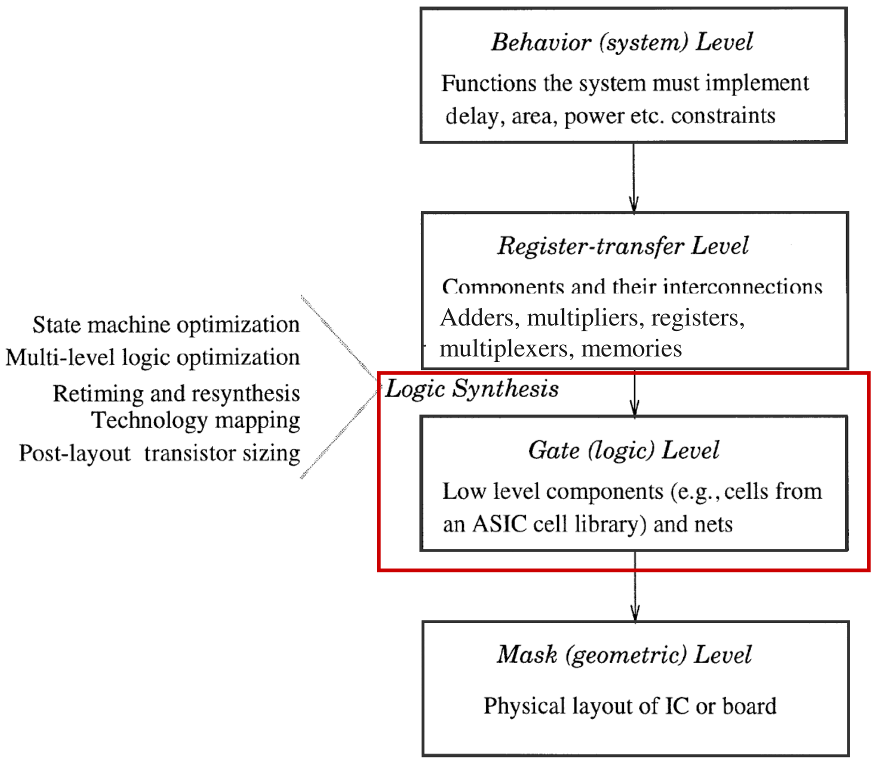
Address & R/W control

Memory under test

Data-out

Response analyzer

Logic synthesis



Behavior (system) Level

Functions the system must implement delay, area, power etc. constraints

Register-transfer Level

Components and their interconnections Adders, multipliers, registers, multiplexers, memories

Logic Synthesis

State machine optimization
Multi-level logic optimization
Retiming and resynthesis
Technology mapping
Post-layout transistor sizing

Gate (logic) Level

Low level components (e.g., cells from an ASIC cell library) and nets

Mask (geometric) Level

Physical layout of IC or board

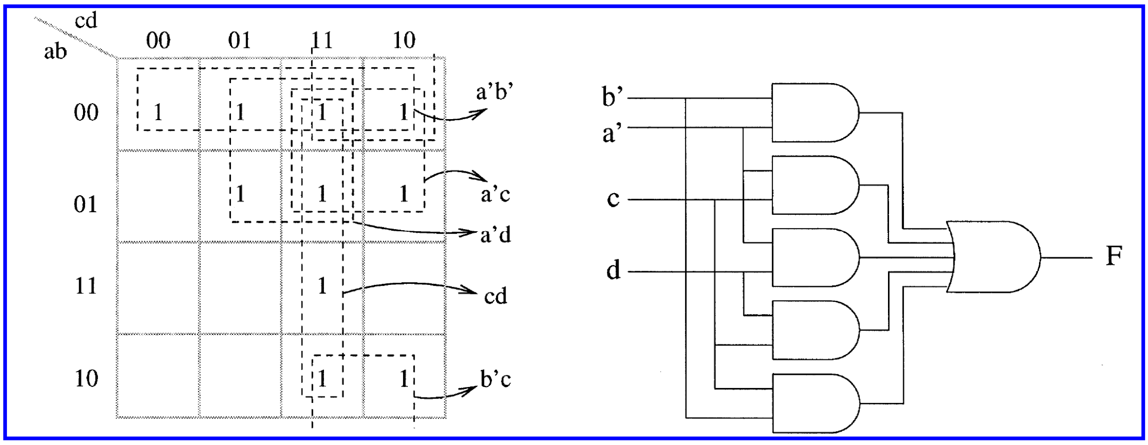
Logic synthesis (cont'd)

- Logic synthesis deals with the synthesis and the optimization of circuits at the logic (gate) level.
- Digital circuits typically have sequential and combinational components.
- These can be specified by finite-state machines, state transition diagrams, Boolean equations, schematic diagrams or HDL descriptions.
- Logic synthesis includes a range of optimisations and refinements:
  - ✓ Logic optimization of Boolean functions.
  - ✓ State machine optimization by state minimization and encoding.
  - ✓ Retiming and resynthesis.
  - ✓ Technology mapping.
  - ✓ Post-layout transistor sizing.
- The optimisation steps are selected according to the chosen optimization metric (area, speed, power or trade-off between them), and they are either technology-independent (the first two) or technology-dependent (the others).

Combinational logic optimisation

- Combinational circuits can be modeled by two-level sum-of-products expressions, which can be optimised by two-level minimization tools (Espresso, Mini, Presto).
- The logic optimisation is based on table-based methods (Karnaugh maps), which are used to minimise a Boolean function.

$$F = a'b'c'd' + a'b'c'd + a'b'cd' + a'b'cd + a'bc'd + a'bcd' + ab'cd' + a'bcd + ab'cd + abcd$$



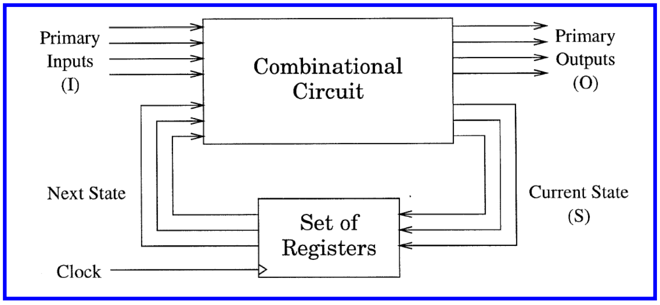
$$F = a'b' + a'c + a'd + cd + b'c$$

Combinational logic optimisation (cont'd)

- The example previously discussed demonstrate a two-level optimization methodology. The final circuit implementation for the example has two stages of logic.
- However, cell libraries used to map the gates in the logic circuit to the gates available from the foundry, usually have more complex gates which are a combination of several gates such as AND-OR, OR-AND, or NOR-AND gates.
- To fully utilize these cell libraries, multi-level logic optimization techniques are used. These techniques are not restricted to two-level logic networks but instead deal with multiple-level logic circuits.
- This provides the necessary flexibility required to map the logic network to complex cells in the technology library, hence optimizing area and delay.
- However, multi-level optimization techniques are not exact, i.e. only heuristics exist for modeling and optimizing multiple-level networks.

Sequential logic optimisation

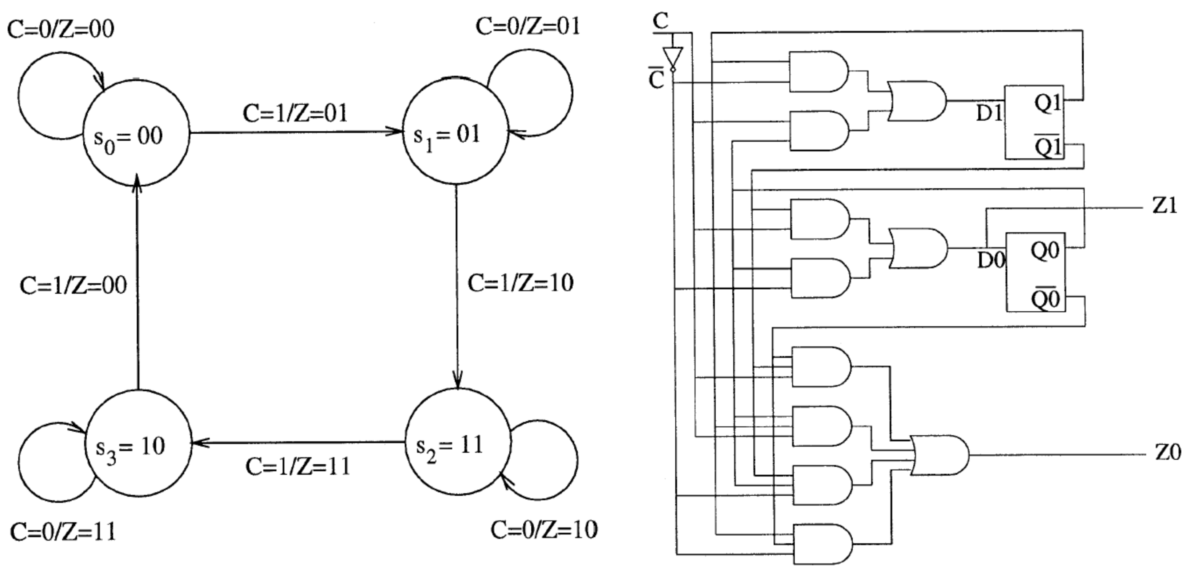
- Sequential circuits are usually represented by FSM models. At every clock cycle the data computed by the combinational circuit is stored in the registers along with other state and control information.



- In a sequential circuit represented by an FSM, the set of states, inputs and outputs, S, I, and O correspond to k flip-flops outputs ($Q_0, \ldots, Q_{k-1}$), n input signals ($I_0, \ldots, I_{n-1}$) and m output signals ($O_0, \ldots, O_{m-1}$).
- The finite-state machine model is usually represented using state transition diagrams.
- State transition diagrams are optimised by state minimization and state encoding.

Sequential logic optimisation (cont'd)

Example - Modulo-4 counter: the circuit counts from 0 to 3 back to 0 when the count signal C is 1. When C = 0, the counter stays in the same state.

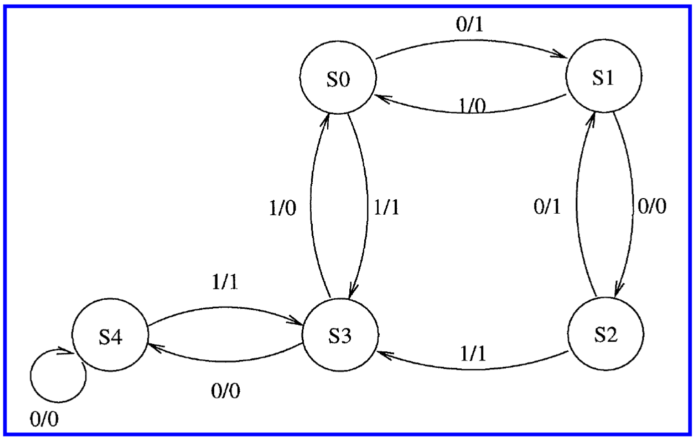


Sequential logic optimisation (cont'd)

- State minimization aims at reducing the number of machine states used to represent an FSM.
- Since the minimum number of bits required to encode *n* states is $\log_2 n$, reducing the number of states can lead to a reduced number of flip-flops.
- It also leads to fewer transitions, fewer logic gates, and fewer inputs per gate.
- These reductions not only lead to lower area cost but also speed up the design and reduce the power consumption.
- State minimization can be done by finding *equivalent* states and by using '*don't-care'* information to remove states. Two states are equivalent if and only if, for every input, both the states produce the same output and the corresponding next states are equivalent.

Sequential logic optimisation (cont'd)

Example for FSM state minimization:

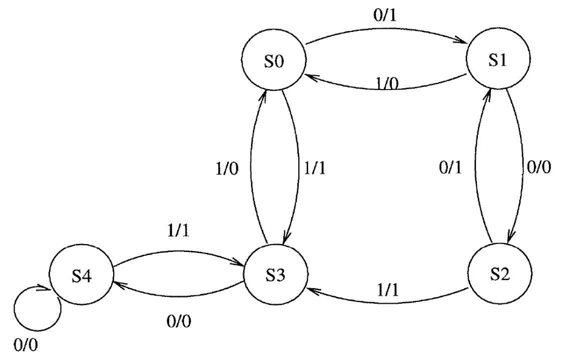


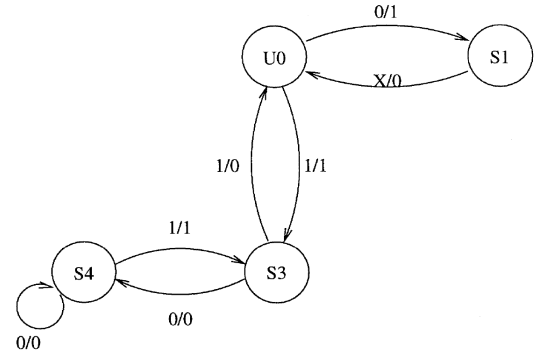
- State minimization is performed in two steps:
  - ✓ Firstly, we separate the states with the same outputs for the same inputs.
  - ✓ Secondly, we compare the next states for each state in the same group.
  - ✓ If the next state for two states within a group is in the same group, then the two states are considered equivalent, and they are combined in a single state.

Sequential logic optimisation (cont'd)

Pres.	Next State/Out	
State	Inp I=0	Inp I=1
$u_0 = \{s_0,s_2\}$	$u_1, u_1/1$	$u_1, u_1/1$
$u_1 = \{s_1,s_3\}$	$u_0, u_2/0$	$u_0, u_0/0$
$u_2 = \{s_4\}$	$u_2/0$	$u_1/1$

Pres.	Next State/Out	
State	Inp I=0	Inp I=1
u_0	$s_1/1$	$s_3/1$
s_1	$u_0/0$	$u_0/0$
s_3	$s_4/0$	$u_0/0$
s_4	$s_4/0$	$s_3/1$





Sequential logic optimisation (cont'd)

- After the states have been minimized, state encoding is performed to assign a binary representation to the states of the FSM.

- In the previous example the minimized FSM has four states, whereas the original state transition graph had five states.

- Hence, whereas it would have taken 3 bits to encode the five states in the original FSM, the reduced FSM requires only 2 bits for the encoding.

- Fewer encoding bits implies fewer flip-flops in the circuit and, hence, reduced area and increased speed of the final design.

- In general, there are several other encoding methodologies such as gray encoding, bus-invert encoding etc., which are used to reduce circuit switching or bus switching in order finally to reduce the power consumption of a circuit.

Technology mapping

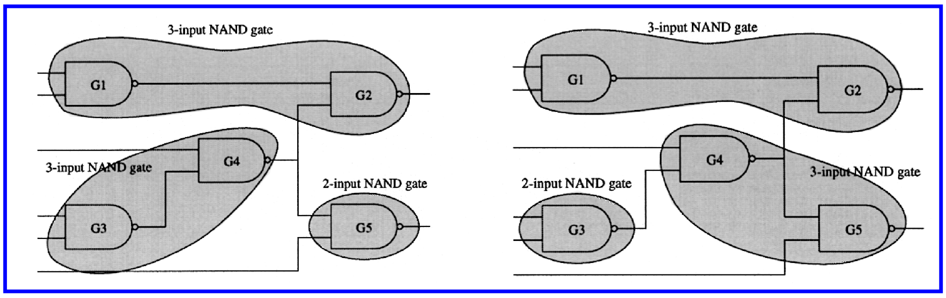
- Technology mapping forms the link between logic synthesis and physical design. After logic synthesis, a circuit-level schematic or netlist of the design (containing the elements with their interconnections) is generated using a vendor-independent logic library.
- This library has elements such as low-level gates, flip-flops, latches, and usually multiplexers, counters, and adders.
- Typically, a netlist translator along with a vendor-specific library is used to replace the vendor-independent generic elements and generate the netlist in a particular vendor's netlist format.
- The process of transforming the generic cell based logic network into a vendor library-specific network is known as library binding or technology mapping.
- This step allows us to retarget the same design to different technologies and implementation styles.
- Typically, the cell library vendor provides different libraries optimized for area, performance, power.

Technology mapping (cont'd)

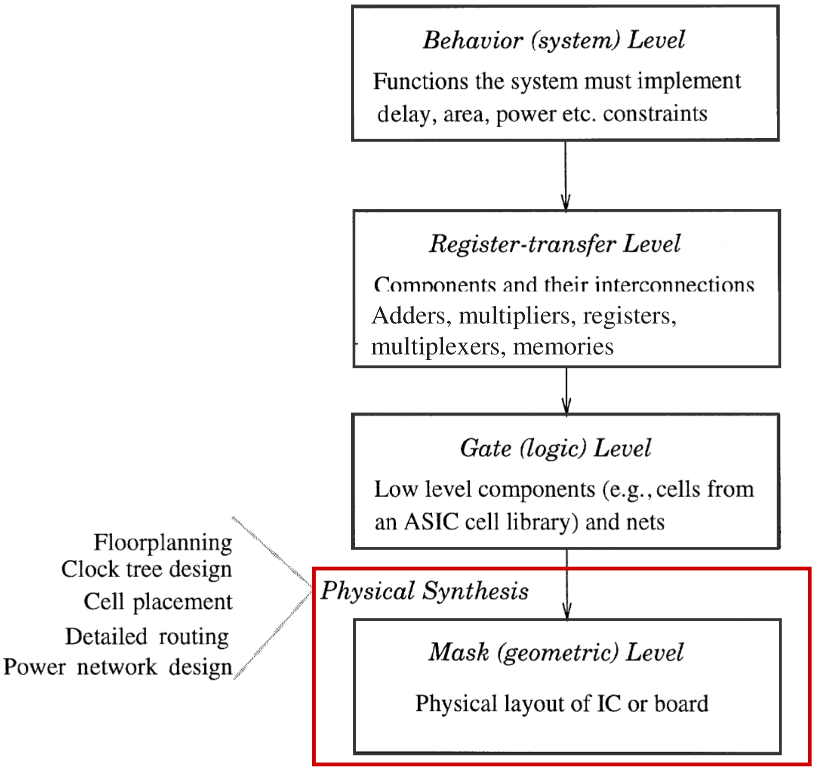
- Each cell in the vendor library contains a physical layout of the cell, a timing model (delay characteristics and capacitances on each input), a wire load model, a behavioral model (in VHDL or Verilog), circuit schematic, cell icon (for schematic tools), and for bigger cells, its routing and testing strategy.
- CAD tools use the timing characteristics to analyze the circuit and determine the capacitances at each node in the netlist, and use delay formulas along with the timing characteristics of each element to compute the delays for each node.
- Wiring capacitances are included by estimating a wire-load model initially and then later using the back-annotation information from the floorplanning and place-and-route tools.

Technology mapping (cont’d)

- Cell-library binding (technology mapping) is in fact the process of transforming the set of Boolean equations or the Boolean network into a logic gate network with the gates in the cell library.
- Cell-library binding approaches are classified into two types: rule-based and tree-based approaches.
- Rule-based approaches replace parts of the logic network with equivalent cells from the cell library, based on some specific rules.
- The algorithms used in tree-based approaches attempt to find a cover of all the gates in the given logic graph using the cell-library cells so as to minimize the area, delay and recently the power consumption.



Physical synthesis



Behavior (system) Level

Functions the system must implement
delay, area, power etc. constraints

Register-transfer Level

Components and their interconnections
Adders, multipliers, registers,
multiplexers, memories

Gate (logic) Level

Low level components (e.g., cells from an ASIC cell library) and nets

Floorplanning
Clock tree design
Cell placement
Detailed routing
Power network design

Physical Synthesis

Mask (geometric) Level

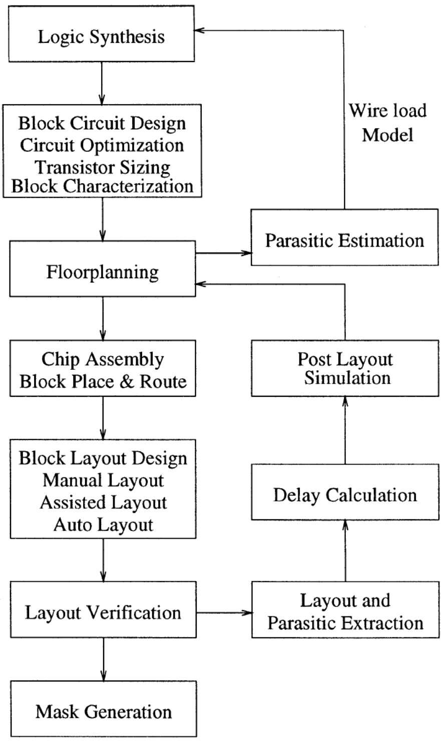
Physical layout of IC or board

Physical synthesis (cont'd)

Creation of an initial map of the location of the various blocks.

Apart from placing and routing of blocks, includes the creation of the clock distribution architecture. Clock is distributed through balanced clock tree with a low-enough slew.

By design rule checkers.



Simulation

- Extensive simulation and functional verification techniques are used by designers at every stage of the design to ensure that no bugs are introduced in the process of refining the design from the behavioral level to the final layout.
- The simulations of the RTL, logic, and physical level descriptions are performed by different kind of simulators.
- Logic-level simulators simulate the circuit at the logic gate level and are used extensively to verify the functional correctness of the design.
- Circuit-level simulation, which is the most accurate simulation technique, operates at a circuit level. The SPICE program is the most known circuit simulation and analysis tool.
- SPICE simulates the circuit by solving the matrix differential equations for circuit currents, voltages, resistances, and conductances.
- SPICE is quite slow, but accurate and useful for blocks characterization.

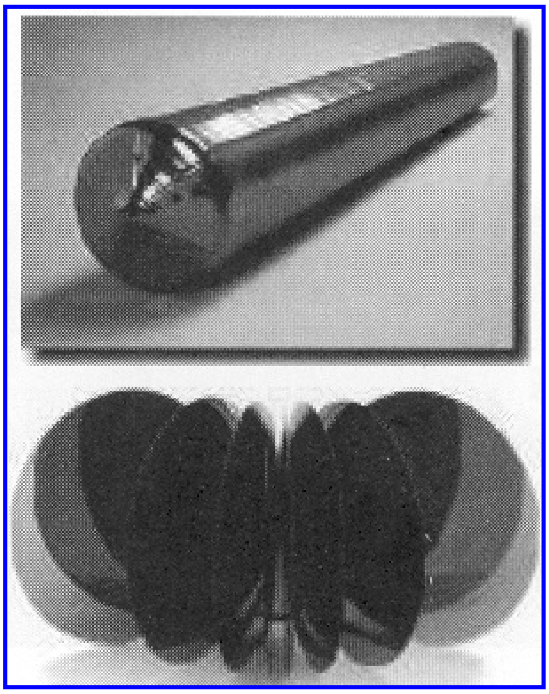
Simulation (cont'd)

- Switch-level simulators, on the other hand, model transistors as switches and, unlike logic simulators, wires are not assumed to be ideal but instead are assumed to have some capacitance.
- Logic-level simulators are typically event-driven. These model the system as a discrete event system by defining appropriate events of interest and how the events are propagated throughout the model.
- Hardware description languages such as VHDL and Verilog have been designed based on event-driven simulation semantics (Modelsim).
- For the delay calculation during the physical synthesis, static timing analysis is used (PrimeTime tool).
- It analyses the paths in the circuit netlist in order to compute the delay along the various paths and determine the critical path(s) in the circuit.
- The timing analysis is performed by using the gate delay, rise and fall times, capacitance, load values of the cell library.

Emulation and verification

- Since testing and correcting a chip once it has been manufactured is a difficult and expensive task, it is essential to verify functional and timing characteristics of the design.
- FPGAs are increasingly being used for circuit prototyping and verification due to their ease of reconfigurability and programming.
- Once the netlist of the circuit design has been generated, it is used to program an FPGA-based circuit consisting of one or several FPGAs (depending on the size of the design).
- Test patterns are then applied to this design to check its functionality.
- Outputs of the emulation circuit are compared with the responses expected according to the system’s specification.
- If design errors are found, the FPGA boards can easily be reprogrammed after the design has been fixed, and it is this ease of reconfigurability that makes FPGAs an attractive, although expensive, prototyping system.

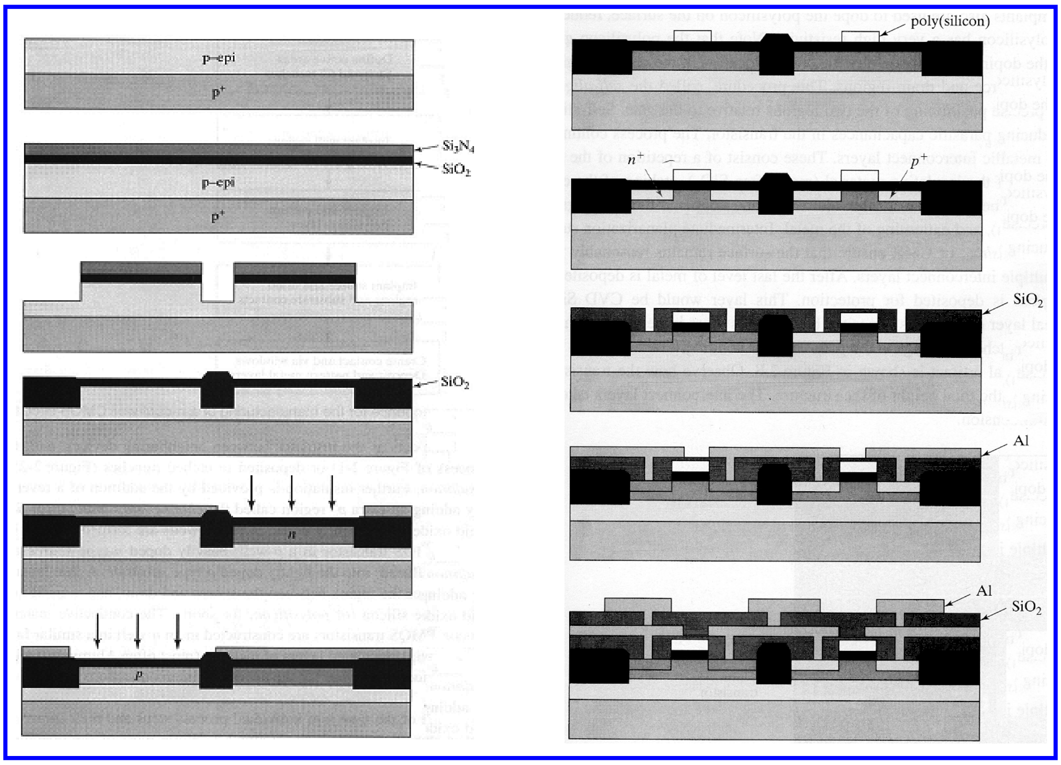
Masks generation and chip fabrication



Single-crystal silicon ingot

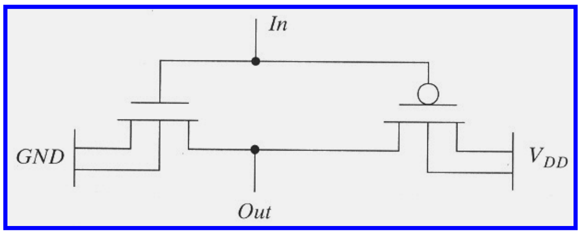
Silicon wafers
Diameter: 30 cm
Depth < 1mm

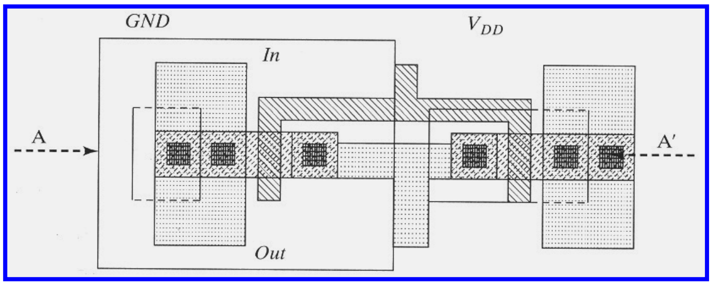
Masks generation and chip fabrication (cont'd)

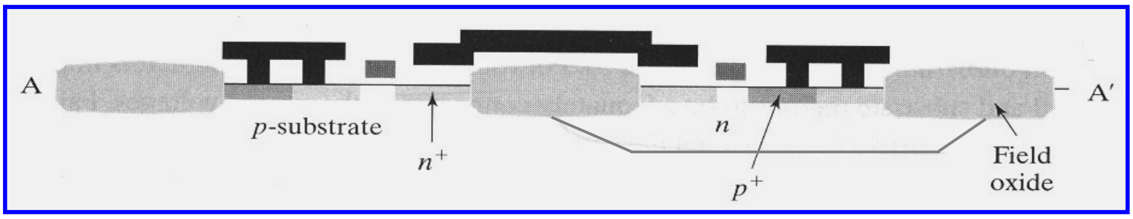


Masks generation and chip fabrication (cont’d)

CMOS inverter
Out = NOT (In)

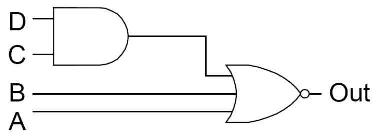


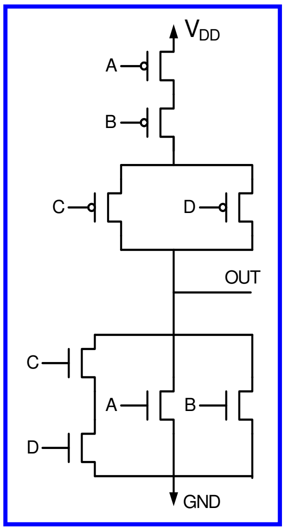


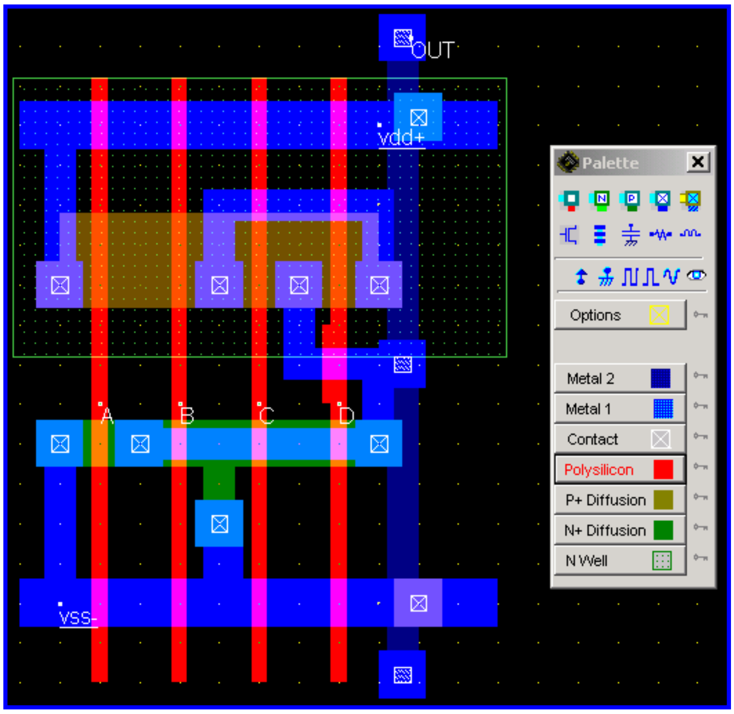


Masks generation and chip fabrication (cont’d)

Out = Not (A + B + C D)

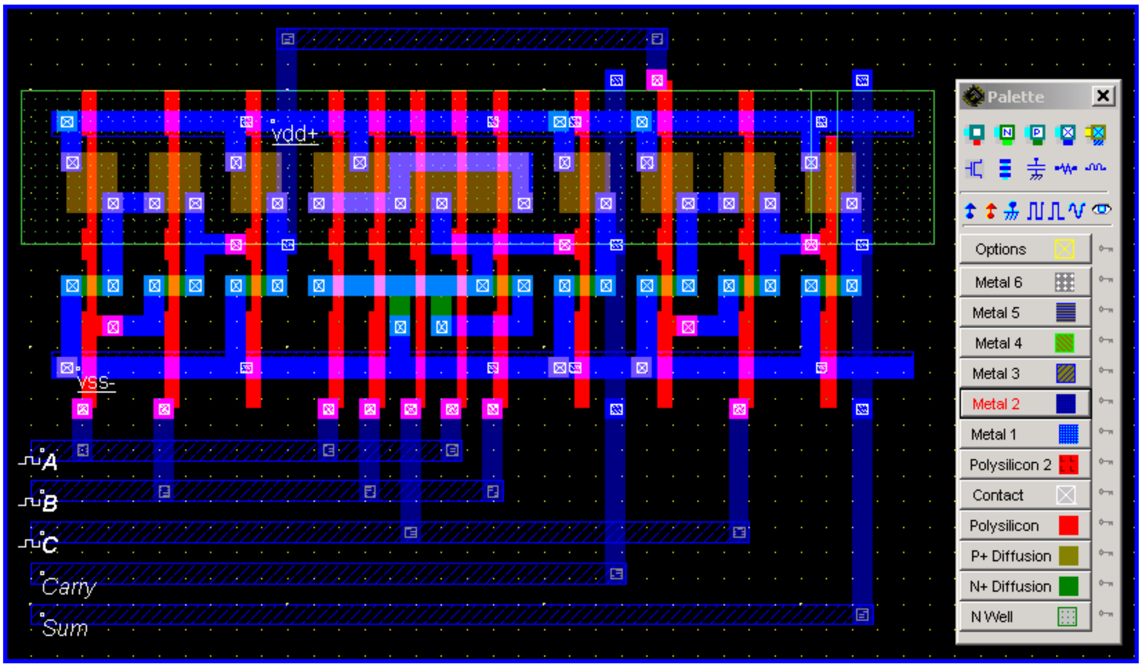






Masks generation and chip fabrication (cont'd)

Full adder: $\boxed{C_o = AB + (A+B)\,C_i \qquad S = A \oplus B \oplus C_i}$

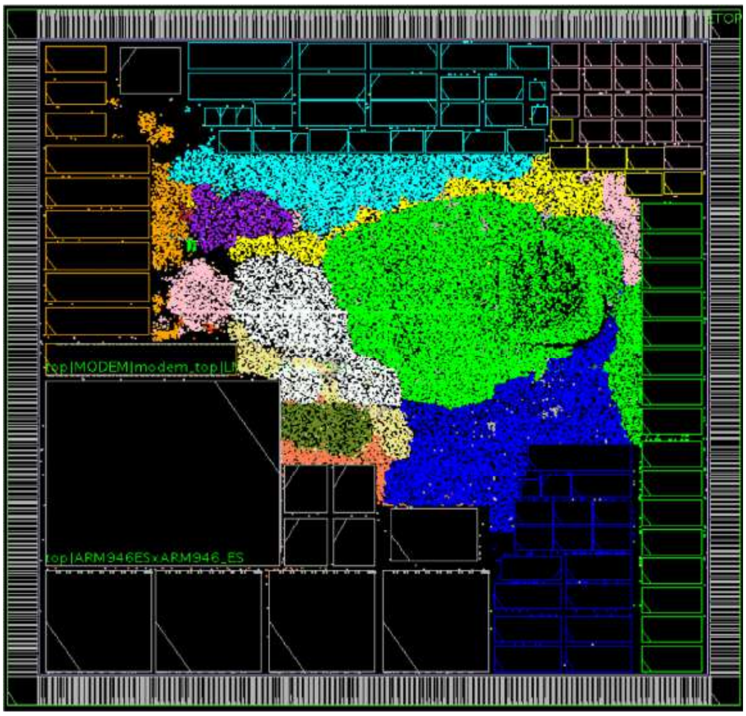


Tests after manufacturing

- By far one of the most expensive phases in the production of an integrated circuit, testing is performed by applying test patterns to the unit being tested and comparing the unit's responses with the expected outputs.
- Automatic test pattern generation (ATPG) tools use the circuit netlist to derive the sequence of the test vectors which exercise as many paths in the design as possible.
- Manufacturing tests can be classified into functional tests, diagnostic tests, and parametric tests.
- Functional tests are simple tests which determine if a chip is functional.
- Diagnostic tests are more involved since they aim at debugging the manufactured chip to determine which component in the chip has failed and possibly locate the fault within the component.
- Parametric tests check for clock skew, delay faults, noise margins, clock frequencies, etc. in the range of working conditions, such as supply voltage and temperature, for which the chip is supposed to function.

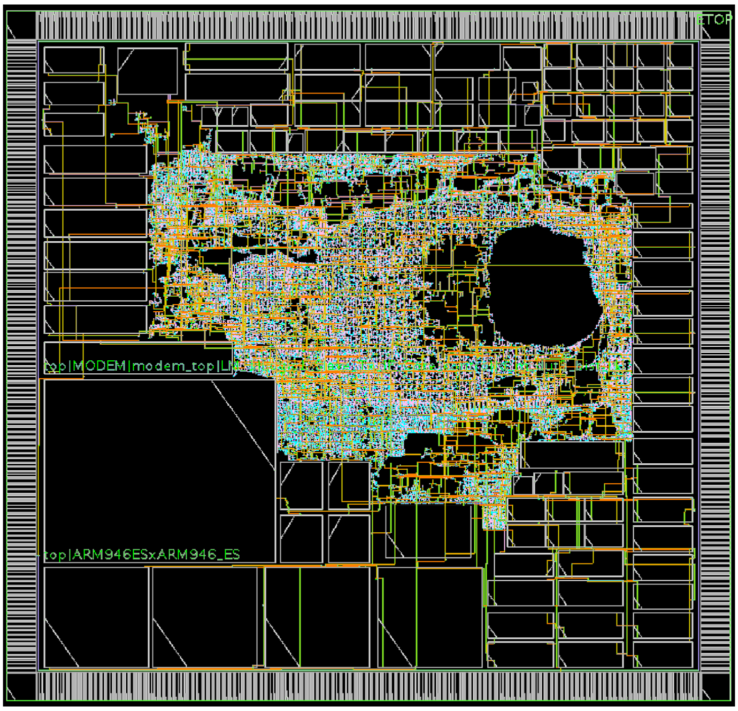
Example of fabricated chip (placement)

EASY project: Wireless LAN System-on-Chip

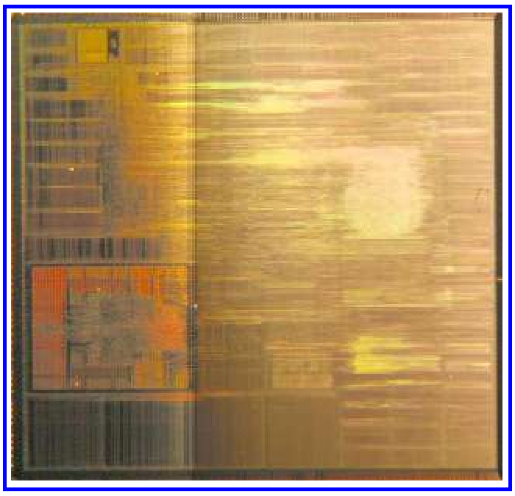


Example of fabricated chip (clock routing)

EASY project: Wireless LAN System-on-Chip



Example of fabricated chip



EASY project: WLAN System-on-Chip

Design	INTRACOM
Fabrication	ST

- Technology: 0.18 microns, Supply voltage: 1.8 V (core), 3.3 V (pads)
- Chip area: 88.51 sq. mm, Core area: 73.55 sq. mm
- Chip complexity: 4,400,000 equivalent gates (over 17.5 millions transistors !)
- Core area occupied by logic: 43.10 sq. mm, Core area occupied by memory: 30.45 sq. mm
- Total metal (six layers) interconnections: 47 m !, Power and clock wires: 10.1 m !

Conclusions

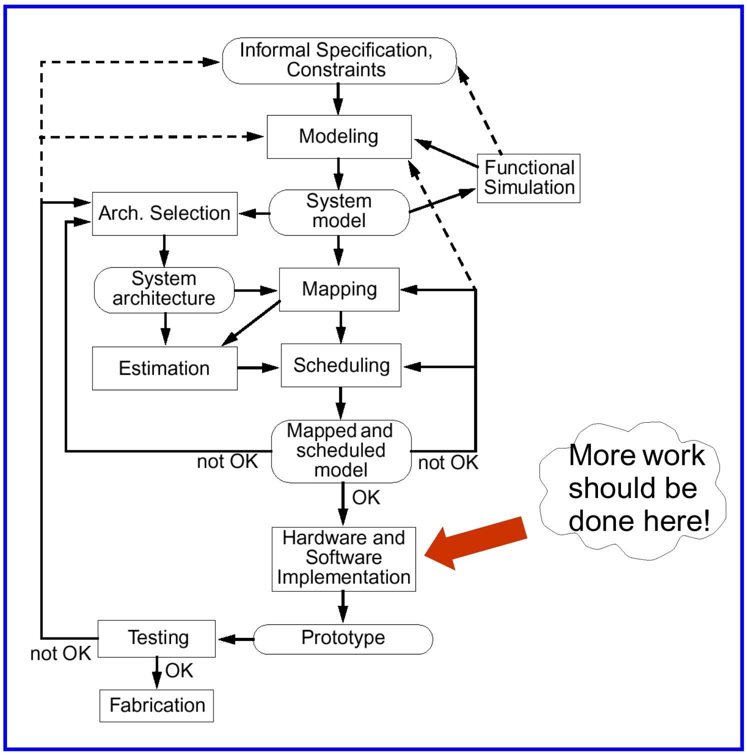
- As advances in semiconductor technology continue to provide the ability to put more on silicon with increasing circuit densities and performance, there is an increasing use of CAD tools to automate the design process.
- Complexity has also led to the extensive use of language-based approaches (HDLs) for digital design.
- At a lower level of abstraction, logic synthesis tools are used in order to handle large and complex designs.
- Design for testability is important in hardware design.
- The linking of the physical design and logic synthesis is becoming important since the effectiveness and accuracy of logic synthesis is impacted by the feedback and parasitic information provided by floorplanning tools.
- Efficient hardware simulators, emulators and prototyping environments are also required in modern hardware design.

3d. Low-level synthesis: Software generation

University of Thessaly

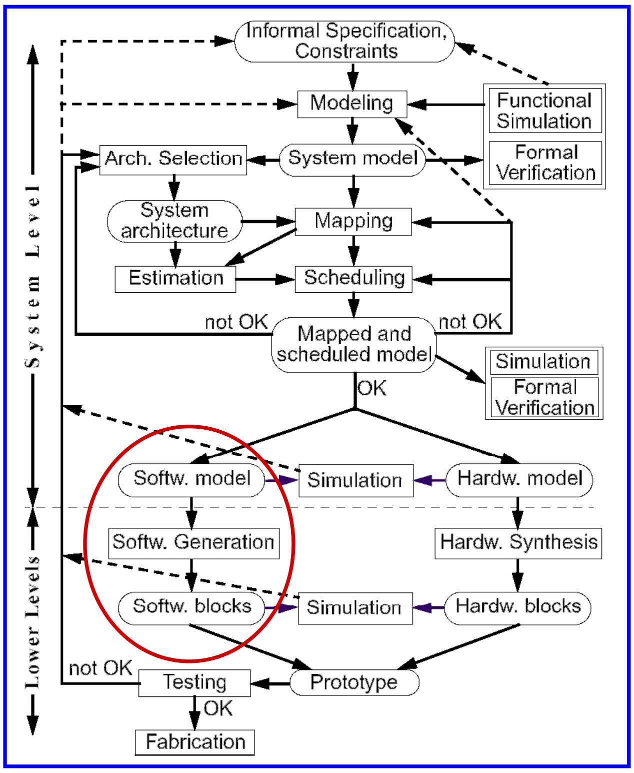
Department of Computer and Communication Engineering

Design flow



More work should be done here!

Design flow (cont'd)



- During the implementation phase, hardware and software components have to be developed (implemented) in a coordinated way.
- Hardware-software co-simulation is important.

Software generation

- There are some issues making different the software synthesis for embedded systems:
  - ✓ Increasing amount of software.
  - ✓ Different types of processors (CISCs, RISCs, DSPs, Microcontrollers, ASIPs etc.).
  - ✓ Strong time constraints are imposed.
  - ✓ Limited amount of memory available.
  - ✓ The hardware support and interaction has to be considered during the software generation.
  - ✓ Code optimisations are needed in order to make use of the particular features of the underlying architecture.
  - ✓ Very often the application is not based on a general purpose operating system, so an application-specific RTOS or a small RT kernel has to be generated together with the software.

Software generation (cont'd)

- Representing the program using a design pattern (computational model).
- Encoding in an implementation language (C, C++ etc.).
- Compiling and assembling for the target processor.
- Generation of a real-time kernel or adapting to an existing real-time operating system (RTOS).
- Testing and debugging (in a development environment).
- There are many available tools which perform automatically many of the low-level software implementation tasks:
  - ✓ Code generators (from software model to C).
  - ✓ Compilers, assemblers and linkers.
  - ✓ Test generators and debuggers.
  - ✓ Emulation and co-simulation tools and platforms.

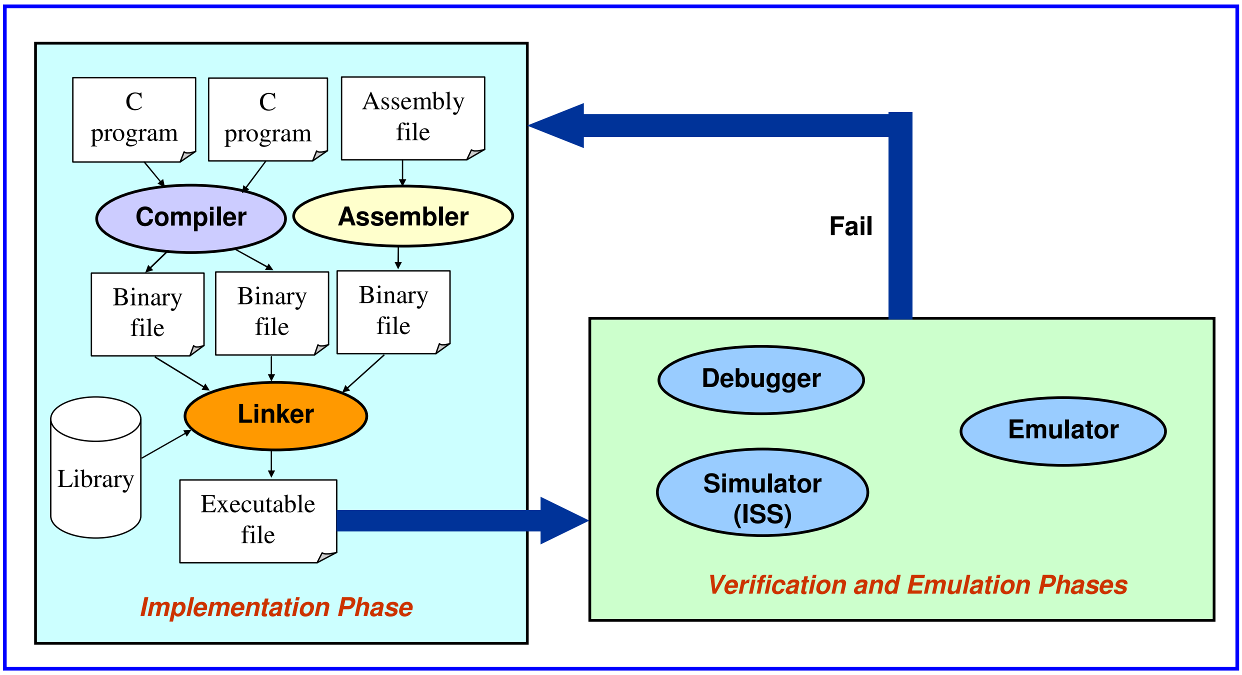
Software generation tools

- Usually, software generation has to do with two processors: the development processor (on which we write and debug our program) and the target processor (to which we will run our final program and which will form part of our embedded system).
- Compilers translate structured programs into assembly programs. Structured programming languages use high-level constructs that simplify programming, so each of them may be translated to several assembly instructions. Modern compilers perform optimization to reduce the code size improving the performance.
- In embedded systems development, cross-compilers are used in most cases. They are executed in the development processor, but generate code for the target processor.
- Assemblers translate assembly instructions to binary machine instructions.
- Linkers allow the creation of a program from separately-assembled files: combines the machine instructions of each program into a single program, perhaps incorporating instructions from a library.

Software generation tools (cont'd)

- Debuggers help the programmers to evaluate and correct their programs. They run on the development processor and support stepwise program execution. Whenever the program stops, the user can examine values of various memory & register locations.
- Instruction-set simulators (ISS) run on the development processor, but execute instructions of target processor.
- Emulators support debugging of the program while it is executed on the target processor (use of board hosting the target processor).
- In specific in-circuit emulators the board may have the capability for connection with the real embedded system.
- The availability of low-cost and high-quality development environments for a processor heavily influences the choice of a processor.

Software generation flow



Representing (modeling) & encoding the program

- Representing the program using a design pattern (computational model).

- A design pattern is a generalized description of the design for modeling programs (software tasks).

- The developer determines the details to customize the pattern (computational model) to the particular programming problem.

- State machines are useful in many contexts: parsing the user inputs, responding to stimulations, controlling the outputs.

- After the determination of the programming problem's states and the interaction between them, the program is implemented (encoded) in a high-level language (e.g. C, C++).

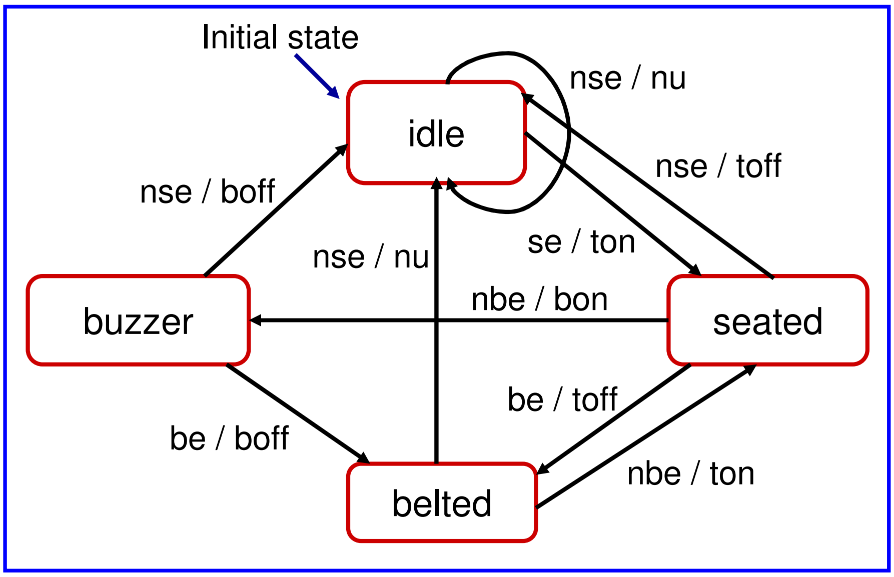
State machine example: Car belt controller

Inputs:

nse (no seat)
se (seat)
be (belt)
nbe (no belt)

Outputs:

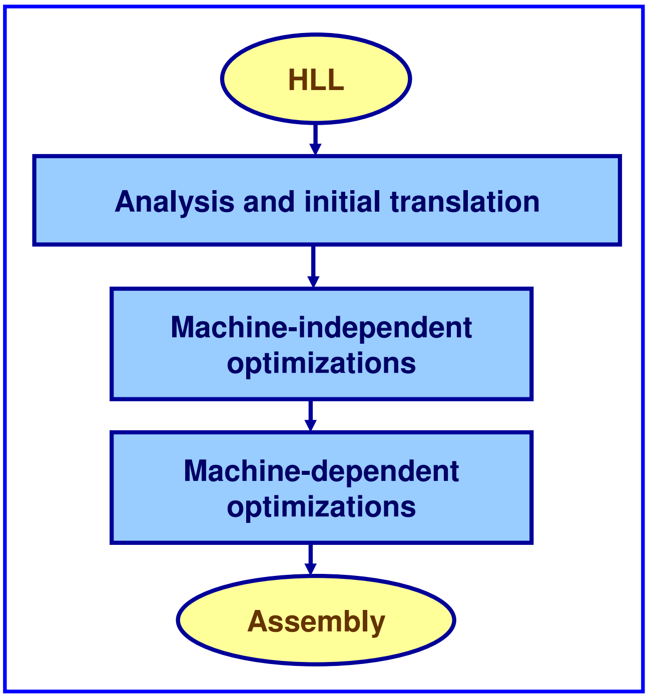
nu (null)
toff (timer off)
ton (timer on)
bon (buzzer on)
boff (buzzer off)



Implementation in C

```
#define IDLE 0
#define SEATED 1
#define BELTED 2
#define BUZZER 3
{
int state = IDLE;
switch (state) {
    case IDLE: if (no seat) { state = IDLE; }
               if (seat) { state = SEATED; timer on = TRUE; }
          break;
    case SEATED: if (belt) { state = BELTED; }
                 if (no belt) { state = BUZZER; buzzer on = TRUE; }
                 if (no seat) { sate = IDLE; timer off = TRUE;}
          break;
…
```

Compilation of the program



HLL

Analysis and initial translation

Machine-independent optimizations

Machine-dependent optimizations

Assembly

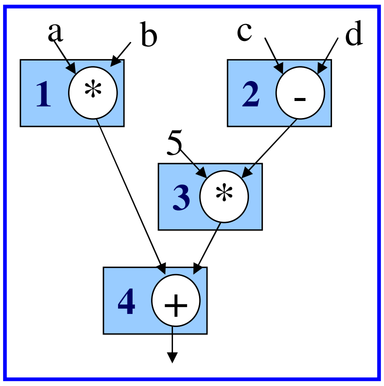
- Compilation is not only a translation to assembly, but also optimization of the code.

- Compiler has the ability to determine the quality of the code in terms of use of the CPU and memory resources, and code size.

Compilation of the program (cont'd)

- The source code is not a good representation for the compiler, so initially it is translated into an intermediate form (usually in a data flow graph), in order to be manipulated and optimised by the compiler.
- The representation should be low-level enough to allow optimization and estimation, and high-level enough to avoid excessive details (register allocation, instruction selection etc.)
- Graphs are then translated into instructions with optimization decisions, and then instructions are further optimised by the compiler.

Data Flow Graph



a*b + 5*(c-d)

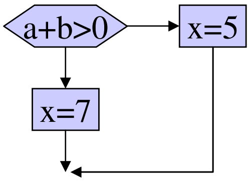
```
ADR r4,a
MOV r1,[r4]
ADR r4,b
MOV r2,[r4]
MUL r3,r1,r2

ADR r4,c
MOV r1,[r4]
ADR r4,d
MOV r5,[r4]
SUB r6,r4,r5

MUL r7,r6,#5

ADD r8,r7,r3
```

Corresponding Code

Compilation of the program (cont'd)

- Control code generation: representation of "if … then … else" high-level language instructions (control data flow graphs):

```
if (a+b > 0)
   x = 5;
else
   x = 7;
```



- Processes linkage: combination of several code modules into a single executable (code generation for passing parameters and results between the modules).
- Code module ordering: code modules are placed in specific positions in the memory space by using load map or linker flags.
- Simplification of the expressions (e.g. a*b+a*c = a*(b+c)), and dead code elimination.
- Loop transformations in order to reduce loop overhead, to increase opportunities for pipelining and parallelism, and to improve memory usage.

Compilation of the program (cont'd)

- Loop transformations:
  - ✓ Loop unrolling: reduces loop overhead.

```
for (i=0; i<4; i++)
    a[i] = b[i] * c[i];
⇨
for (i=0; i<2; i++)
a[i*2] = b[i*2] * c[i*2];
a[i*2+1] = b[i*2+1] * c[i*2+1];
```

  - ✓ Loop fusion: combines two loops into one.

```
for (i=0; i<N; i++) a[i] = b[i] * 5;
for (j=0; j<N; j++) w[j] = c[j] * d[j];
⇨
for (i=0; i<N; i++)
a[i] = b[i] * 5; w[i] = c[i] * d[i];
```

  - ✓ Loop distribution: breaks one loop into two.
  - ✓ Loop tiling: breaks one loop into a nest of loops.

Compilation of the program (cont'd)

- Register allocation: choose register to hold each variable, and determine lifespan of variable in a register.
- Instruction scheduling: in pipelined machines, execution time of one instruction depends on the nearby instructions.

FI	DI	FO	EI	SR		
	FI	DI	FO	EI	SR	
		FI	DI	FO	EI	SR

Fetch instruction (FI), Decode instruction (DI), Fetch operands (FO), Execute instruction (EI), Store results (SR)

- Instruction selection: there are several ways to implement an operation or sequence of operations, and the compiler has to find the optimum set of instructions for the implementation.
- After the compilation the assembler translates assembly instructions to binary machine instructions, by replacing opcodes and operands with binary equivalents, and translating symbolic labels into actual addresses.

Validation and testing of the program

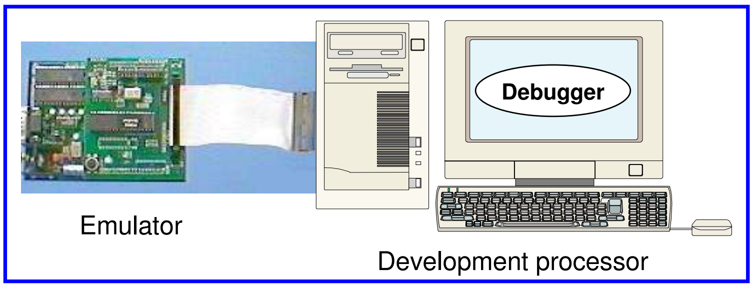
- The basic testing procedure contains three steps:
  - ✓ Provide the program with inputs.
  - ✓ Execute the program.
  - ✓ Compare the outputs with the expected results.
- By using debuggers the program is executed on the development processor in a stepwise manner. Whenever the program stops, the user can examine values of various memory and register locations, and compare them with the expected results.
- The use of instruction-set simulators gives us: control over time, set breakpoints, look at register values, set values, and step-by-step execution.
- Types of software testing:
  - ✓ Black-box testing: tests are generated without knowledge of program internals.
  - ✓ Clear box testing: tests are generated from the program structure.

Validation and testing of the program (cont'd)

- Clear-box testing requires:
  - ✓ Controllability: ability to cause a particular internal condition to occur.
  - ✓ Observability: ability to see the effects of a state from the outside.
- Clear-box testing generally tests selected program paths:
  - ✓ Control program to exercise a path.
  - ✓ Observe program to determine if the path was properly executed.
  - ✓ Branches and loops testing: tests all possible conditions.
- Black-box testing is made from the specifications (not from the code):
  - ✓ By selecting inputs from the specifications and determine the required outputs.
  - ✓ By generate random tests and determine the appropriate outputs.

Emulation of the program

- Emulators support debugging of the program while it is executed on the target processor.
- An emulator typically consists of the a debugger coupled with a board connected to the development processor via a cable. The board consists of the target processor plus some support circuitry.
- In in-circuit emulators the board has the capability for connection with the real embedded system.
- Such emulators enable the developers to control and monitor the program's execution in the actual embedded system.
- They can be quite expensive if they run at real speeds.



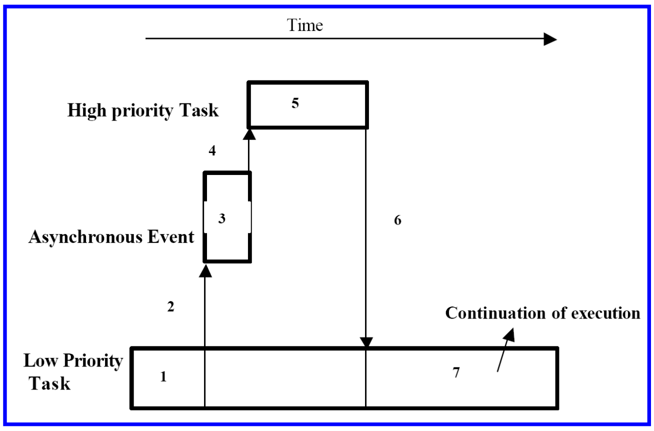
Device programmers are used to download a binary program from the development processor's memory into the target processor's memory.

Real-time operating system (cont'd)

- The real-time operating system (RTOS or RT kernel) is a layer of software providing low-level services to the application layer (set of software processes executed on the processor).
- It is responsible for the sharing of the central processing unit amount the executed processes (scheduling), according to the applied policy.
- Also, it is responsible for the sharing of the memory resources.
- Provides the software required for servicing various hardware interrupts as well as device drivers for driving the peripherals of the system (memory controllers, I/O devices etc.).
- Handles the software interrupts generated through system calls. The operating system provides to the application software an interface to the hardware through the system-call mechanism (implemented with ISRs – Interrupt Service Routines).

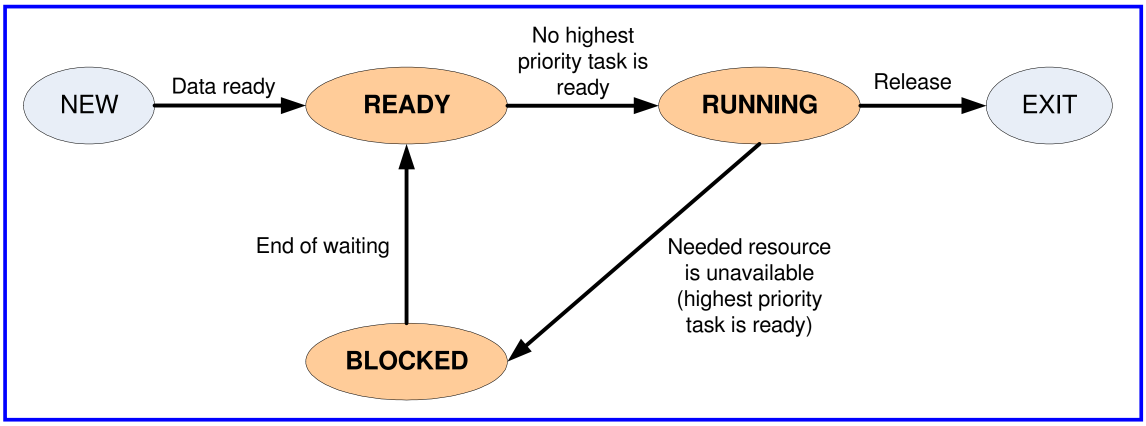
Real-time operating system (cont'd)

- RTOS can have two types of operation: non-preemptive and preemptive.
- In non-preemptive operation (that is not used much in RT applications), each task explicitly give up control of the CPU. A new task will gain control of the CPU only when the current task gives up the CPU.
- In preemptive operation, the RTOS itself ensures that the highest-priority task ready to run is always given control of the CPU.
- Asynchronous events (for changing the CPU control) are handled by ISRs. Upon its completion an ISR invoke the RTOS, which decides to run the more important task.



Real-time operating system (cont'd)

A system task may be in one of three states: running (currently executing), ready to execute or blocked (waiting). A task may not be able to execute until, for example, its data has arrived. Once its data arrives, it moves to the ready state.


NEW
Data ready
READY
No highest priority task is ready
RUNNING
Release
EXIT
End of waiting
Needed resource is unavailable (highest priority task is ready)
BLOCKED

RTOS state diagram

Real-time operating system (cont'd)

- Some desirable characteristics of an RTOS are:
  - ✓ Small kernel size.
  - ✓ Fast interrupt service (in general high performance).
  - ✓ User defined scheduling policy.
  - ✓ Support used language and microprocessor.
  - ✓ Compatibility with the used tools (compiler, assembler, linker, debugger).
  - ✓ Provision of the services need in the specific application.
  - ✓ Easy integration to our system.
  - ✓ Technical support.
  - ✓ Licensing issues.
- RTOS examples:

Commercial	Open source
VxWorks Embedded Linux Solaries OS-9 Windows CE	eCOS RTLinux Nut/OS

Conclusions

- There are several issues making different the software synthesis for embedded systems, such as the different types of used processors, the strong time constraints, the limited amount of available memory, the interaction with the hardware.

- Efficient compiler support is needed to perform the required code optimisations.

- Test generators, debuggers and simulation/emulation tools and platforms, are used in order to ensure the correctness of the developed software.

- Very often the application is not based on a general purpose operating system, so an application-specific RTOS or an RT kernel has to be integrated with the software.

3e. Power optimization in embedded systems

University of Thessaly

Department of Computer and Communication Engineering

Power consumption in embedded systems

- Embedded systems design consists of realizing a desired functionality while satisfying some design constraints such as performance (throughput, latency), size (silicon area in chip design), power consumption and cost.
- In recent years, the design trade-off of performance versus power consumption has received much attention, mainly due to the presence of large number of mobile systems that need to provide services with the energy releasable by a battery of limited weight and size.
- Recent design methodologies have addressed the problem of power optimization design, aiming to provide system realization while reducing its power consumption.
- When considering the digital part of an embedded system, we can distinguish three major types of units consuming significant power: computation units, communication units and storage units.

Power consumption in embedded systems (cont'd)

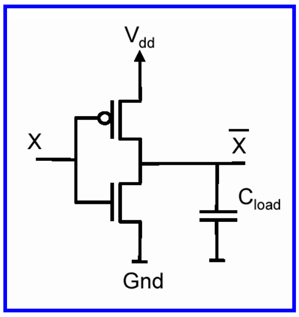
- Since software does not have a physical realization, we need to analyze the impact of embedded software execution on the system's power consumption.
- Choices for software implementation affect the power consumption of embedded systems.
- For example, software compilation affects the instructions used by the computing elements, each one bearing a specific energy cost.
- Software storage and data access in memory also affects power balance, and data representation (e.g. encoding) affects power consumption of communication resources (e.g. buses).
- Power optimization techniques can be applied at all major design phases: modeling, system-level synthesis, and lower-level synthesis (hardware synthesis, software generation).

CMOS power consumption

- Average power consumption in digital CMOS circuitry:

$$\boxed{P_{avg} = P_{dynamic} + P_{short-circuit} + P_{leakage} + P_{static}}$$

Static CMOS gate



  - ✓ $P_{dynamic}$ is the power consumed due to charging and discharging of the capacitive loads, and is given by the product of the load capacitance, the square of the supply voltage and the frequency.
  - ✓ $P_{short\text{-}circuit}$ is the power consumed due to short-circuit currents between the supply rails during switching.
  - ✓ $P_{leakage}$ is the power consumed due to leakage currents.
  - ✓ P_{static} is the static power consumption occurred in some CMOS implementations.
- The first two components are strongly dependent on the transition activity of the circuitry (switched capacitance).

CMOS power consumption (cont'd)

$$P_{dynamic} = s\, C_L V_{DD}^2 \frac{1}{T} = s\, C_L V_{DD}^2\, f$$

$$T_d \propto \frac{C_L V_{DD}}{K(V_{DD} - V_{TH})^{\alpha}}$$

s: switching activity factor

- The power consumption is reduced by lowering the supply voltage. However, in this case the delay will be increased.
- The reduction of the load parasitic (switching) capacitance may lead to simultaneous improvement of power consumption and delay of a logic block. However, the driving capability of the subsequent block will be reduced, increasing its delay.
- Due to the fact that power and delay are conflicting metrics, our design goal has to be the trade-off between speed and power.
- The power consumption is reduced by applying slower frequency. Then, the circuit will become slower, and we have to increase parallelism in order to compensate the speed loss (with an area overhead).

Power optimization at algorithmic level

- The most abstract representation (model) of a system is the function it performs.
- The choice of the algorithm for performing a function (whether implemented in hardware of software) affects the system's power consumption.
- In addition, the choice of the parameters for the implementation of an algorithm (e.g. word-width) is a degree of freedom that can be used to reduce the power consumption.
- After specification/modeling, the designer take the key decisions on the system architecture (what is the hardware support required for implementing the functionality with the selected algorithm).
- Architectural choices such as type of used processors, use of specific circuits for power-hungry tasks, may have significant influence on the system's power consumption.

Power optimization at algorithmic level (cont'd)

- Assume the existence of a library with multiple algorithms for computation of common functions, and that the power and performance of the library elements are pre-estimated for a given generic architecture (i.e. processor).
- For each function call in the specification, we can select the algorithm that minimizes power consumption while satisfying performance constraints.
- Also, the data structure have impact on the power consumption (e.g. number representation in DSP systems).
- Computational energy can be reduced by applying data-flow transformations in DSP algorithms.
- Storage energy can be reduced if we consider that results computed early are needed much later in time, and thus the storage requirements are increased (we have to exploit locality).
- Also, a highly parallel computation with significant communication between parallel threads requires a complex and power-hungry communication infrastructure.

Power optimization at algorithmic level (cont'd)

- Computational kernels are the inner loops of a computation, where the most time is spent during execution.
- Profiling an algorithm execution flow under typical input streams can easily detect computational kernels. Profiling data are collected from the executable specification.
- To substantially reduce power consumption, each computational kernel is optimized as a stand-alone application and implemented on dedicated hardware that interfaces with the less frequently executed sections of the algorithm.
- During system operation, when the computation is within a computational kernel, only the kernel processor is active (and dissipates power), while the rest of the system can be shut down. Otherwise,the kernel processor is disabled.
- Performing a computation on dedicated hardware is usually one or two orders of magnitude more power efficient than using a general purpose processor.

Power optimization at algorithmic level (cont'd)

- Approximate processing: the key idea here is that power consumption can be drastically reduced by allowing some inaccuracies in the computation (e.g a video user may be satisfied with low video quality when watching a television show, but they may require high quality when reading a page on screen).

- Approximate computation algorithms can adapt the quality of service to power constraints and user requirements.

- Certain operations may be implemented with limited accuracy to reduce energy costs. For example, a cos(x) function can be approximated as a Taylor expansion.

Power optimization at the system level

- The balance between software and hardware can vary widely, depending on the application.

- Even though dedicated hardware is more energy efficient than software running on processors, the latter has many compensating advantages, such as flexibility, ease of late debugging, low-cost, and fast design time.

- Example: a processor can easily implement a FIR filter by performing a sequence of multiplications and additions. On the other hand, a custom hardware architecture for FIR filtering can be created with just an adder, a multiplier and few registers, which results in quite lower power consumption.

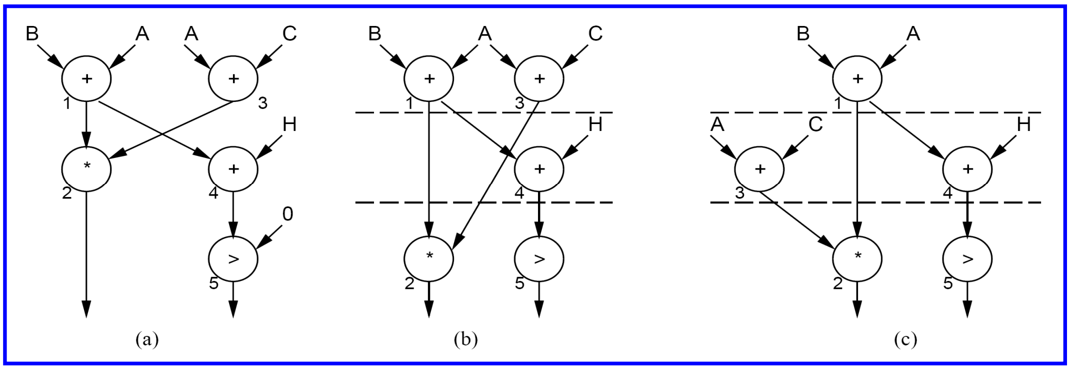
Power optimization in application-specific circuits

- The most direct way to reduce power is to scale down the supply voltage. Unfortunately, CMOS circuits get slower as the supply voltage decreases.
- Under these conditions, a good approach to reduce power consumption is to make a circuit faster than its performance constraint, and then decrease the supply voltage until the constraint is matched again (power-driven voltage scaling).
- To make the circuit faster, techniques for high-performance computation such as parallelization and pipelining are used.
- The limitation in the pipelining technique is the registers overhead.
- In parallelization speedup technique, the duplication of a part of the datapath to allow more that one computation at the same time, imposes large area overhead.
- Another limitation of power-driven voltage scaling is that it becomes less effective as technology scales down, due to the fact that voltage supplies are moving to lower levels reducing the room available for voltage scaling.

Power optimization in application-specific circuits (cont'd)

- The power-driven voltage scaling can be extended by allowing multiple supply voltages on a single system.
- The key idea is to save power in noncritical functional units by powering them with a down-scaled supply voltage. In this way throughput remains unchanged, but the overall power consumption is reduced.
- If we allow multiple clock domains on a single system, we can clock noncritical subsystems at slower frequencies, thereby saving significant power without reducing overall system performance.
- If a system component is idle (i.e. it is not performing any useful work), we can set its clock frequency to zero and nullify dynamic power consumption. This technique is known as clock-gating.
- Reduction of switching activity: One way is to reduce the number of operations (i.e. by transforming the dataflow graph of a computation comprising of elementary operations), and a second way is to increase the correlation between successive patterns at the input of functional units.

Power optimization in application-specific circuits (cont'd)



(a) (b) (c)

- The constraint on execution time is 3 clock cycles, the resource constraints are: 2 adders, 1 multiplier, 1 comparator.
- (b) and (c) show two schedules compatible with the resource constraints.
- Observe that additions 1 and 3 share one of the operands. If we perform both additions with the same adder, its average switching per operation will be low, because one of the operands remains the same.
- In the schedule (b) it is not possible to implement additions 1 and 3 with the same adder, because the two operations are scheduled in the same control step. On the other hand, the schedule (c) allows sharing, and it leads to a more energy-efficient implementation.

Power optimization in processors

- Even though dedicated computation units are very energy efficient, they are not flexible.
- In many cases, flexibility is a primary requirement, either because initial specifications are incomplete or because they change over time or because computation units must be reprogrammable to some degree.
- From the energy-efficiency viewpoint, processors suffer from three main limitations:
  - ✓ They have an intrinsic power overhead for instruction fetch and decoding.
  - ✓ They tend to perform computations as a sequence of instruction executions, and cannot take advantage of algorithmic parallelism.
  - ✓ They can perform only limited number of elementary operations, as specified by their instruction set, and thus they must reduce any complex computation to a sequence of elementary operations.

Power optimization in processors (cont'd)

- In order to be more energy efficient many processor families have "low-power" versions with reduced supply voltages. Supply voltage reduction requires some adjustments in device technology and in circuit design (critical path redesign etc.), in order to keep performance at a desired level and ensure correct functionality.
- The energy efficiency in processor design can be enhanced by using the clock-gating technique in order to avoid useless switching activity in idle units.
- Specialized instructions are often used as a power and performance enhancing technique. The basic idea is to provide few specialized instructions and the required architectural support that allow a processor to execute in a more efficient way.
- For example in some RISC processors variables and intermediate results of data-processing instructions can be stored in registers avoiding repeated transfers from/to memory.
- Compilers in such processors handles the optimization of the registers' use (more complex compilers).

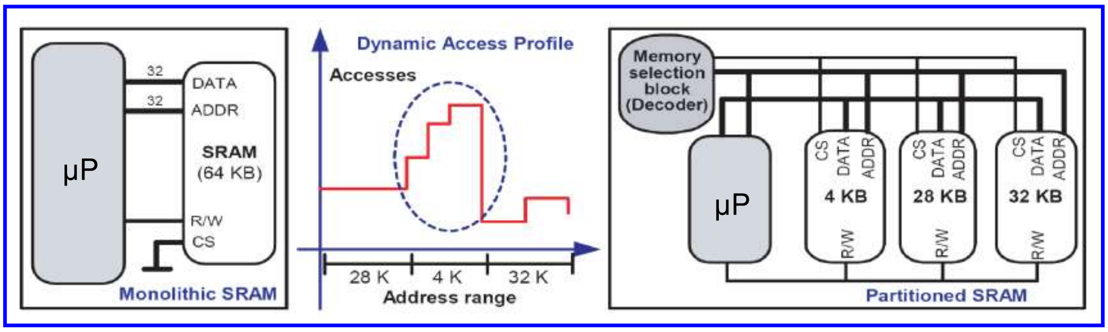
Power optimization in ASIPs

- When designing a processor for a specific application, energy (and performance) optimization can take advantage from the knowledge of the target application to obtain a highly optimized specialized processor.
- ASIPs (application specific instruction-set processors) are a compromise solution between fixed processor cores and dedicated functional units.
- The energy efficiency in ASIPs is based on the fact that they use a reduced number of instructions (able to support the target application).
- The target application is first compiled for a complete instruction set, then the executable code is profiled, and instructions that are never used or that can be substituted by others are dropped from the application specific instruction set.
- After that the processor with the reduced instruction set is synthesized and the code is executed on it.
- The instruction set reduction also leads in simpler decoding and execution units, which enhance the energy efficiency.

Power optimization in memories

- During the algorithmic design, the designer has to improve locality i.e. the results of a computation should be “consumed” by subsequent computations as soon as possible, thereby reducing the need for temporary storage, thus reducing the memory accesses.
- Also, the designer has to use efficient data representations that reduce the amount of inessential information stored in memory (i.e. data compression).
- In some specific processors variables and intermediate results of data-processing instructions can be stored in registers avoiding power consuming repeated accesses to memory (register allocation).
- From the architectural point of view, in order to reduce the power consumed due to memory accesses we can exploit the concept of memory hierarchy.
- Low hierarchy levels are made of small memories, close to computation units and coupled with them, while high hierarchy levels are made of larger memories far from computation units and shared.
- Each memory level can be partitioned into several independent blocks (banks) in order to reduce the cost of a memory access in a given level.
- However, memory partitioning leads to higher addressing complexity and area overhead.

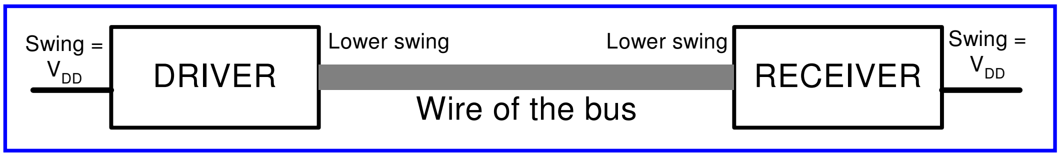
Power optimization in memories (cont’d)



- The profile (obtained by instruction-level simulation) gives for each address in the range, the number of reads and writes to the memory during the execution of the target application.
- A small subset of the addresses is accessed very frequently. A power-optimal partitioned memory organization consists of three memory cuts and a memory selection block.
- The larger cuts contain the top and bottom part of the range, while ‘hot’ addresses are stored into a small memory.
- The average power in accessing the memory is decreased, because a large fraction of accesses is concentrated on a small memory, and the memory banks that are not accessed are disabled through chip select (CS).

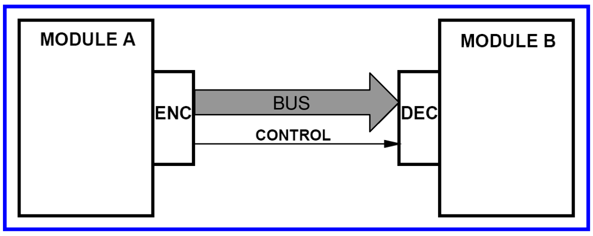
Power optimization in communication

- Communication is carried out over a set of metal wires (bus). The total capacitance of wires supporting communication is much larger than the average capacitance of local wires.
- Communication power can therefore be reduced by either scaling down the voltage swing or reducing the average number of signal transitions.
- Swing reduction: We can decrease quadratically the power consumption by reducing the voltage swing on the high-capacitance wires of a bus.
- Low-swing signalling is beneficial also for performance because it takes less time for a signal to complete a small swing than a large swing.
- The trend in reducing voltage levels is limited by noise margins. The reduction of noise margins affects the circuits reliability.



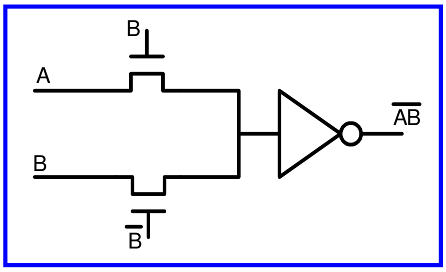
Power optimization in communication (cont'd)

- Signal encoding technique for power reduction on buses.
- The basic idea is to encode the binary data send through the communication channel (bus) in order to minimize its average switching activity, which is proportional to dynamic power consumption.
- The data from the source modules is encoded, transmitted on the bus and decoded at the destination module.
- Buses have relatively large parasitic capacitance, so that the energy dissipated in data transfers is significant and dominates the extra energy required to encode and decode the signals at both sides.
- There are several low-transition activity encoding and decoding schemes (gray, bus-invert, T0, adaptive etc.).



Power optimization at circuit level

- Static CMOS circuit design technique has a major low-power characteristic: power is only consumed during output switching – no static power is dissipated except leakage.
- Alternatively in some cases other logic styles can be used.
- For example logic styles based on pass transistors may be beneficial in designing arithmetic circuits (adders, multipliers) due to their specific characteristics such as reduced transistor count and compact layout.



NAND2 gate

- Transistor sizing: appropriate sizing of transistors in CMOS circuits can be applied for minimizing the power consumption under a given delay constraint.
- Driving high capacitive loads (e.g. long wires) with proper drivers (inverter chains).

Power optimization at technology level

- Deep-submicrometer technologies and nanotechnologies are used in order to reduce the physical (parasitic) capacitance and thus the power consumption of CMOS circuits.
- Alternative technologies (Silicon-on-Insulator, SOI) and Multi-Chip Modules (MCMs) fabrication for the same reason.
- Multiple threshold voltage technologies:
  - ✓ Assignment of low threshold voltages to critical paths to meet the required frequency.
  - ✓ Assignment of high threshold voltages to the rest circuitry in order to minimize leakage (subthreshold current).

$$\boxed{T_{\mathrm{d}} \propto \frac{C_{\mathrm{L}} V_{\mathrm{DD}}}{K(V_{\mathrm{DD}} - V_{\mathrm{TH}})^{\alpha}}}$$

$$\boxed{I_{ds} = K e^{(V_{gs}-V_{th})/nV_T}\left(1 - e^{-\frac{V_{ds}}{V_T}}\right)}$$

Power optimization in software

- Software execution corresponds to performing operations on hardware, as well as accessing and storing data.
- Thus software execution involves power consumption for computation, storage and communication.
- The energy cost of executing a program depends on its machine code and on the corresponding processor architecture.
- Since the machine code is derived from the source code through compilation, it is the compilation process that also affects energy consumption.
- The energy cost of machine code depends on:
  - ✓ Type of operations.
  - ✓ Number of operations.
  - ✓ Order of operations.
  - ✓ Way of storing data (addressing modes, use of registers vs. memory arrays etc.).

Power optimization in software (cont'd)

- The traditional compiler goal is to speed up the execution of the generated code by reducing code size (which is related with latency in execution time).
- Executing machine code of minimum size consumes minimum energy, if we neglect the interaction with memory and assume a uniform energy cost for each instruction.
- The development of specific low-power compilers is still an open field for research.
- Examples of compiler power optimizations are:
  - ✓ Selective loop unrolling, which reduces the loop overhead.
  - ✓ Instructions selection in order the execution of the code to access registers which is much less energy consuming than accessing memory.
  - ✓ Instruction scheduling (reordering) for low energy by reducing the interinstruction effects that cause switching on the instruction bus, and in some other processor units such as instruction decoder.
  - ✓ Note that, the priority of scheduling an instruction is inversely proportional to the Hamming distance from the currently scheduled instruction.

Power optimization through RTOS

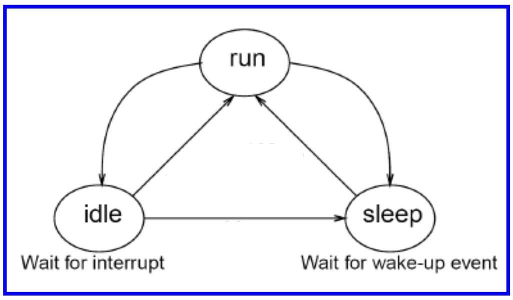
- The heart of any RTOS is the task scheduler and apart from satisfying the real-time constraints, has to be energy-aware.
- Energy-efficient task scheduling is called dynamic power management (DPM) and it is a design methodology that dynamically reconfigures an embedded system to provide the requested services and performance levels with a minimum number of active components.
- DPM contains a set of techniques that achieve energy-efficient computation by selectively turning off (or reducing the performance of) system components when they are idle (or partially unexploited).
- A power manager implements a control procedure based on some observations and/or assumptions about the workload of the system.
- An example of power management is the shutdown of a component in laptops after a fixed inactivity time, under the assumption that it is likely that the component will remain idle if it has been idle for a specific period.

Power optimization through RTOS (cont'd)

- For clocked hardware units, energy can be saved by reducing the clock frequency (or by stopping the clock), or by reducing the supply voltage (or by powering off the unit).
- Power shutdown is a solution that eliminates all sources of power consumption, however a major disadvantage is the wake-up recovery time, which is higher than in clock-gating solution.
- There are several reasons for migrating the power manager to software (RTOS):
  - ✓ Software power managers are easy to write and to modify.
  - ✓ The RTOS can manage computational, storage and I/O tasks of the system.
- Recent power managers include workload predictive techniques and they can adapt the clock speed and the supply voltage in several components of the system.

Power optimization through RTOS (cont'd)

- The StrongARM SA-1100 processor is an example of power manageable component, and has three modes of operation: run, idle, sleep.
- Run mode is the normal operating mode (every on-chip resource is functional). The chip enters run mode after successful power-up and reset.
- Idle mode allows a software process to stop the CPU when not in use, while continuing to monitor interrupt requests. In idle mode, the CPU can be brought back to run mode quickly when an interrupt occurs.
- Sleep mode offers the greatest power savings, and consequently the lowest level of available functionality.
- In the transition from run or idle, the processor performs shutdown. In a transition from sleep to any other state, the chip performs a complex wake-up sequence before it can resume normal activity.



Conclusions

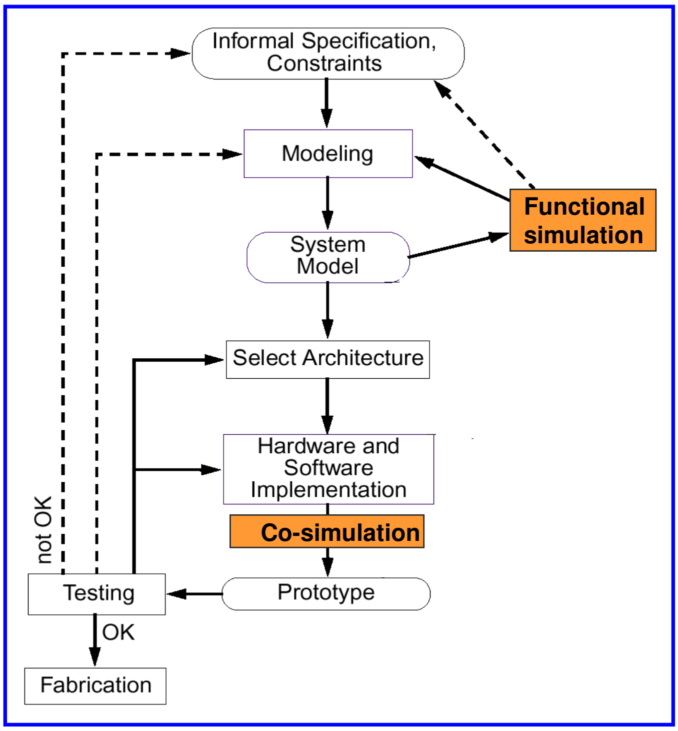
- In recent years, power consumption is one of the most critical design parameters, mainly due to the presence of large number of mobile systems.
- Embedded systems design aims at achieving a balance between performance and energy efficiency.
- Power optimization techniques can be applied in both hardware and software domains and in all levels of the design hierarchy: algorithmic, system, circuit, technology.
- In addition, energy-efficient design must target all types of resources that can be sources of power consumption: computation, communication and storage resources.

4. Verification and Co-simulation

University of Thessaly

Department of Computer and Communication Engineering

Simplified design flow



Verification of embedded systems

Verification is the process of determining that a design is correct and should include:

- Specification verification (functional simulation): checks if the developed system model satisfies the initial functional specification of the system. It is a solution to deal with high complexity, i.e. to move to higher levels of design abstraction for the system's functional verification.

- Implementation verification: checks if a lower-level model resulted after one or several refinement steps, correctly implements a higher-level model. This can be performed by co-simulation at several levels of abstraction and by prototyping (hardware-software prototyping platforms) before the final stage of the implementation.

Specification verification

- The specification verification is a process that proves mathematically that several properties of the system are true or that the functionality of the system is correct, based on a developed specification model.

- After the initial specification of functionality and a specific set of constraints (time, power etc.), a more formal specification of the functionality, based on some modeling concept (e.g. FSMs or Data-flow models) is generated.

- Then, this model (system's executable specification) that can be in Matlab, C, UML etc., is simulated in an execution environment with a proper test bench (inputs to the system), in order to check the correctness of the initial system functionality.

Simulation process

- The simulation process occupies an increasing part of the total development time for real-time systems and today it is often the bottleneck in the development process.

- The increase in time for the simulation stage is mainly dependent on three parameters:
  - ✓ Increased software complexity.
  - ✓ Increased hardware complexity.
  - ✓ Complex interaction between hardware and software.

- To shorten the development process it is a key demand to decrease the simulation and in general the verification time.

Simulation process (cont'd)

- The simulation processes for software and hardware design, have traditionally been two completely separated activities within the total design process.

- In traditional design of a mixed (hardware-software) project, software is typically developed after the hardware design has begun to stabilize and prototypes are available for integrating and testing the software and the hardware.

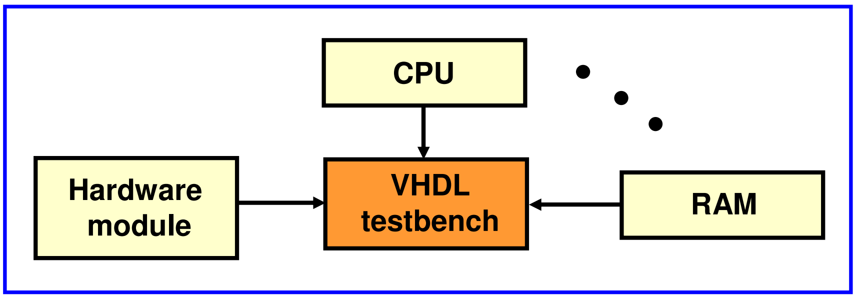
- The hardware-software integration and simulation is the most time consuming part of the project, due to the fact that there is limited observability of the operation of the hardware as the software executes and inability to control all of the elements of the design (especially the peripherals running synchronously with a microprocessor).

Software simulation methods

- In the absence of target hardware, verification of software code was mostly managed using development tools such as debuggers and simulators running in a host-machine environment.
  - ✓ Native compiled software (NCS): the software is compiled for the host processor and then executed on it. Debugging of NCS can then be done by the debug tools of the host processor.
  - ✓ Emulation of the target processor instruction set (Instruction-Set Architecture - ISA model). Software verification is performed through instruction-set simulation.
  - ✓ In both cases the software has to be complemented with additional code that simulates the absent hardware components. Software components called stubs are used to simulate the required interfaces.
- Another method (more realistic and more accurate) for software verification is to use existing (incomplete) processor hardware as a prototype. Incomplete hardware can be processor development boards or FPGA prototyping boards (drawback: software is verified late in the design process).

Hardware simulation methods

- On the hardware side typically the designer is interested in verification of the interaction (accesses, handshaking, interrupts, signals) between software and specific hardware components.
- One approach is to use a testbench in which the hardware module to be simulated is instanced as a component.
- By using models of the surrounding components (e.g CPUs, memories) stimulation inputs can be generated, thus enabling simulation of the responses according to specification.



Hardware simulation methods (cont'd)

- Models of surrounding components is difficult to be implemented and usually they are not available.
- A solution is to use specific tools (often custom and application-depended) that produce input files (force files) for the VHDL testbench. These files are produced according to the specification.
- These force files can be used, for example, to fill the memories of the hardware module to be simulated with the valid contents or to produce the proper control signals, “substituting” the processor running the embedded software.
- Finally, the system hardware can be verified by prototyping (FPGA-based hardware-software prototyping platforms) before the final stage of the implementation.
- The major advantage when using FPGAs is the ability to make fast changes compared to ASIC, and the disadvantage is that the timing is much slower than that of the ASIC.

Co-simulation

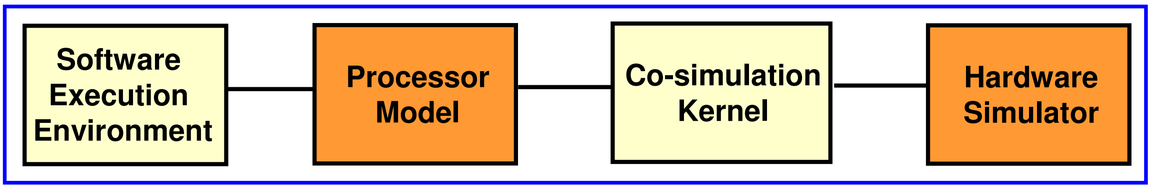
- The co-simulation problem: How to simulate hardware and software components at the same time.
- Difficulties:
  - ✓ Different simulation platforms are used
  - ✓ Software runs fast while hardware is relatively slow; how to run the system simulation as fast as possible, and keep the two domains synchronized.
  - ✓ Slow models provide full details and produce accurate results, while fast models do not produce enough timing information and simulation is less accurate.

Co-simulation (cont'd)

- Co-simulation has been introduced as an alternative to the use of testbenches.
- The target is to couple a software execution environment with a hardware simulator.
- Co-simulation allows two engineering groups to “talk” together.
- Co-simulation allows earlier integration of the system and provide a significant performance improvement for system verification.

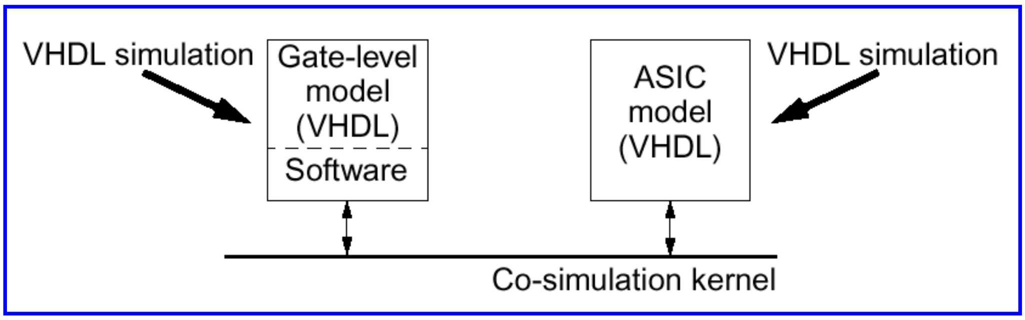
Co-simulation (cont'd)

- In co-simulation approaches:
  - ✓ Software is executed on a simulator running on the host processor (e.g. instruction-set simulator by using an instruction-set architecture – ISA model of the processor) or on the actual hardware platform.
  - ✓ An emulation (model) of the processor is used or its actual hardware platform.
  - ✓ Several methods are used for its interface with the hardware part of simulation.
  - ✓ A kernel is used to co-ordinate the communication between the two parts of simulation.



Basic co-simulation approaches

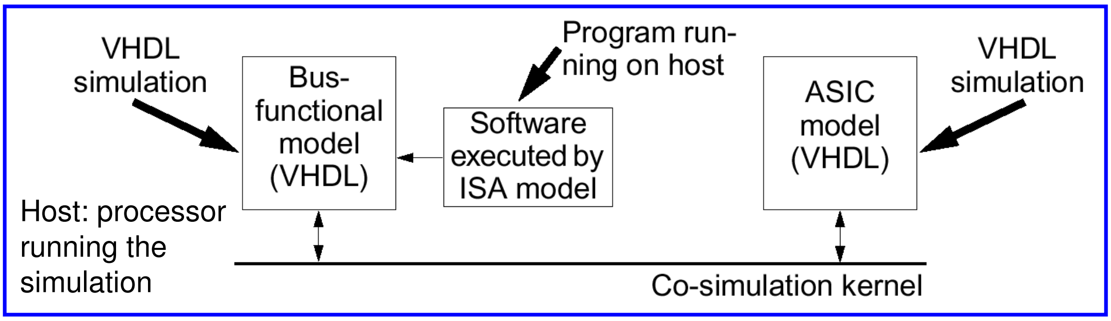
1st approach: Use a detailed (gate-level of RTL-level) processor model



- Processor components (memory, data path, instruction decoder etc.) are discrete models that execute the embedded software.
- The software is running on the hardware model of the processor, i.e. it is the event driver of the VHDL model.
- Interaction between processor and other components is performed using event-driven simulation capabilities of the hardware simulator.
- The co-simulation kernel is a controlling unit (implemented in software) that co-ordinates the interface/communication between the two parts of simulation.

Basic co-simulation approaches (cont'd)

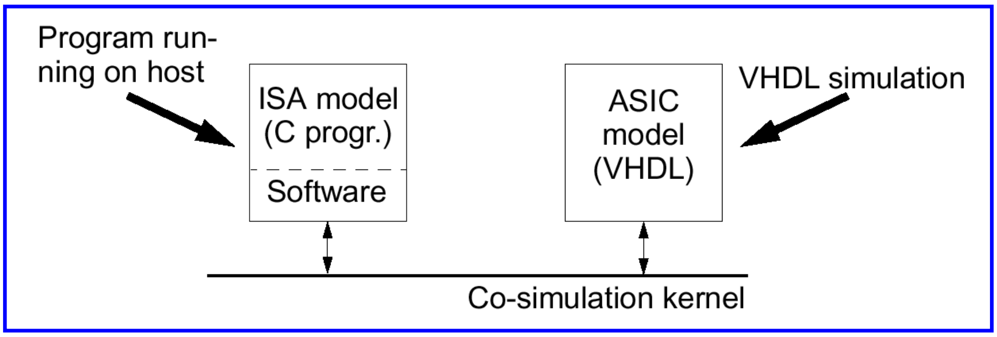
2nd approach: Use a bus model of the processor (cycle-based simulation)


VHDL simulation
Bus-functional model (VHDL)
Program running on host
Software executed by ISA model
ASIC model (VHDL)
VHDL simulation
Host: processor running the simulation
Co-simulation kernel

- The bus model of the processor only simulates the activity of bus interface of the processor without executing the software
- The bus model converts the software operations of the processor to I/O operations. The events to be handled by the VHDL simulator are reduced to the ones of the processor I/O signals.
- The software is executed on an instruction-set model of the processor (e.g ARMulator for ARM processors family) and provide timing information in clock cycles (clock cycles number required for a given sequence of instructions).
- Less accurate but faster simulation model.

Basic co-simulation approaches (cont'd)

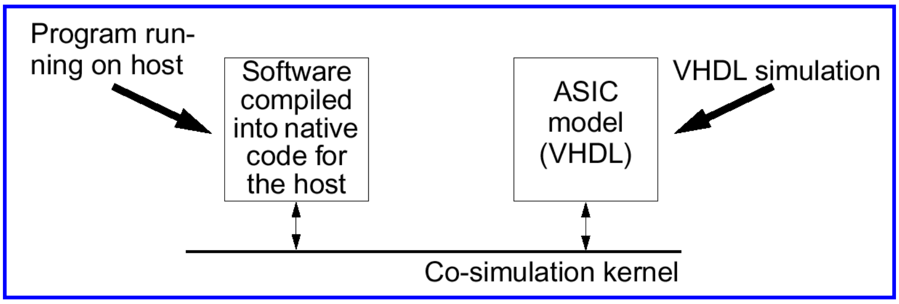
3rd approach: Use an instruction-set model of the processor



- There is no hardware model of the target processor. The software is executed on an instruction-set model (usually written in C).
- The execution of the software on the instruction-set model provides interface information (including timing) needed for co-simulation.
- Simulation is fast, but timing accuracy depends on the interface information.
- Example: The Seamless tool (kernel) implements an interface between ISS model and the HDL simulator that executes the hardware models of the peripherals.

Basic co-simulation approaches (cont'd)

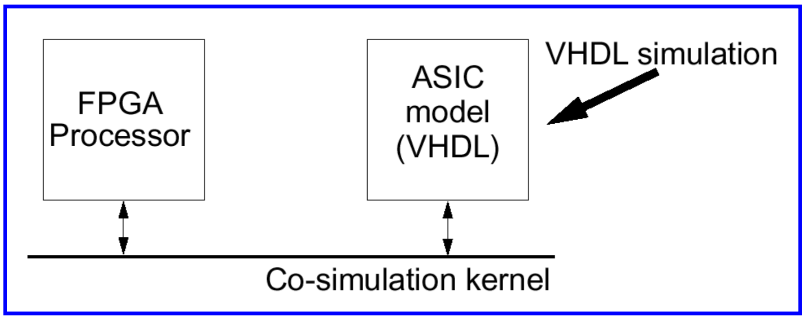
4th approach: Use a compiled model of the processor



- There is no hardware model of the target processor. The software is compiled in native code for the host processor.
- The native code models are fast models translating the embedded software into native code for the processor doing the simulation (e.g. code for a DSP can be translated into Sparc assembly code for execution on a workstation).
- The execution of the native code provides interface information (including timing) needed for co-simulation.
- Simulation is very fast, but timing accuracy depends on the interface information.

Basic co-simulation approaches (cont'd)

5th approach: Use the processor's physical hardware or emulate the processor by FPGA prototyping



- If the processor exists in hardware form, the physical hardware can often be used to model the processor in the simulation.
- Alternatively, the processor could be modeled (emulated) using an FPGA prototype.
- Advantage: simulation speed.
- Disadvantage: Availability of the physical processor or availability of the FPGA-based prototype.

Summarizing the co-simulation approaches

- The processor that executes the embedded software can be modeled with the following approaches:
  - ✓ Detailed processor models supported by hardware simulators.
  - ✓ Bus models that only simulate the activity of bus interface of the processor without executing the software.
  - ✓ Instruction-set (IS) models usually implemented in a C program that interprets the embedded software.
  - ✓ Compilation of the embedded software into assembly code for the processor doing the simulation.
  - ✓ Physical processor or FPGA-based prototyping of the processor.

Domains coupling

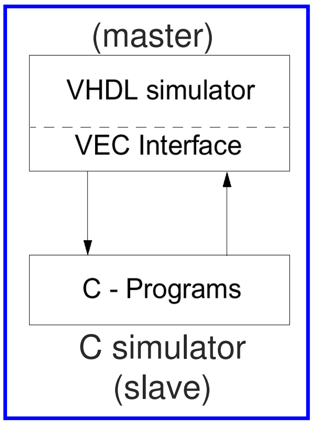
- In the three of the five previous approaches a program running on the host, which simulates the software component, interacts with the hardware simulator.

- Main problems:

  ✓ Providing timing information across the boundaries.

  ✓ Coupling of the two domains.

Possible solutions



- There is an attribute in VHDL that allows parts of the code to be written in languages other than VHDL.

- These parts of the code, named VEC procedures (VHDL emulation of C procedures), perform data conversion and call the C functions.

- Disadvantages:
  ✓ The C program has to be re-organised as C functions that can be called by the VEC procedures.
  ✓ No concurrent simulation is possible. This problem can be solved with the communication of the simulator with the C programs through a software bus (that can be a modelled component in the simulation tool).

Conclusions

- The basic problem of co-simulation is how to simulate hardware and software together so that simulation to be fast and accurate.
- The existing simulation techniques have to be extended to combine simulation of hardware and software components.
- Different simulation platforms and techniques are used.
- Software runs fast while hardware simulation is relatively slow. So, the problem is how to run the system simulation as fast as possible and keep the two domains synchronized.
- Slow models provide full details and produce accurate results while fast models do not produce enough timing information and the simulation is not accurate.
- The verification of an embedded system can be also performed by prototyping (hardware-software prototyping platforms) before the final implementation. This issue is the subject of a next lecture.

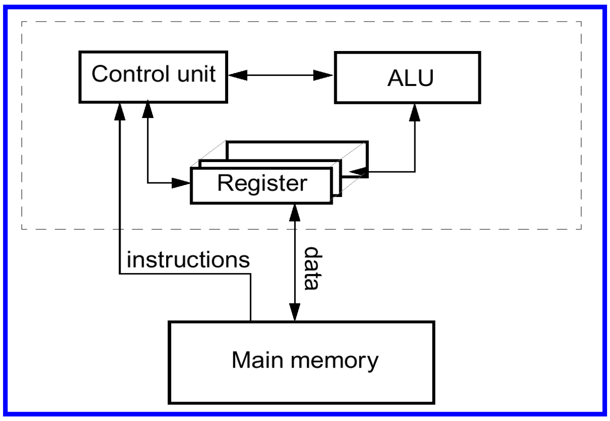
5a. Reduced instruction set computing (RISC) machines

University of Thessaly

Department of Computer and Communication Engineering

General purpose processor architecture

- The primary function of a CPU is to execute the instructions fetched from the main memory.
- An instruction tells to the CPU to perform one of its basic operations (arithmetic, logic, transfer of data from/to memory etc.).
- The control unit interprets (decodes) the instructions to be executed and tells to the other components what to do.
- The CPU includes a set of registers which are temporary storage devices for holding intensively used data and intermediate results.

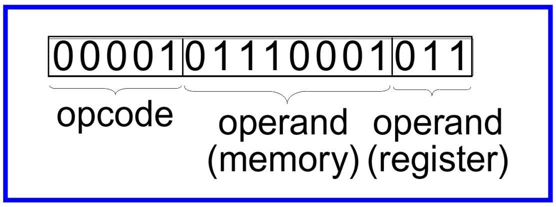


General purpose processor architecture (cont'd)

- The datapath of the processor consist of the circuitry for transforming data and for storing temporary data (ALU and registers).
- The ALU is capable to transform data through operations such as addition, subtraction, logical (AND, OR, invert), shifting, and to generate status signals (stored in the status register) indicating particular data conditions (e.g. when an addition generates a carry).
- Registers are capable to store temporary data that may include data from the memory not yet sent to the ALU, data coming from the ALU that will be needed for later ALU operations or will be sent back to memory, and data that have to be moved from one memory location to another.
- The control unit consists of circuitry for controlling the flow of data in the datapath according to the executed instructions. Contains the program counter that holds the memory address of the next instruction, and the instruction register to hold the fetched instruction’s address.
- Memory is used for storing program (instructions) and data and can be on-chip (faster access) or off-chip. To reduce the access time a local copy of a portion of memory may be kept in smaller, fast, on-chip and expensive cache memory.

Machine instructions

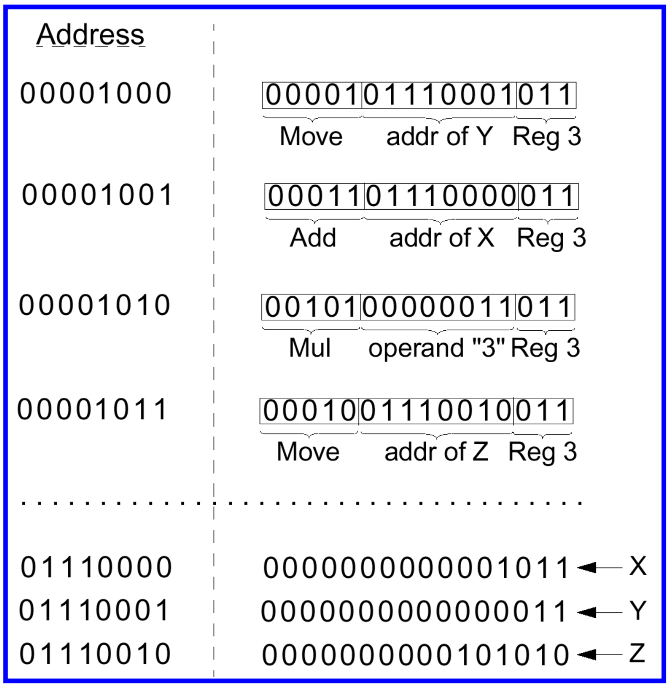
- A CPU can only execute machine instructions, and each CPU has a set of specific machine instructions (instruction set) which is able to recognise and execute.
- A machine instruction is represented as a sequence of bits which have to define: what has to be done (operation code), to whom the operation applies (source operands), where does the result go (destination operands), and how to continue after the end of the operation.
- The representation of a machine instruction is divided into fields, each field contains one item of the instruction specification (opcode, operands etc.), and the fields are organised according to the defined instruction format.



$\underbrace{\boxed{00001}}_{\text{opcode}}\underbrace{\boxed{01110001}}_{\text{operand (memory)}}\underbrace{\boxed{011}}_{\text{operand (register)}}$

Machine instructions (cont'd)

The following four instructions perform Z:=(Y+X)*3:



```
Address

00001000    00001|01110001|011
            Move   addr of Y  Reg 3

00001001    00011|01110000|011
            Add    addr of X  Reg 3

00001010    00101|00000011|011
            Mul   operand "3" Reg 3

00001011    00010|01110010|011
            Move   addr of Z  Reg 3

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

01110000    0000000000001011 ← X
01110001    0000000000000011 ← Y
01110010    0000000000101010 ← Z
```

Instruction set aspects

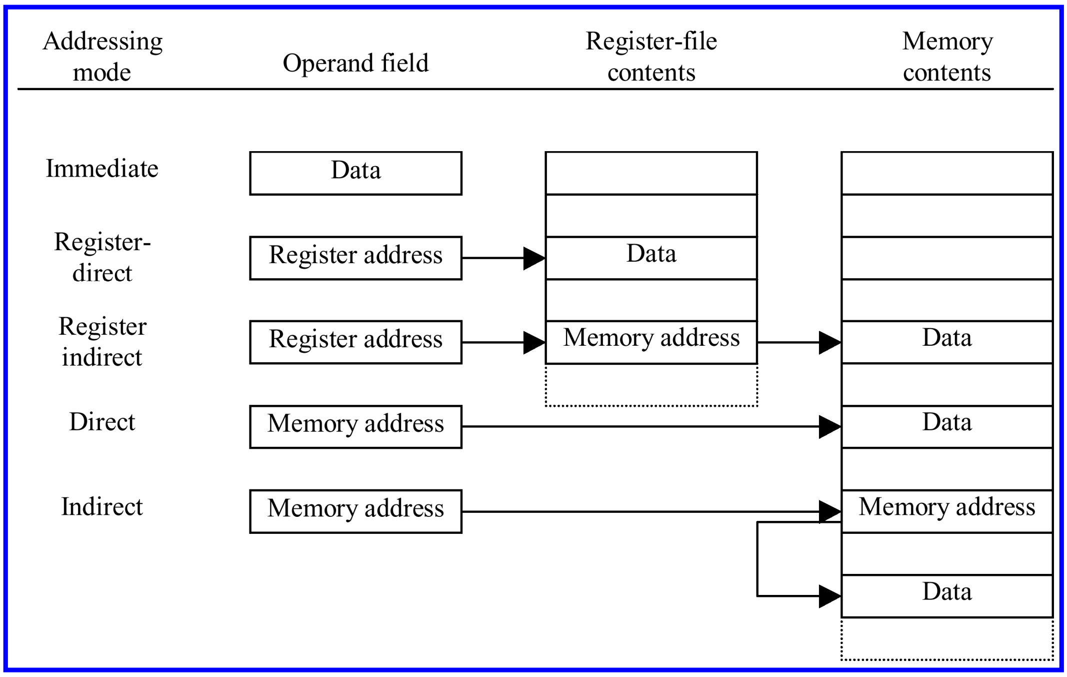
- Machine instructions are of four types:
  - ✓ Data transfer between memory and CPU registers.
  - ✓ Arithmetic and logic operations.
  - ✓ Branch instructions.
  - ✓ I/O transfer instructions.
- Important aspects in instruction set design:
  - ✓ Number of addresses.
  - ✓ Types of operands.
  - ✓ Addressing modes.
  - ✓ Operation types.
  - ✓ Instruction format.

Instruction set aspects (cont'd)

- Branch instructions determine the address of the next program instruction, based probably on datapath status signals.
- There are three categories of branch instructions:
  - ✓ Unconditional jumps determine the address of the next instruction.
  - ✓ Conditional jumps do the same thing only if a condition is true.
  - ✓ Call and return instructions: a call instruction, in addition to the indication of the address of the next instruction, saves the address of the current instruction so that a subsequent return instruction can jump back to the instruction immediately, following the most recent invoked call instruction.
- The instructions’ operand field specifies the location of the actual data that takes part in an operation. Source operands serve as input to the operation, while a destination operand stores the output.
- The number of operands per instruction varies among processors, but even in the same processor the number of operands per instruction may vary depending on the instruction type.

Instruction set aspects (cont'd)

- The data location in the operand field are indicated through several addressing modes. The main processors' addressing modes are:



Addressing mode | Operand field | Register-file contents | Memory contents

Immediate — Data

Register-direct — Register address → Data

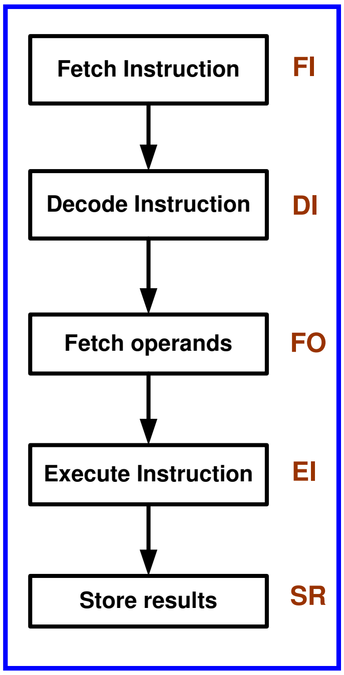
Register indirect — Register address → Memory address → Data

Direct — Memory address → Data

Indirect — Memory address → Memory address → Data

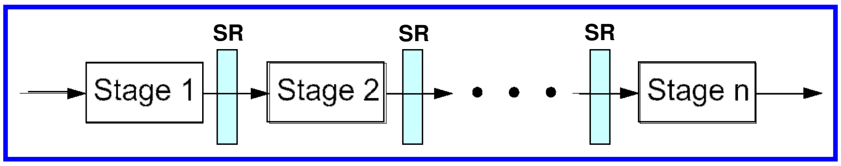
Typical instruction cycle

- Each instruction is performed as a sequence of steps, and the steps corresponding to one instruction are referred as instruction cycle:
  - ✓ **FI**: move the next instruction from the memory into the instruction register.
  - ✓ **DI**: determine what operation is represented by the fetched instruction.
  - ✓ **FO**: move the instruction's operand data from the memory into the appropriate registers (includes the calculation of the operand address).
  - ✓ **EI**: move of data from the appropriate registers to the ALU, execute the operation and feed the results to an appropriate register.
  - ✓ **SR**: write the content (result) of the register into the memory.



Instruction pipelining

- Pipelining is used to increase the instruction throughput of a processor.
- The instruction cycle contains several operations that are executed successively.
- This implies much hardware, but only a part of this hardware works at a given moment, dependent on the stage of the instruction cycle that is executed.
- With pipelining, the execution of multiple instructions is overlapped.
- Different parts of the hardware, work for different instructions at the same time (no additional hardware is required, except of stage registers).
- Machine cycle (or clock cycle) is the time required for moving an instruction from one stage of the pipeline to the next (longest time required for data to move from one stage register to the next).
- The execution of an instruction takes several machine cycles as it passes through the pipeline.

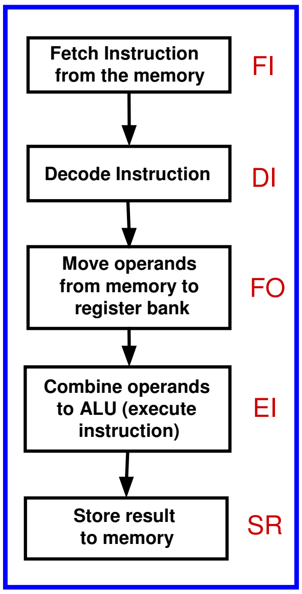


Instruction pipelining (cont'd)

Example 1: Five stage pipeline execution in general purpose processor

Clock cycle	1	2	3	4	5	6	7	8
	FI	DI	FO	EI	SR			
		FI	DI	FO	EI	SR		
			FI	DI	FO	EI	SR	
				FI	DI	FO	EI	SR

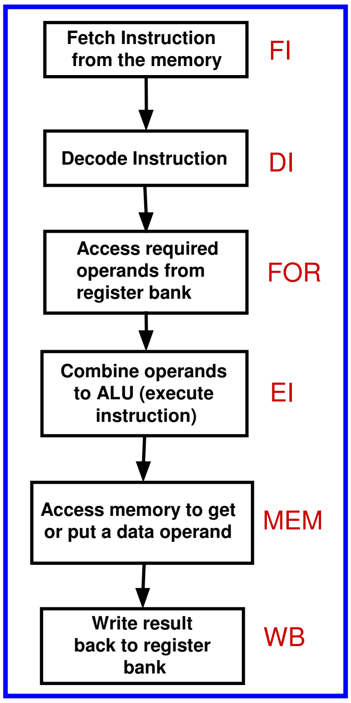
Latency: 5 cycles
Throughput: 1 cycle


Fetch Instruction from the memory
FI
Decode Instruction
DI
Move operands from memory to register bank
FO
Combine operands to ALU (execute instruction)
EI
Store result to memory
SR

Instruction pipelining (cont'd)

Example 2: Six-stage pipeline execution for load-store (RISC) architectures

Clock cycle	1	2	3	4	5	6	7	8
	FI	DI	FOR	EI	MEM	WB		
		FI	DI	FOR	EI	MEM	WB	
			FI	DI	FOR	EI	MEM	WB



Fetch Instruction from the memory — FI

Decode Instruction — DI

Access required operands from register bank — FOR

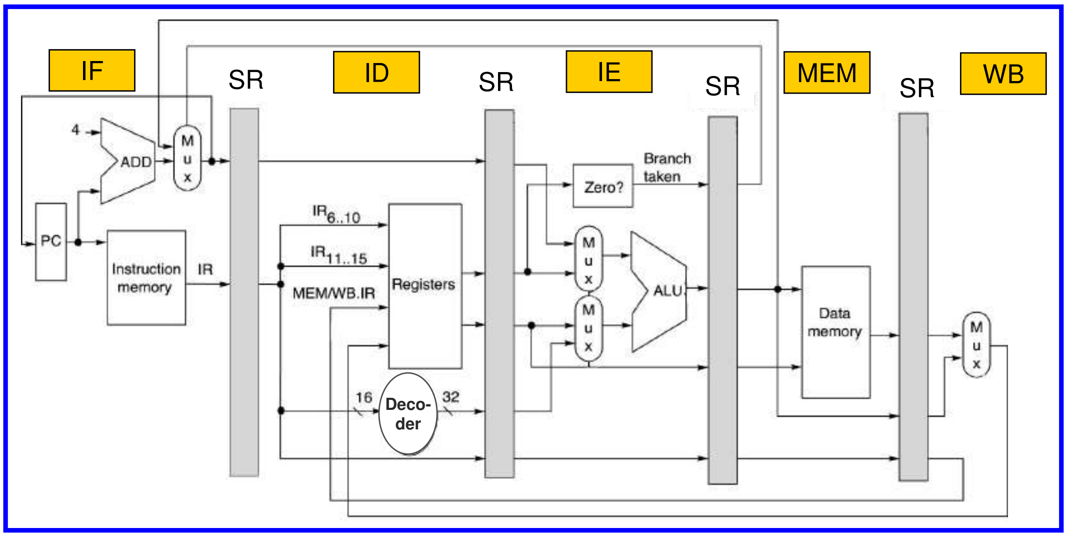
Combine operands to ALU (execute instruction) — EI

Access memory to get or put a data operand — MEM

Write result back to register bank — WB

Instruction pipelining (cont'd)

Example 3: Five stage pipeline execution (MIPS processor)



In the first stage, the instruction is fetched from memory (IF). It is then decoded and its operands are read from the registers (ID). Next, it is executed using the operands (EX) or in the case of load and store instructions, a memory address is calculated. In the next stage (MEM), load and store instructions access memory using the computed address, either retrieving a value from memory (load) or storing a value to memory (store). Registers are updated with the result of execution (WB).

Instruction pipelining (cont'd)

- Pipeline hazards are situations that prevent the next instruction in the instruction stream from executed during its designated cycle.
- In that case the instruction is stalled.
- All the instructions later in the pipeline are also stalled, and no new instructions are fetched. The instructions earlier than the stalled one can continue.
- There are three type of hazards:
  - ✓ Structural hazards.
  - ✓ Data hazards.
  - ✓ Control hazards.

Instruction pipelining (cont'd)

- Structural hazards occur when a certain resource (memory or functional unit) is requested by more than one instruction at the same time.
- In order to avoid them some resources are duplicated or pipelined in order to support several instructions at a time.
- A classical way to avoid hazards at memory access is by providing separate data and instruction caches.
- The first instruction fetches in the FO stage an operand from memory. The memory does not accept another access during that cycle.

Clock cycle	1	2	3	4	5	6	7	8
	FI	DI	FO	EI	SR			
		FI	DI	FO	EI	SR		
			stall	FI	DI	FO	EI	SR

Instruction pipelining (cont'd)

- Data hazards occur when an instruction needs the result produced by another instruction, but this result has not yet been generated.
- Instruction 1: MUL R2, R3 R2 → R2 x R3
- Instruction 2: ADD R1, R2 R1 → R1 + R2
- Before the execution of its FO stage, the ADD instruction is stalled until the MUL instruction has written the result into R2.
- Time penalty of data hazards can be reduced by feedback the ALU result to the ALU input (a multiplexer can be used). If the hardware detects that the value needed for the current operation is the one produced by the previous operation it selects the forwarded result as ALU input, instead of the value from register or memory.

Clock cycle	1	2	3	4	5	6	7	8
	FI	DI	FO	EI	SR			
		FI	DI	stall	stall	FO	EI	SR

Instruction pipelining (cont'd)

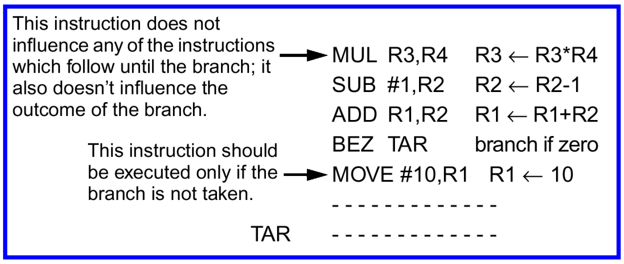
- Control hazards are produced by branch instructions.
- The instruction following the branch is fetched, because before the DI stage of the branch instruction is finished it is not known if a branch will be executed. Later the fetched instruction is discarded.
- After the FO stage of the branch instruction, the address of the target is known and it can be fetched.

Clock cycle	1	2	3	4	5	6	7	8	9
Branch	FI	DI	FO	EI	SR				
Target		~~FI~~	stall	FI	DI	FO	EI	SR	
					FI	DI	FO	EI	SR

Instruction pipelining (cont'd)

- Branch instructions can dramatically affect the pipeline performance.
- Such instructions are very frequent in current programs (20 - 30%), so it is very important to reduce the delay penalties produced by branches.
- The idea in order to cope with the branch delays is to let the CPU do some useful work during the delay cycles caused by the branch instruction.
- It is duty of the compiler to find an instruction which can be moved from its original place into the branch delay slot after the branch and which will be executed regardless of the outcome of the branch.

Initial program

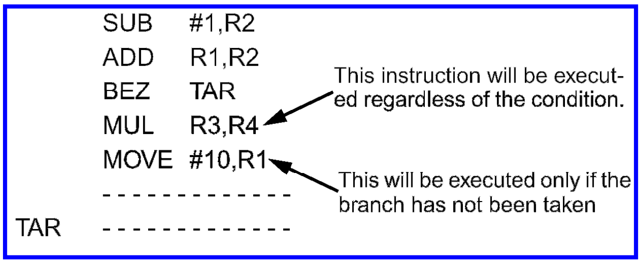


This instruction does not influence any of the instructions which follow until the branch; it also doesn't influence the outcome of the branch.

```
     MUL  R3,R4     R3 ← R3*R4
     SUB  #1,R2     R2 ← R2-1
     ADD  R1,R2     R1 ← R1+R2
     BEZ  TAR       branch if zero
     MOVE #10,R1    R1 ← 10
     - - - - - - - - - - - - -
TAR  - - - - - - - - - - - - -
```

This instruction should be executed only if the branch is not taken.

Program produced by compiler



```
     SUB    #1,R2
     ADD    R1,R2
     BEZ    TAR
     MUL    R3,R4
     MOVE   #10,R1
     - - - - - - - - - - - - -
TAR  - - - - - - - - - - - - -
```

This instruction will be executed regardless of the condition.

This will be executed only if the branch has not been taken

Instruction pipelining (cont'd)

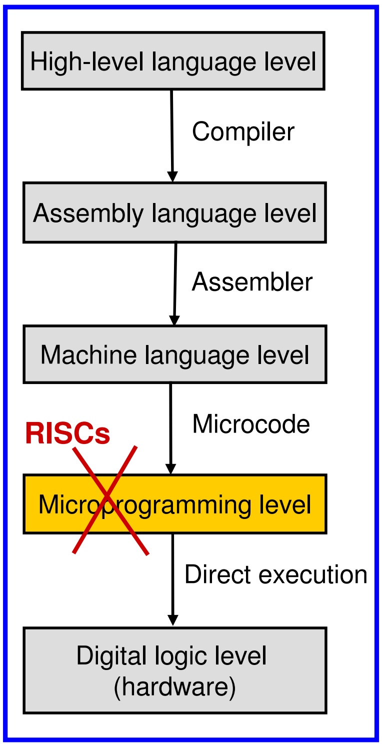
- If the compiler is not able to find an instruction to be moved after the branch (into the branch delay slot), a NOP instruction (an instruction that does nothing) has to be placed after the branch (the delay penalty will be the same).
- Statistics show that sophisticated compilers are able to find such an instruction for 60-80% of the branches.
- It is important to apply correct branch prediction to improve performance.
- Based on the predicted outcome, the respective instruction can be fetched, as well as the instruction following it.
- If, after the execution of the branch condition, it turns out that the prediction was correct, execution continues.
- On the other hand, if the prediction is not fulfilled, the fetched instructions must be discharged and the correct instruction must be fetched.
- There are dynamic prediction strategies (that take into account the history of conditional branches) and static prediction strategies (that do not take into consideration the execution history and use heuristics).

Processor architectural models

- Several instruction set architectural models have existed over the last three decades.
- First, CISC (complex instruction-set computers) with variable instruction formats, numerous memory addressing modes and large number of instruction types.
- The CISC philosophy was to create instruction sets leading in program formats close to those of high-level language programs, in order to simplify the compiler technology.
- RISC (reduced instruction-set computers) philosophy is based on uniform instruction lengths, limited addressing modes and reduced number of operation types.
- RISC concepts allow the design of machines to be more easily pipelined, improving the speed of the processor.
- VLIW (very long instruction word) architecture model allow multiple instructions handling per clock cycle. A fixed number of instructions are included in a single long instruction, and the compiler has the responsibility for the parallel execution of the instructions.

Processor architectural models (cont'd)

- The increased use of high-level languages (HLLs) is one of the reasons leaded to the development of complex processor architectures.
- Due to the fact that these languages contain complex control structures such as *if*, *while* and *case,* a significant semantic gap has been created (between HLLs and assembly), making the construction of modern compilers quite difficult.
- A popular approach to solve the problem was the inclusion of new (complex) instructions at the assembly/machine language levels in order to handle high-level instructions (such as multiple branches, i.e. case), and of special addressing modes for procedure calls, and writing/reading values in the memory.
- These complex instructions and addressing modes make use of the microcode before their execution by the processor's hardware (CISC machines). This may lead to the reduction of the processor speed.
- On the other side, RISC machines simplify the instruction set, and avoid the microprogramming level by executing their simple instructions directly to the hardware. However, this requires more complex and efficient compilers.



High-level language level

Compiler

Assembly language level

Assembler

Machine language level

Microcode

RISCs

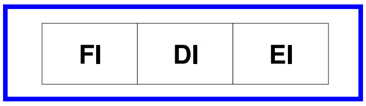
Microprogramming level

Direct execution

Digital logic level (hardware)

Main characteristics of RISC architectures

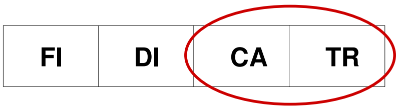
- The instruction set is limited and includes only simple instructions. The goal is to create an instruction set containing instructions that execute quickly.
- Most of the RISC instructions (for data processing) are executed in a single machine cycle (after fetching and decoding). There is pipelined operation without memory reference.



- The simplicity of the RISC instructions make them hardwired, while CISC architectures have to use microprogramming to implement complex instructions.
- Having simple instructions results in reduced complexity of the control unit and the data path, and as a sequence the processor can work in high clock frequency (small cycle time or clock period).
- The pipeline processing is used efficiently due to the fact that the instructions are simple and have similar execution times.

Main characteristics of RISC architectures (cont'd)

- RISC machines are based on a load-and-store architecture: only LOAD and STORE exchange data with the memory, and all other instructions operate only with registers (register-to-register instructions).
- Thus, only the few instructions accessing the memory need more than one cycles (two cycles after fetching and decoding):

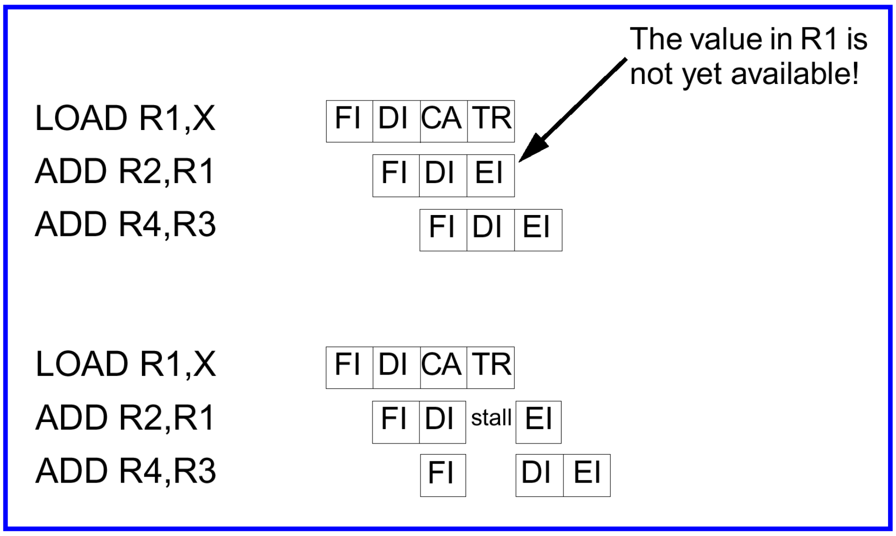


CA: calculate operand address
TR: transfer operand from or to memory

- These instructions (LOAD and STORE) use only few addressing modes, which are usually: direct, indirect, register indirect.

Main characteristics of RISC architectures (cont'd)

- The delay-load problem: LOAD instructions (similar to the STORE instructions) require memory access and their execution cannot be completed in a single clock cycle. However, in the next cycle a new instruction is started by the processor.



The value in R1 is not yet available!

LOAD R1,X — FI | DI | CA | TR

ADD R2,R1 — FI | DI | EI

ADD R4,R3 — FI | DI | EI

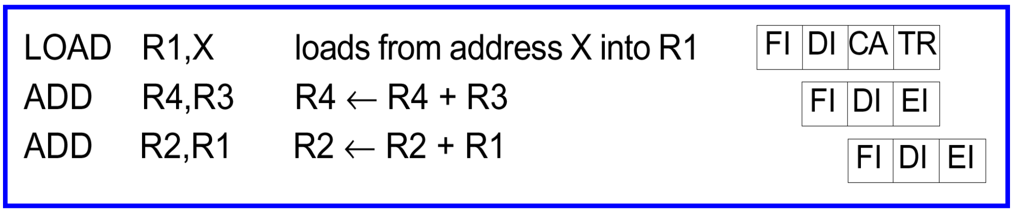
LOAD R1,X — FI | DI | CA | TR

ADD R2,R1 — FI | DI | stall | EI

ADD R4,R3 — FI | DI | EI

Main characteristics of RISC architectures (cont'd)

- It is responsibility of the compiler to find an instruction for execution after the LOAD instruction that does not need the loaded value, in order to avoid stalling.

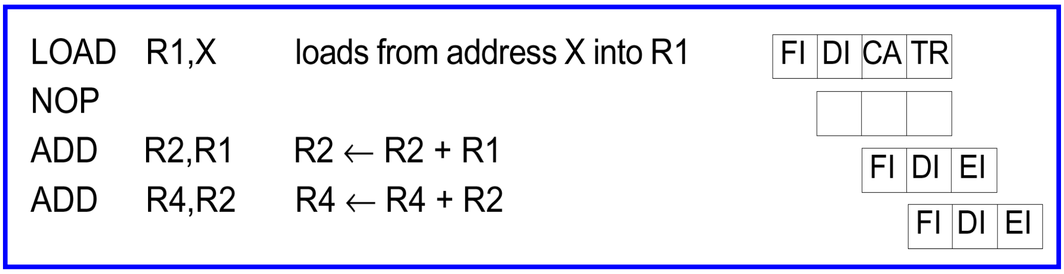


LOAD R1,X — loads from address X into R1 — FI | DI | CA | TR

ADD R4,R3 — R4 ← R4 + R3 — FI | DI | EI

ADD R2,R1 — R2 ← R2 + R1 — FI | DI | EI

- For the following sequence the compiler has generated a NOP instruction after the LOAD, because there is no instruction to fill the load-delay slot.



LOAD R1,X — loads from address X into R1 — FI | DI | CA | TR

NOP

ADD R2,R1 — R2 ← R2 + R1 — FI | DI | EI

ADD R4,R2 — R4 ← R4 + R2 — FI | DI | EI

Main characteristics of RISC architectures (cont'd)

- RISC instructions are of fixed length and uniform format.
- This makes the loading and decoding of instructions simple and fast, and it is not needed to wait until the length of the instruction is known in order to start decoding the following one.
- Decoding is simplified because opcode and address fields are located in the same position for all instructions.
- Example - Basic instruction format of RISC I:

8	5	5	1	13
OPCODE	DEST	SOURCE	A	OFFSET

- ADD (common instruction):
  - ✓ If A=0 (register direct mode) the first operand is taken from the SOURCE register, the second is taken from a register defined by the 5 LSBs of the OFFSET field, and the result is stored in the DEST register.
  - ✓ If A=1 (immediate mode) the second operand is a constant of 13 bit (OFFSET filed).

Main characteristics of RISC architectures (cont'd)

- RISC machines contains a large number of registers.
- Variables and intermediate results can be stored in registers and do not require repeated loads and stores from/to memory (reduce number of LOAD and STORE instructions). The number of registers may be up to 500 !
- Traditionally, when a new procedure is called, the registers have to be saved in memory since they contain values of variables and parameters of the calling procedure. After the returning to the calling procedure the values have to be loaded from the memory.
- To avoid the above time consuming process, the large number of registers in RISC machines gives the opportunity to allocate a new set of registers to the called procedure and the resister set assigned to the calling one remains untouched.
- Compilers in RISC machines handles the optimization of the registers' use, which makes the RISC compilers more complex that those of CISC ones.
- RISC compilers use several techniques in order to handle the large number of registers and to increase the level of their usability. A common method in several RISC compilers is the use of overlapping register windows or banks.

Main characteristics of RISC architectures (cont'd)

- RISC processors have small die size: they are simple processors and thus they require fewer transistors and less silicon area.
- As a result, a RISC CPU leaves more free area for performance-enhancing features (cache memory, DSP and memory management functions).
- RISC processors require small development time: they are simple processors and thus they require less design effort and therefore have low design cost.
- RISC machines have generally poor code density as a consequence of the fixed instruction set.
- In the absence of cache memory, poor code density leads to more main memory bandwidth being used for instruction fetching.
- Where code density is of prime importance, some RISC processors use a compressed instruction set (encoding with fewer bits).
- For example, the ARM processors have incorporated a novel mechanism called Thumb architecture which uses a 16-bit compressed form of the original 32-bit instruction set and employs decompression in the instruction pipeline.

RISC versus CISC

RISC machines	CISC machines
Most instructions require one cycle	Complex instructions require multiple cycles
Only LOAD/STORE instructions refer to memory	Any instruction can refer to memory
High-level of pipelining (due to simple instructions)	Low-level of pipelining (due to complex instructions)
Instructions are executed directly to hardware	Instructions are interpreted by microcode
Instructions of fixed length and uniform format	Instructions of variable length and format
Poor code density (due to simple instructions)	High code density (due to complex instructions)
Simple control unit and datapath (lead to smaller size & development time)	Complex control unit and datapath (lead to large size & development time)
High clock frequency (due to pipelining & simplicity)	Low clock frequency (due to complexity)
Reduced number of instructions and addressing modes	Large number of instructions and addressing modes
Complexity in compilers	Complexity in microprogramming
Large number of registers	Small number of registers

RISC versus CISC (cont'd)

CISC Architecture:

Pentium

Number of instructions: 235
Instruction size: 1 - 11 bytes
Instruction format: not fixed
Addressing modes: 11
Number of general purpose registers: 8

RISC Architecture:

Sun SPARC

Number of instructions: 52
Instruction size: 4 bytes
Instruction format: fixed
Addressing modes: 2
Number of general purpose registers: up to 520

RISC versus CISC (cont'd)

- Equation for expressing the processors' performance ability:

$$\boxed{\frac{\text{time}}{\text{program}} = \frac{\text{time}}{\text{cycle}} \text{ x } \frac{\text{cycles}}{\text{instruction}} \text{ x } \frac{\text{instructions}}{\text{program}}}$$

- The CISC approach attempts to minimize the number of instructions per program, but suffers in number of cycles per instruction.
- RISC approach does the opposite: reduces the cycles per instruction at the cost of number of instructions per program.
- Also, the RISC approach exhibits reduced cycle time due to the reduced complexity of the control unit and the datapath.
- Most of the current processors are not typical RISC or CISC machines, but try to combine advantages of both approaches.
- In embedded systems, we choose a RISC machine mainly due to its reduced complexity and its suitability in terms of instruction set characteristics for the given application implemented by the given embedded system.

Principles in RISC machines design

- Analysis of the application to be implemented in order to find the key operations (use statistics from programs implementing the target application or similar applications).
- Design of an optimal datapath (ALU and registers) for the execution of the key operations. The data path have to be optimized for the target application, and the used programming language / compiler.
- The time needed for the fetching of the operands from the registers their passing through the ALU and the storing of the result back to a register (data path cycle time) must be optimized.
- Design of a suitable instruction set for the optimal implementation of the key operations.
- Inclusion of extra instructions (with less usability) that do not slow down the machine.
- Selection of few suitable addressing modes by taking into account how they affect the speed of the processor.

Conclusions

- Processors (CISC or RISC) try to solve the same problem: to cover the gap between HLLs and assembly (semantic gap).
- They do it in different ways: CISC machines are going the traditional way of implementing complex instructions, while RISC machines try to simplify the instruction set.
- Innovations in RISC architectures are based on an analysis of a large set of widely used programs.
- The main features of RISC architectures are: reduced number of simple instructions, small number of addressing modes, load-store architecture, instructions of fixed length and format, availability of many registers.
- RISC architectures use pipelining efficiently.
- The simplification of the instruction set as well as the handling of the large number of registers and the optimization of their use, lead to a need for complex compilers in RISC machines.
- Modern architectures often include both RISC and CISC features.

5b. The ARM processor

University of Thessaly

Department of Computer and Communication Engineering

History

- 1985: Acorn developed the first commercial RISC processor.
- 1990: The cooperation of Acorn and Apple leads to the creation of Advanced RISC Machines (ARM) in UK.
- 1991: Development of the first embeddable RISC processor (ARM6).
- 1992-1994: Vendors as Sharp and Shamsung asked license to use the ARM processor for their embedded applications.
- 1993: Development of ARM7 (suitable for multimedia applications).
- 1995: Development of the Thumb architecture and the ARM8.
- 1996-2000: Alcatel, Philips, Sony and Erickson (for Bluetooth) asked license to use ARM processors in their applications.
- 2001-2006: The percentage of ARM in the market of 32-bit embedded RISC processors approached 80%.

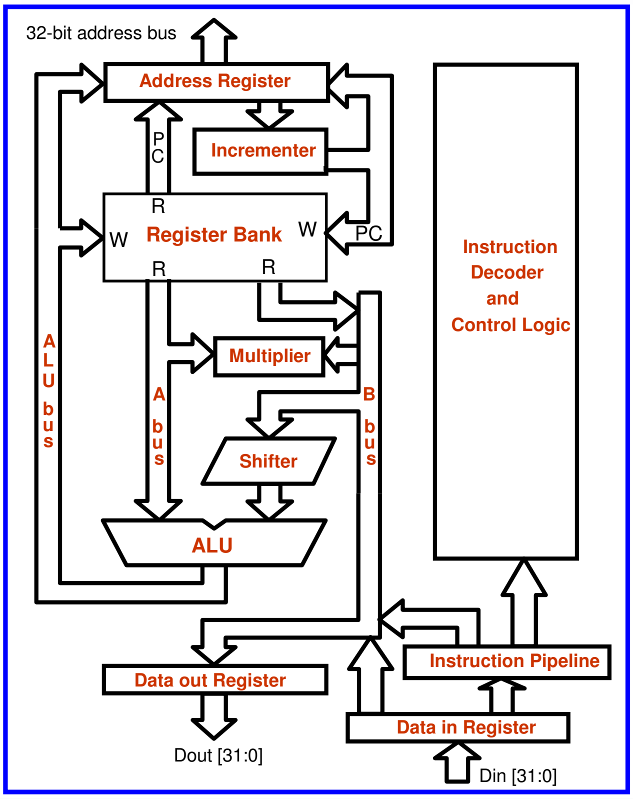
Introduction

- ARM is one of the most commonly used RISC processors (mainly as embedded core, but also as discrete component):
  - ✓ Hard IP block: fully characterized on a target technology and exploits the area, power and performance advantages of full-custom layout design.
  - ✓ Soft IP block: synthesized netlists on RTL level.
- Uses fixed-length 32-bit instructions.
- Contains a large number of 32-bit general purpose registers.
- It is based on a load-store architecture: instructions that reference to memory just move data without doing processing.
- The processing instructions use values stored in registers only, and exhibit single cycle execution directly by the hardware (without the need of microcode).
- Has simple and pipelined architecture that leads in small implementations (in terms of area), and low energy consumption.
- It is used for relatively “small” applications that require quite high performance.

ARM7 family

- The basic processor of the ARM7 family is the ARM7TDMI.
- ARM7TDMI is the most widely used 32-bit embedded RISC processor. Optimized for cost and low-power applications, this solution provides the low power consumption, small size, and high performance needed in portable, embedded applications.
- The family also include processor cores with unified cache memory, as well as synthesizable (embedded) cores in Verilog and VHDL, ready for compilation into commercial technologies. Optimized for flexibility and featuring an identical feature set to the hard macrocell, it improves time-to-market by reducing development time.
- In the next slides the basic processor of the ARM7 family is analyzed, while at the end of the lecture some details are provided concerning the evolution of the ARM architecture in most recent families such as ARM9, ARM9E, ARM10 and ARM11.

ARM7 architecture



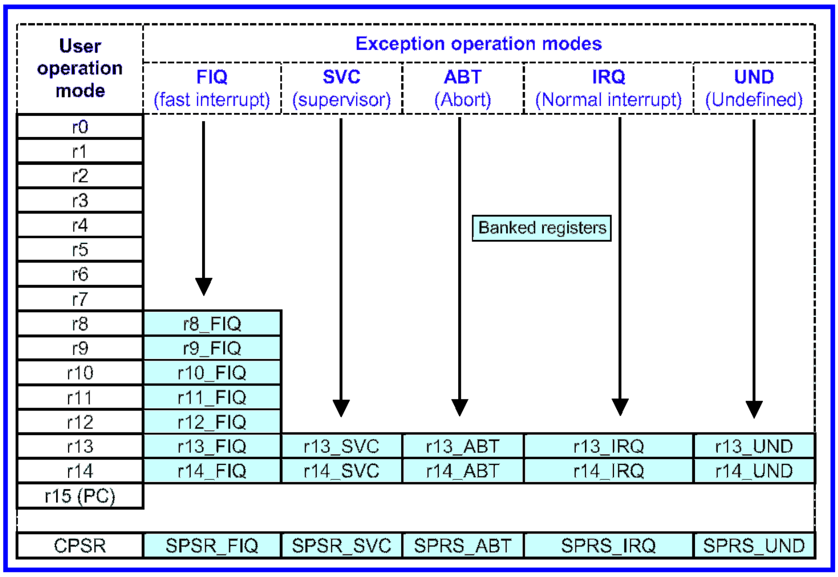
Main components

- Register bank with two read ports and one write port used to access any register, plus single read and write ports that give special access to r15 (PC). The extra write port allows the update of PC as the instruction fetch address is incremented, and the read port allows to an instruction to continue after a data address has been computed.
- Shifter can shift an operand by any number of bits.
- ALU and multiplier perform logic and arithmetic functions required by the instruction set.
- Address register and incrementer select and hold the memory addresses and generate sequential addresses when required.
- Data registers and instruction pipeline hold data passing to and from memory, and instructions coming from memory.
- Instruction decoder and associated control logic managing the datapath operation.

ARM register bank

- The processor's instruction set defines the operations that the programmer can use to change the state of the processor.
- This state comprises the values stored to the processor's visible registers and to the memory.
- Each instruction performs a defined transformation from the processor state before the instruction is executed to the state after it has completed.
- Although the processor have many invisible registers involved in executing an instruction, the values of these registers before and after the execution of the instruction, are not significant.
- Only the values in the visible registers are significant.
- When writing user-level programs, only 15 general-purpose 32-bit registers (r0-r14), the program counter (r15) and the current program status register (CPSR) need to be considered.
- The remaining registers are used only for handling exceptions (e.g. interrupts).

ARM register bank (cont'd)



User operation mode	Exception operation modes				
	FIQ (fast interrupt)	SVC (supervisor)	ABT (Abort)	IRQ (Normal interrupt)	UND (Undefined)
r0					
r1					
r2					
r3					
r4					
r5					
r6					
r7					
r8	r8_FIQ				
r9	r9_FIQ				
r10	r10_FIQ				
r11	r11_FIQ				
r12	r12_FIQ				
r13	r13_FIQ	r13_SVC	r13_ABT	r13_IRQ	r13_UND
r14	r14_FIQ	r14_SVC	r14_ABT	r14_IRQ	r14_UND
r15 (PC)					
CPSR	SPSR_FIQ	SPSR_SVC	SPRS_ABT	SPRS_IRQ	SPRS_UND

- ARM has 37 32-bit registers: 31 general-purpose, including a program counter (PC) and 6 status registers.
- In all operation modes 15 general-purpose registers (r0 to r14), one status register and the program counter are visible.
- Registers r0 to r7 are unbanked registers, i.e. each of them refers to the same physical register in all processor modes.
- Registers r8 to r14 are banked registers. Each of them refers to a physical register according to the current processor mode. Specific names are used for the particular physical registers of each operational mode.

ARM register bank (cont'd)

- r15 (program counter) holds the address of the next instruction.
- r13 is used as a stack pointer (SP). Each exception mode has its own banked version of r13, which indicates a stack dedicated to that exception mode and stores to this stack the values of the user registers (in order not to corrupt the state of the user program that was being executed when the exception occurred).
- r14 is used as link register (LR) and has two special functions:
  - ✓ When an exception occurs, the exception mode's version of r14 is set to the exception return address. The return from the exception mode is performed by copying back r14 to the PC.
  - ✓ In each mode, the mode's own version of r14 is used to hold subroutine return addresses. When a subroutine is called, r14 is set to the subroutine return address, and the return is performed in a similar way with that of the exception mode.
- The CPRS (current program status register) is used in user-level programs to store the condition code bits (used for record a result of a comparison operation, and to control whether or not a conditional branch is taken).
- The SPSRs (saved program status registers) are used to save the state of the CPSR of the task before the exception occur.

ARM exceptions

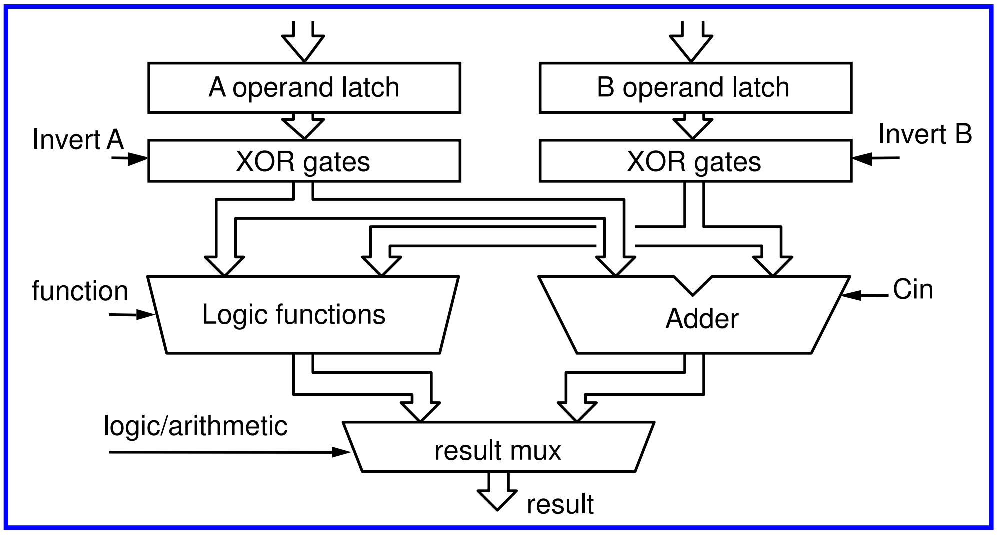
- Exceptions are used to handle unexpected events arising during the program execution, such as interrupts or memory faults.
- ARM exceptions are considered in three groups:
  - ✓ Exceptions generated directly from instruction execution: software interrupts, undefined instructions (absent coprocessor instruction), prefetched aborts (memory faults during fetch).
  - ✓ Exceptions generated as side effects of an instruction: data aborts (memory fault during load-store instructions).
  - ✓ Exceptions generated externally: reset, IRQ (interrupt), and FIQ (fast interrupt).

Exception modes priorities
Reset (highest priority)
Data Abort (data access memory abort)
FIQ (fast interrupt - interrupt with the highest priority)
IRQ (normal interrupt)
Prefetch Abort (instruction fetch memory abort)
Software interrupt, Undefined instruction

ARM exceptions (cont'd)

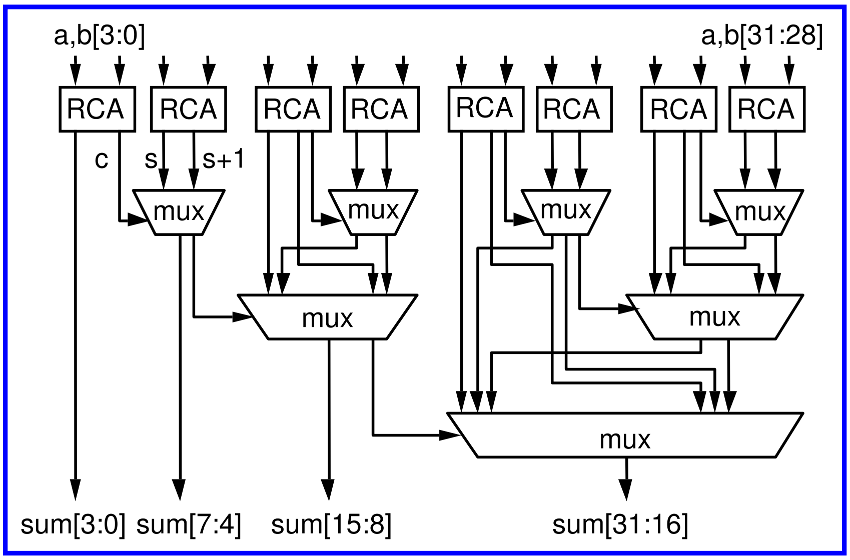
- When an exception arises, ARM completes the current instruction (not in reset mode in which the current instruction is stopped) and then departs from the current instruction sequence to handle the exception:
  - ✓ The current state is saved by copying the PC into r14 of the new mode and the CPSR into SPSR of the new mode.
  - ✓ The processor operating mode is changed to the appropriate exception mode.
  - ✓ The PC is forced to a specific value depending on the mode of operation.
  - ✓ The two low physical registers (r14, r13) in each exception mode are used to hold the return address (PC) and to save to the memory other user registers (stack pointer). FIQ mode has additional registers to give better performance by avoiding to save to the memory user registers.
- The return to the user program is achieved by restoring the user state exactly as it was when the exception occurred:
  - – Any modified user registers are restored from the stack.
  - – The CPSR is restored from the appropriate SPSR.
  - – The PC must be changed back to the relevant instruction address in the user instruction stream.

ARM ALU organization

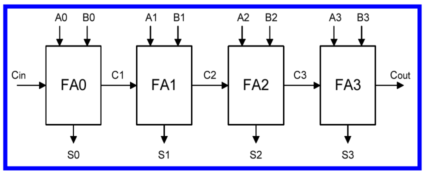


- The input operands are selectively inverted, and then added or combined in the logic and arithmetic units.
- Two parallel units: one for logic and one for add operations.
- A multiplexer selects the proper output.
- ALU uses a carry-select adder in order to optimize its speed.

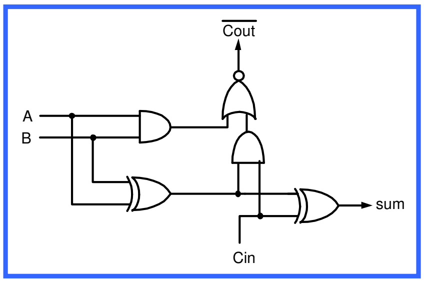
ALU carry-select adder



RCA

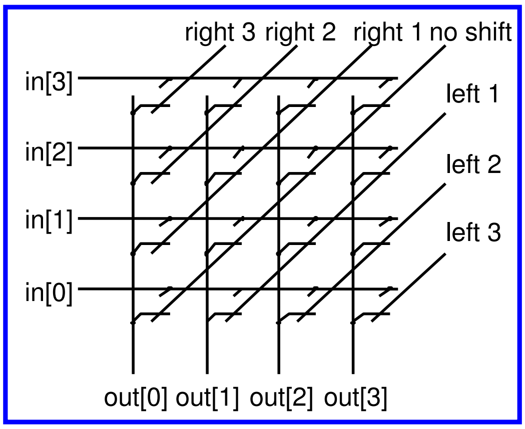


FA



- This adder uses double 4-bit ripple-carry adders and computes in the same time the sum of each addition for a carry-in of both zero and one.
- Then the final result is selected by using the correct carry-in to control a multiplexer.
- The scheme is fast, but consumes extra energy due to the double additions.

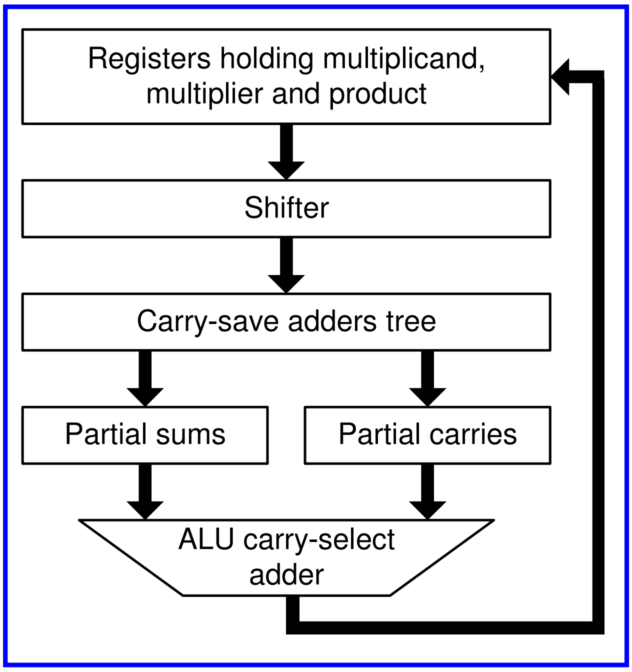
ARM shifter



4x4 switch matrix
(ARM uses a 32x32 matrix)

- The ARM architecture supports instructions which perform a shift operation in series with an ALU operation.
- The shifter performance is critical (contributes directly to the datapath cycle time).
- Some processor architectures tend to have the shifter in parallel with ALU, in order not to affect the datapath cycle time.
- In order to minimize the delay through the shifter, a cross-bar switch matrix is used.
- Each input is connected to each output through a switch matrix (containing switches implemented with NMOS transistors).
- For a left or right shift, one diagonal is turned on.

ARM multiplier

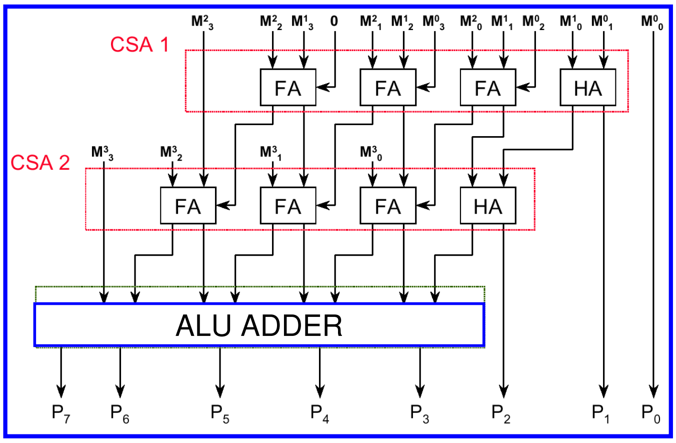


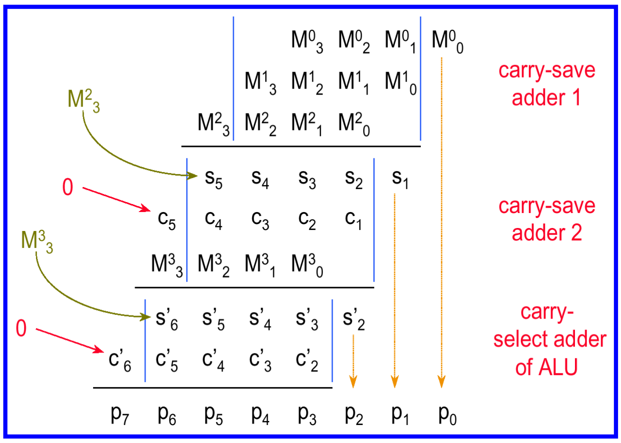
- The high-performance multiplication used in recent ARM cores tries to avoid carry propagate delays associated with adding partial products together.
- Thus, a carry-save adder (CSA) tree is used, which includes 4 layers (stages) with 8 full-adders in each of them. The produced partial sums and carries are combined in the ALU's adder to compute the final product.
- A shifter is used in order to perform the required shifting for the application of the CSA-based multiplication algorithm that has been selected for the multiplier of the ARM processors.
- A multiply instruction can take as few as two cycles (one to load the instruction and a second for the first CSA stage) or as many as five cycles (one to load the instruction and four for the CSA stages).

ARM multiplier (cont'd)

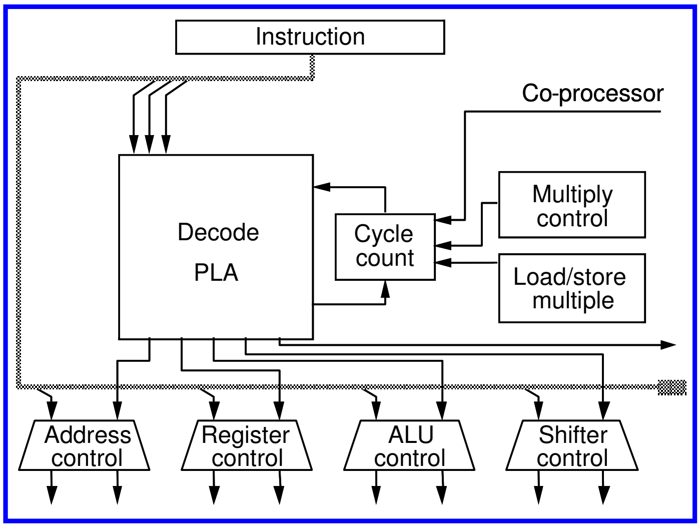
$$\begin{array}{cccccccc}
 & & & & M_3 & M_2 & M_1 & M_0 \\
 & & \times & & m_3 & m_2 & m_1 & m_0 \\
\hline
 & & & & M^0_3 & M^0_2 & M^0_1 & M^0_0 \\
 & & & M^1_3 & M^1_2 & M^1_1 & M^1_0 & \\
 & & M^2_3 & M^2_2 & M^2_1 & M^2_0 & & \\
 & M^3_3 & M^3_2 & M^3_1 & M^3_0 & & & \\
\hline
p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}$$

- Carry-save tree is suitable where the result of a carry-save addition is immediately re-used in another addition.
- A full adder is used for adding the carry-out, the partial-sum and the new data at each stage.
- A final addition stage is needed to combine the partial-sums and the carry-outs into the product.
- No carry is propagated within the stages.





ARM control structures



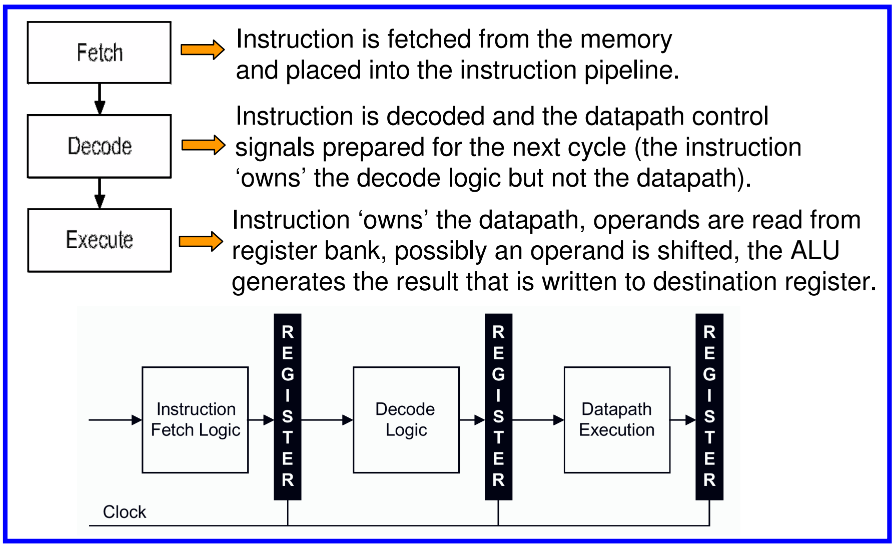
- Instruction decode PLA (programmable logic array): uses some instruction bits and the cycle counter to define what operation will be performed in the next cycle.
- Distributed control logic associated with the datapath blocks (address control, register control, ALU control, shifter control). Uses information from the PLA to select other instruction bits in order to control the datapath.
- Decentralized extra control units for specific instructions that takes variable number of cycles to complete (multiple words load and store, multiply, co-processor operations).

ARM co-processor interface

- ARM architecture has a mechanism for extending its instruction set through the addition of co-processors.
- A co-processor can be a floating-point machine or a machine dedicated to other application-specific operations.
- The co-processor is attached to the bus where the instruction stream flows into the ARM, and copies the instructions into an internal pipeline that mimics the behavior of the ARM instruction pipeline.
- The ARM has the responsibility to control the instruction flow, and each co-processor has its own register bank to perform its operations.
- Co-processor's data processing instructions are completely internal and cause state changes in the co-processor's registers.
- In case of data transfer instructions of the co-processor, the ARM processor generates the memory address, but the co-processor controls the flow of data words.

Operation for data-processing instructions

- ARM is based on a simple pipeline architecture to increase the speed of the flow of instructions to the processor. This enables several operations to take place simultaneously.
- A three-stage pipeline is used, so data processing instructions are executed in a single cycle (after fetching and decoding).
- While one instruction is being executed, its successor is being decoded, and a third instruction is being fetched from memory (latency: 3 cycles, throughput: 1 cycle).

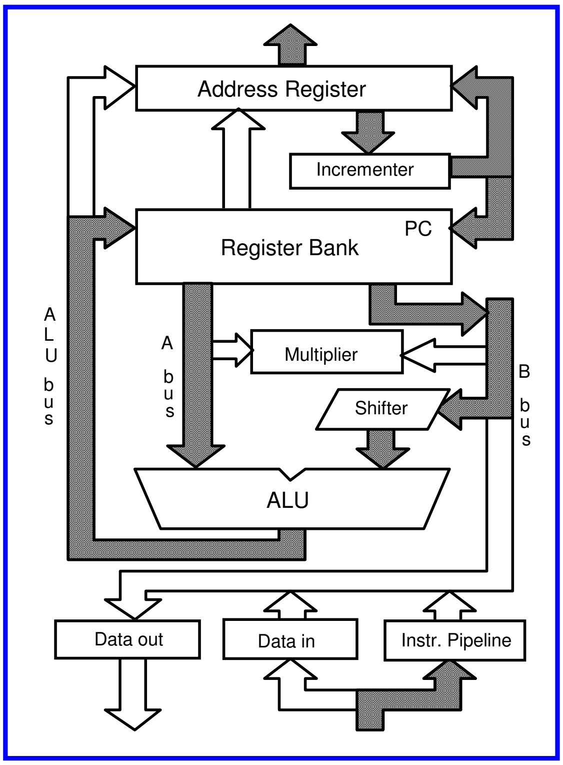


Fetch: Instruction is fetched from the memory and placed into the instruction pipeline.

Decode: Instruction is decoded and the datapath control signals prepared for the next cycle (the instruction 'owns' the decode logic but not the datapath).

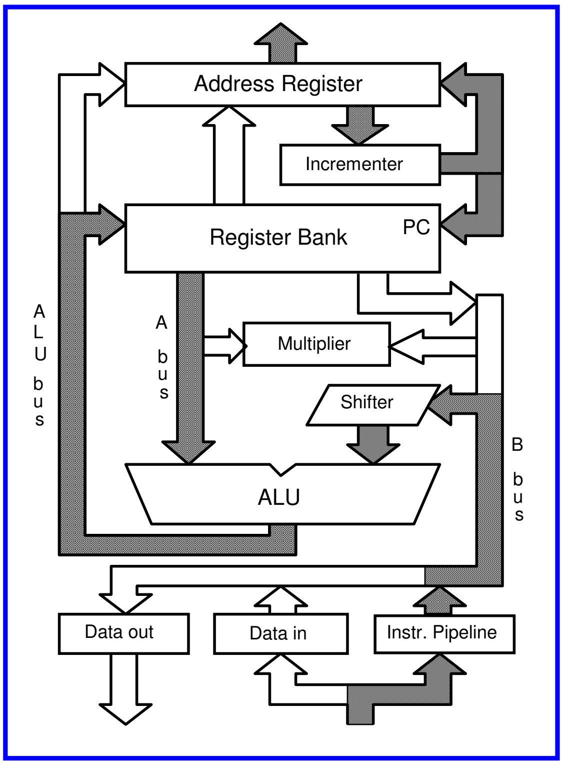
Execute: Instruction 'owns' the datapath, operands are read from register bank, possibly an operand is shifted, the ALU generates the result that is written to destination register.

Operation for data-processing instructions (cont'd)



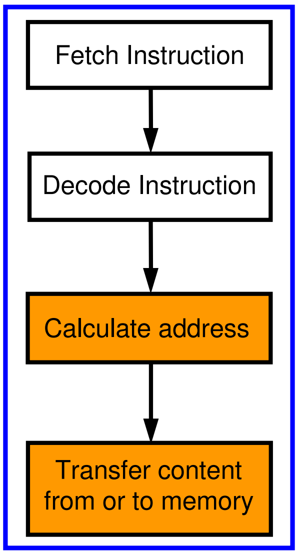
- In register-register data processing instructions (single cycle), two register operands are accessed.
- The value on the B bus can be shifted and is combined with the value on the A bus in the ALU.
- Result is written back into the register bank through the ALU bus.
- PC value is in the address register, from where it is fed into the incrementer and then copied back into r15 and into the address register to be used as the address for the next fetched instruction (loaded into the bottom of the instruction pipeline).

Operation for data-processing instructions (cont'd)



- In register-immediate data processing instructions (single cycle), only one register operand is accessed.
- The immediate value is extracted on the B bus from the current instruction located at the top of the instruction pipeline.
- This value is combined with the value on the A bus in the ALU.
- Result is written back into the register bank.
- PC value is in the address register, from where it is fed into the incrementer and then copied back into r15 and into the address register to be used as the address for the next fetched instruction (loaded into the bottom of the instruction pipeline).

Operation for data-transfer instructions



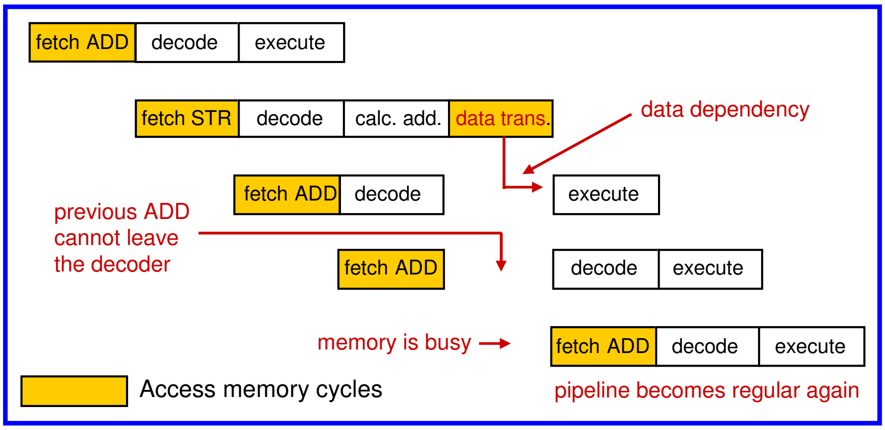
Fetch Instruction

Decode Instruction

Calculate address

Transfer content from or to memory

Example: Sequence of single-cycle ADD instructions with a STORE instruction occurred after the first ADD.



fetch ADD | decode | execute

fetch STR | decode | calc. add. | data trans.

data dependency

fetch ADD | decode | execute

previous ADD cannot leave the decoder

fetch ADD | decode | execute

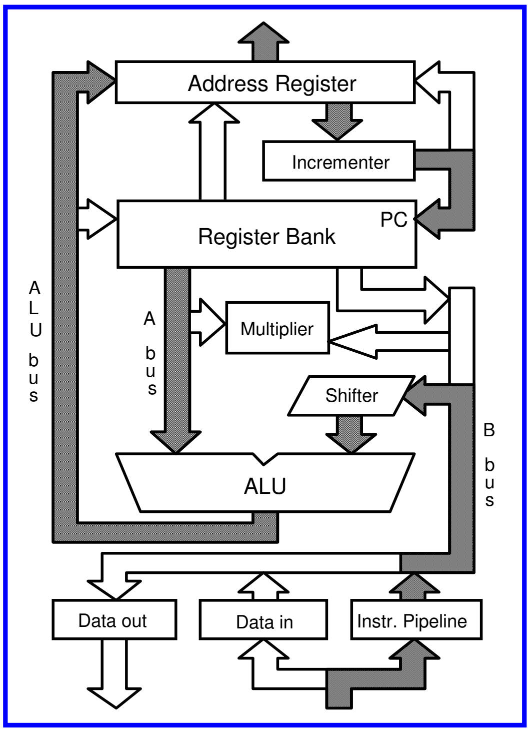
memory is busy → fetch ADD | decode | execute

pipeline becomes regular again

Access memory cycles

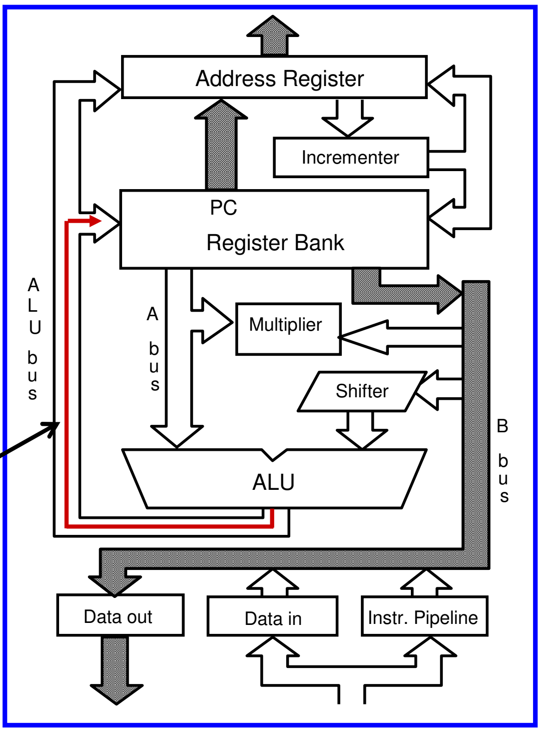
- STORE instruction need two cycles for execution (calculate of address and data transfer to memory). LOAD instruction needs an additional third cycle to transfer the data from the data-in register (in which data are placed from the memory) to the destination register.
- When a multi-cycle instruction is executed between single-cycle operations the flow is less regular, due to the presence of stalls (breaks in the pipeline). We can see in the example that the memory is used in every cycle.

Operation for data-transfer instructions (cont'd)



- STORE instruction needs two cycles.
- First cycle: Address calculation
- A register is used as the base register, to which is added an offset which may be another register or an immediate value.
- The calculated address is sent to the address register through the ALU bus, in order in the second cycle the data transfer to takes place.
- The incremented PC value is stored in the register bank at the end of the first cycle, so that the address register is free to accept the data-transfer address for the second cycle.

Operation for data-transfer instructions (cont'd)



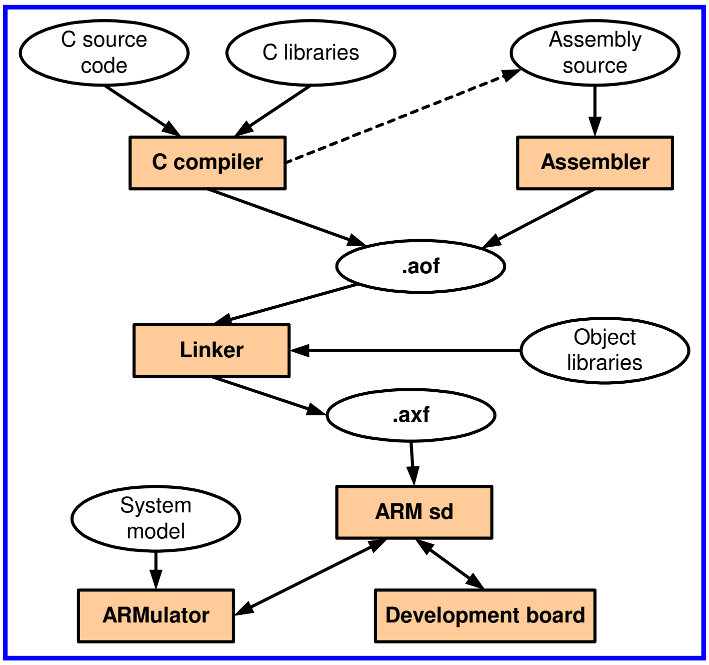
Only for auto-indexing

- Second cycle: Data transfer
- The content of the register to be stored is sent to the Data-out register through the B bus, and then directly to the calculated address of the memory.
- The value of the PC is fed back to the address register to allow to the prefetched instruction to continue.
- LOAD instruction follow a similar way of execution, except that the data from memory are placed in the data-in register on the second cycle, and a third cycle is needed to transfer the data from there to the destination register (through the B and ALU buses).

Operation for branch instructions

- Branch instructions need three cycles.
- Branch instructions compute the target address of the branch in the first cycle (in the same way as the calculation of the address in data-transfer instructions).
- The result is issued as an instruction fetch address, and while the instruction pipeline refills, the return address is copied into the link register (r14) during the second cycle.
- The third cycle is required to complete the pipeline refilling, and to make a small correction to the value stored in the link register in order to be suitable for the instruction which follows the branch.
- Other ARM instructions operate in a similar manner with the three instruction categories described previously.

ARM development tools



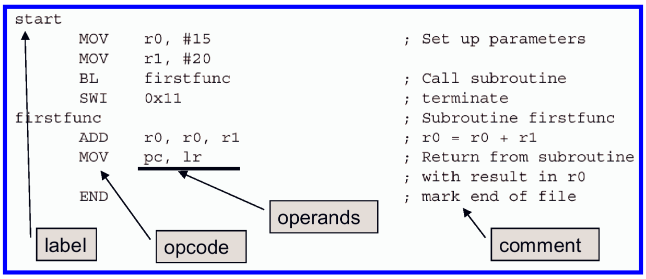
- The tools are intended for cross-development (they run on a different architecture from that for which they produce code).
- C compiler is supported by a library of standard functions.
- Assembler produces object format output that can be linked with output from the compiler. Substitutes names of variables with symbolic references.
- Linker takes one or more object files (sequences of assembly instructions) and combines them into executable format. Substitutes the symbolic references with actual addresses.
- Symbolic debugger (sd) assists in debugging programs running under instruction-set simulator (ARMulator) or on an ARM development board.

Assembly language programming

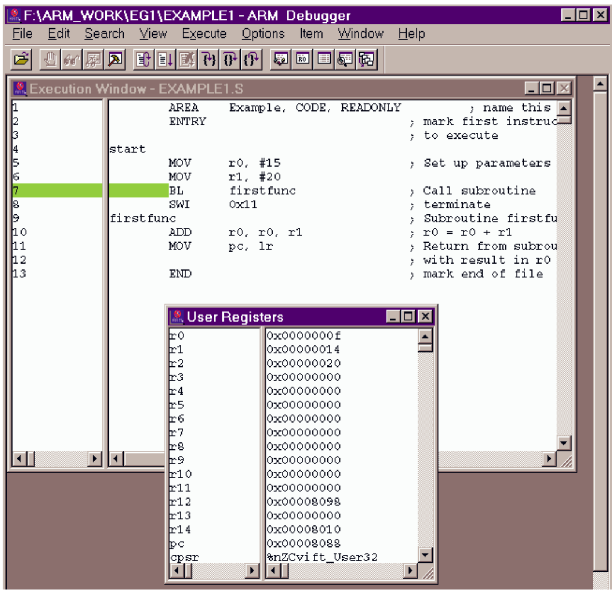
- Assembly language programming in ARM requires the programmer to think at the level of the ARM machine instruction.
- An ARM instruction is 32 bits, so there can be around 4 billion different binary machine instructions !
- Fortunately, there is considerable structure within the instruction space, so the programmer does not have to be familiar with each binary encoding.
- The assembler is a tool which handles most of these details for the programmer.
- After building a program, the programmer can invoke the debugger and step through each instruction (one at a time).
- The debugger allows the control of the execution and the observation of the registers and the memory locations.

Assembly language programming (cont'd)

Example of ARM program and debugger environment



```
start
        MOV     r0, #15                 ; Set up parameters
        MOV     r1, #20
        BL      firstfunc               ; Call subroutine
        SWI     0x11                    ; terminate
firstfunc                               ; Subroutine firstfunc
        ADD     r0, r0, r1              ; r0 = r0 + r1
        MOV     pc, lr                  ; Return from subroutine
                                        ; with result in r0
        END                             ; mark end of file
```



F:\ARM_WORK\EG1\EXAMPLE1 - ARM Debugger
File Edit Search View Execute Options Item Window Help
Execution Window - EXAMPLE1.S
User Registers

```
            AREA    Example, CODE, READONLY         ; name this
            ENTRY                                   ; mark first instruc
                                                    ; to execute
start
            MOV     r0, #15                         ; Set up parameters
            MOV     r1, #20
            BL      firstfunc                       ; Call subroutine
            SWI     0x11                            ; terminate
firstfunc                                           ; Subroutine firstfu
            ADD     r0, r0, r1                      ; r0 = r0 + r1
            MOV     pc, lr                          ; Return from subrou
                                                    ; with result in r0
            END                                     ; mark end of file
```

```
r0      0x0000000f
r1      0x00000014
r2      0x00000020
r3      0x00000000
r4      0x00000000
r5      0x00000000
r6      0x00000000
r7      0x00000000
r8      0x00000000
r9      0x00000000
r10     0x00000000
r11     0x00000000
r12     0x00008098
r13     0x00000000
r14     0x00008010
pc      0x00008088
cpsr    %nZCvift_User32
```

- The main routine of the program (labelled *start*) loads the values 15 and 20 into registers r0 and r1.
- The program then calls the subroutine *firstfunc* by using a branch instruction with link (*BL*).
- The subroutine adds together the two parameters it has received and places the result back into r0.
- It then returns by simply restoring the program counter to the address which was stored in the link register (r14) on entry.
- Upon return from the subroutine, the main program simply terminates using software interrupt (SWI 11). This instructs the program to exit cleanly and return control to the debugger.

ARM instructions

- ARM uses three types of instructions:
  - ✓ Data processing instructions (arithmetic operations, logical operations, register moves, comparisons, shift operations).
  - ✓ Data transfer instructions (register load/store instructions).
  - ✓ Control flow instructions (branch instructions).

Data processing instructions

- Rules apply to ARM data processing instructions:
  - All operands are 32 bits, come either from registers or are specified as constants in the instruction itself.
  - The result is also 32 bits and is placed in a register.
  - 3 operands are used: 2 for inputs and 1 for result.
- Example:

```
ADD     r0, r1, r2      ; r0 := r1 + r2
```

- Works for both unsigned and 2's complement signed numbers.
- This may produce carry out signal and overflow bits, but ignored by default.
- Result register can be the same as an input operand register.

Data processing instructions (cont'd)

- ARM's basic arithmetic operations:

```
ADD     r0, r1, r2      ; r0 := r1 + r2
ADC     r0, r1, r2      ; r0 := r1 + r2 + C
SUB     r0, r1, r2      ; r0 := r1 - r2
SBC     r0, r1, r2      ; r0 := r1 - r2 + C - 1
RSB     r0, r1, r2      ; r0 := r2 - r1
RSC     r0, r1, r2      ; r0 := r2 - r1 + C - 1
```

- RSB stands for reverse subtraction.
- Operands may be unsigned or 2's complement signed integers.
- 'C' is the carry bit in the CPSR.
- ADC, SBC, and RSC are used to operate on data more than 32 bits long in 32-bit architecture.

Data processing instructions (cont'd)

- For example, let’s add two 64-bit numbers X and Y, storing the result in Z
- We need two registers per number (store X in r1:r0, Y in r3:r2, and Z in r5:r4.
- Then:

```
ADDS        r4, r0, r2
ADC         r5, r1, r3
```

- S at the end of an instruction means you want to keep a record of the status of this operation in the CPSR
- Similarly, if we wanted to subtract the two numbers:

```
SUBS        r4, r0, r2
SBC         r5, r1, r3
```

Data processing instructions (cont'd)

- Why '+C –1' ?
- Remember that subtraction is performed by using 2’s complement addition (X-Y: find 2's complement of Y and add it to X).
- Finding the 2’s complement involves inverting all bits and adding 1.
- We only want to add one to the least significant bit in the least significant word.
- So for all following words, this “added 1” needs to be subtracted off.
- In practice, you will rarely want to use arithmetic on more than 32-bits.

Data processing instructions (cont'd)

- ARM's bit-wise logical operations:

```
AND    r0, r1, r2        ; r0 := r1 and r2 (bit-by-bit for 32 bits)
ORR    r0, r1, r2        ; r0 := r1 or r2
EOR    r0, r1, r2        ; r0 := r1 xor r2
BIC    r0, r1, r2        ; r0 := r1 and not r2
```

- BIC stands for 'bit clear', where every '1' in the second operand clears the corresponding bit in the first:

```
r1:     0101 0011 1010 1111 1101 1010 0110 1011
r2:     1111 1111 1111 1111 0000 0000 0000 0000
r0:     0000 0000 0000 0000 1101 1010 0110 1011
```

Data processing instructions (cont'd)

- ARM's register move operations:

```
MOV    r0, r2        ; r0 := r2
MVN    r0, r2        ; r0 := not r2
```

- MVN stands for 'move negated':

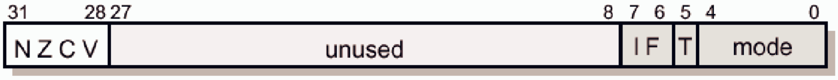
```
r2:     0101 0011 1010 1111 1101 1010 0110 1011
r0:     1010 1100 0101 0000 0010 0101 1001 0100
```

Data processing instructions (cont'd)

- ARM's register comparison operations:

```
CMP    r1, r2          ; set cc on r1 - r2
CMN    r1, r2          ; set cc on r1 + r2
TST    r1, r2          ; set cc on r1 and r2
TEQ    r1, r2          ; set cc on r1 xor r2
```

- Results of subtract, add, and, xor are NOT stored in any registers
- The condition code bits (cc) in the CPSR are set or cleared by these instructions.



- Take the CMN r1,r2 instruction:

  N = 1 if MSB of the addition (r1 + r2) result is '1', else N = 0.

  Z = 1 if the result of the addition is 0, else Z = 0.

  C is set to the carry-out of the addition.

  V is set to the overflow of the addition.

Data processing instructions (cont'd)

- ARM has a clever feature. In any data processing instructions, we can apply to the second register operand a shift operation. For example:

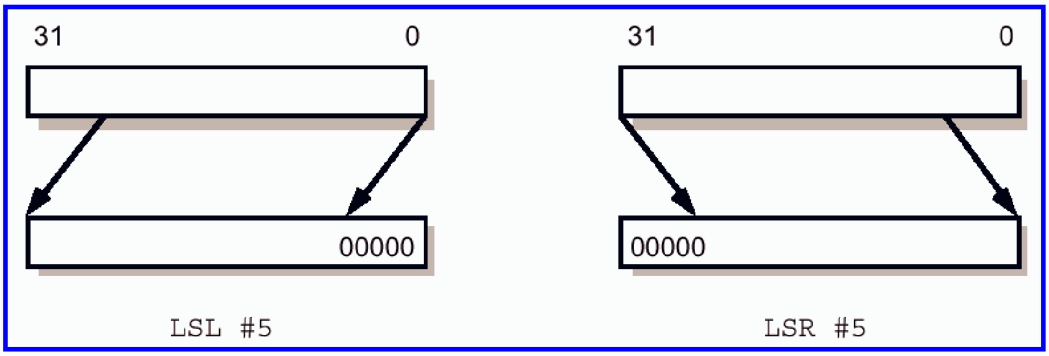
```
ADD     r3, r2, r1, LSL #3
```

- Here LSL means 'logical shift left by the specified number of bits'.
- Note that this is still a single ARM instruction, executed in a single clock cycle.
- In most processors, this is a separate instruction, while ARM integrates this shifting into the ALU.
- It is also possible to use a register value to specify the number of bits the second operand should be shifted by:

```
ADD     r5, r5, r3, LSL r2
```

Data processing instructions (cont'd)

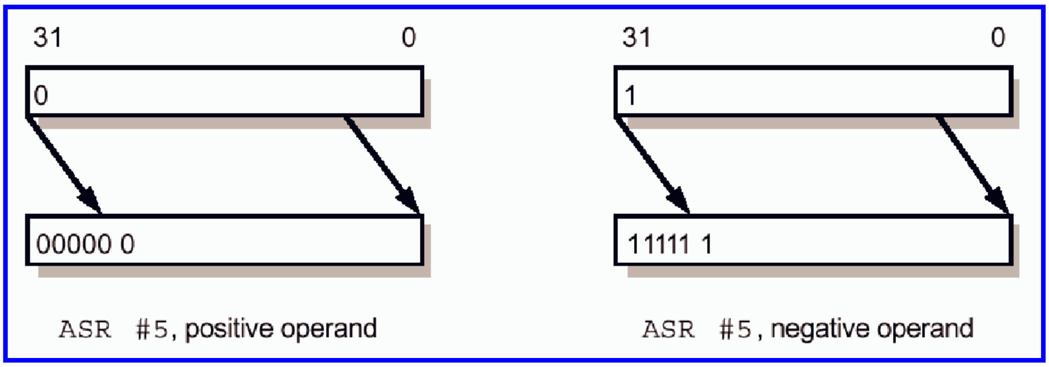
- Six possible ARM shift operations can be used:



- LSL: logical shift left by 0 to 31 places; fill the vacated bits at the least significant end of the word with zeros.
- LSR: logical shift right by 0 to 32 places; fill the vacated bits at the most significant end of the word with zeros.

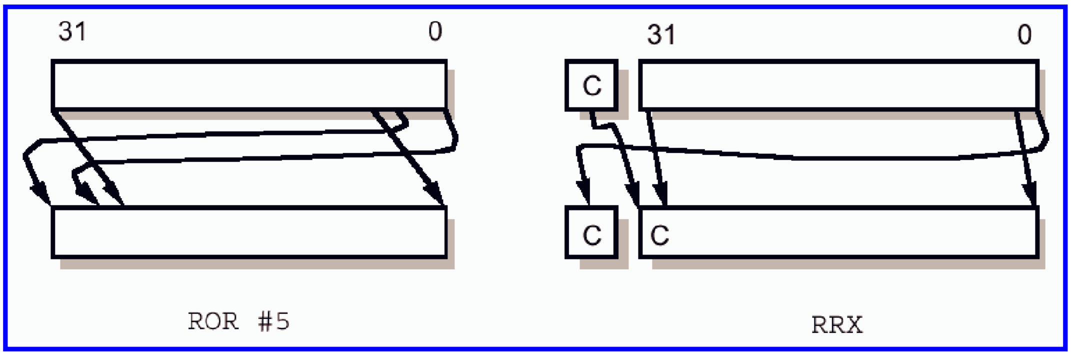
Data processing instructions (cont'd)

- ASL: arithmetic shift left; this is the same as LSL.
- ASR: arithmetic shift right by 0 to 32 places; fill the vacated bits at the most significant end of the word with zeros if the source operand was positive, and with ones it is negative.



Data processing instructions (cont'd)

- ROR: rotate right by 0 to 32 places; the bits which fall off the least significant end are used to fill the vacated bits at the most significant end of the word.
- RRX: rotate right extended by 1 place; the vacated bit (bit 31) is filled with the old value of the C flag and the operand is shifted one place to the right. This is effectively a 33 bit rotate using the register and the C flag.



Data processing instructions (cont'd)

- ARM has a number of multiply instructions:
  - Produce the product of two 32-bit binary numbers held in registers.
  - Result of 32-bit*32-bit is 64 bits. The entire 64-bit result is stored in two registers. Sometimes, only the 32 LSBs bits of the product are stored.
  - A Multiply-Accumulate instruction also adds the product to a running total.

```
MUL r4, r3, r2              ; r4 := r3 x r2
MLA r4, r3, r2, r1          ; r4 := (r3 x r2) + r1
```

- Differences from the other arithmetic operations:
  - Immediate second operands are not supported.
  - The result register cannot be the same as the second source register.

Data processing instructions (cont'd)

- When multiplying by a constant value, it is possible to replace the multiply instruction (load constant to register and use MUL) with a sequence of add and shift operations that have the same effect.

- For instance, multiply by 5 could be achieved using a single instruction:

```
ADD  Rd, Rm, Rm, LSL #2          ; Rd = Rm + (Rm * 4) = Rm * 5
```

- This is obviously faster than the MUL version:

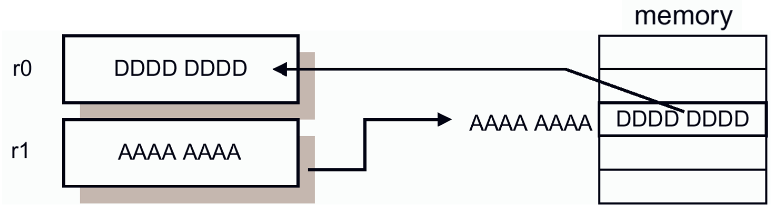
```
MOV Rs, #5
MUL Rd, Rm, Rs
```

Data transfer instructions

- Three basic forms of data transfer instructions:
  - Single register load/store instructions
  - Multiple register load/store instructions
  - Single register swap instructions (combined load and store)
- Use a value in one register (called the base register) as a memory address and either loads the data value from that address into a destination register or stores the register value to memory:

```
LDR    r0, [r1]              ; r0 := mem32[r1]
STR    r0, [r1]              ; mem32[r1] := r0
```

- This is called register-indirect addressing.
- LDR r0, [r1] operation:



r0: destination register

r1: base register

Data transfer instructions (cont'd)

- To load or store from or to a memory location, an ARM register must be initialized to contain the address of that location.

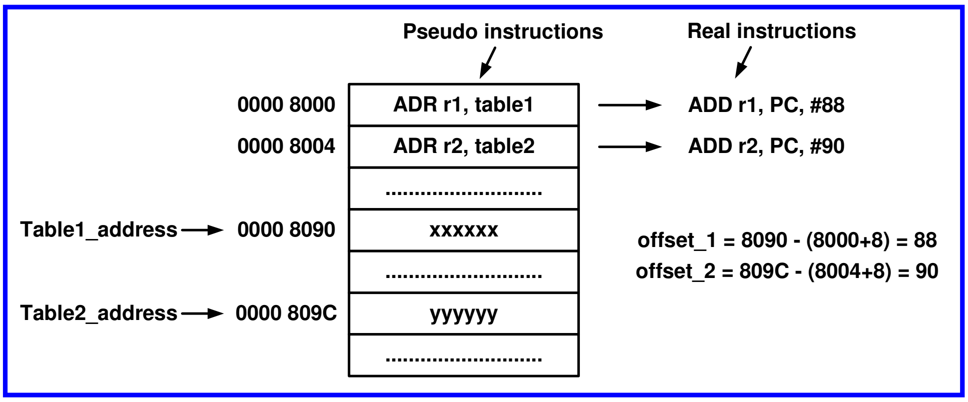
- In order to do that a pseudo instruction is used: ADR (the assembler translate it to a real data processing instruction).

- Example: Copy of data within memory from TABLE1 to TABLE2.

```
 copy          ADR    r1, TABLE1         ; r1 points to TABLE1
               ADR    r2, TABLE2         ; r2 points to TABLE2
               LDR    r0, [r1]           ; load first value ....
               STR    r0, [r2]           ;   and store it in TABLE2
               .......
TABLE1         ......                    ; <source of data>
               ......
TABLE2         ......                    ; <destination of data>
```

Data transfer instructions (cont'd)

- The way in which the ADR pseudo instruction works is: the desired data address value is often close to the code being executed and thus it is possible to exploit the fact that the program counter (r15) is close to that desired address.
- A data-processing instruction can be employed by the assembler to add a small offset (pc-relative offset) to r15.
- So, ADR instruction is translated by the assembler into an instruction that adds a constant to the pc value (r15) and puts the result in the proper register.
- The pc-relative offset is calculated as**: table1_address – (pc value + 8).** The value 8 accounts for the fetch and decode cycles of the current instruction. Note that, the memory is organized in Bytes.



0000 8000	ADR r1, table1	ADD r1, PC, #88
0000 8004	ADR r2, table2	ADD r2, PC, #90
	....	
0000 8090	xxxxxx	
	....	
0000 809C	yyyyyy	
	....	

offset_1 = 8090 - (8000+8) = 88

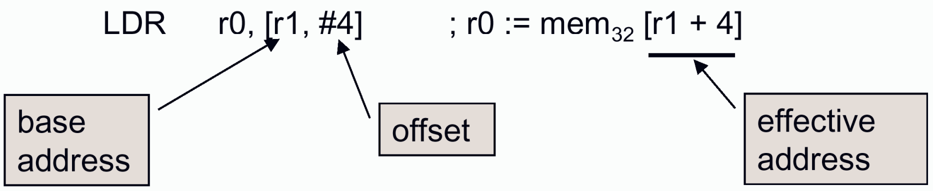
offset_2 = 809C - (8004+8) = 90

Data transfer instructions (cont'd)

- Extend the copy program further to copy the next word:

```
copy        ADR   r1, TABLE1    ; r1 points to TABLE1
            ADR   r2, TABLE2    ; r2 points to TABLE2
            LDR   r0, [r1]      ; load first value ….
            STR   r0, [r2]      ;    and store it in TABLE2
            ADD   r1, r1, #4    ; step r1 onto next word
            ADD   r2, r2, #4    ; step r2 onto next word
            LDR   r0, [r1]      ; load second value …
            STR   r0, [r2]      ;    and store it
            ……
```

- Simplify with pre-indexed addressing mode:



LDR r0, [r1, #4] ; r0 := mem_{32} [r1 + 4]

Data transfer instructions (cont'd)

- Pre-indexed addressing does not change r1. Sometimes, it is useful to modify the base register to point to the new address. This is achieved by adding a '!', and we then we have auto-indexing:

```
LDR    r0, [r1, #4]!     ;  r0 : = mem32 [r1 + 4]
                         ;  r1 := r1 + 4
```

- The '!' indicates that the instruction should update the base register after the data transfer

- In post-indexed addressing,the base address is used without an offset as the transfer address, after which it is always modified. Using this, we can improve the copy program more:

```
LDR r0, [r1], #4 ;
r0:= mem32 [r1]
r1:= r1 + 4
```

```
copy        ADR   r1, TABLE1      ; r1 points to TABLE1
            ADR   r2, TABLE2      ; r2 points to TABLE2
loop        LDR   r0, [r1], #4    ; get TABLE1 1st word ….
            STR   r0, [r2], #4    ;    copy it to TABLE2
            ???                   ; if more, go back to loop
            ……
TABLE1      ……                    ;  < source of data >
```

LDR and STR instructions are repeated until the required number of values has been copied into TABLE2, and then the loop is exited.

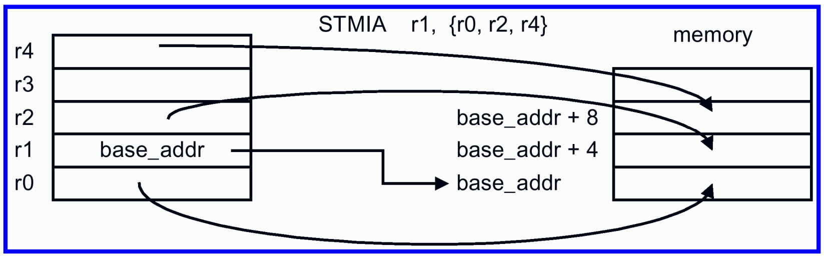
Data transfer instructions (cont'd)

- As another variation, the size of the data item which is transferred may be a single 8-bit byte instead of a 32-bit word. This option is selected by adding a letter B onto the symbolic operation code:

LDRB r0, [r1] ; r0 : = mem$_8$ [r1]

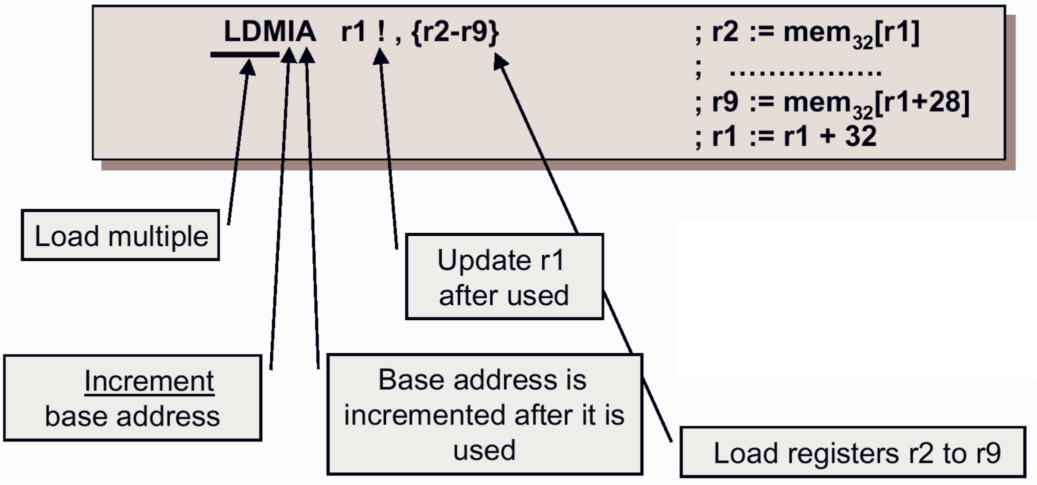
- LDR and STR instructions only load/store a single 32-bit word.
- ARM can load/store any subset of its registers in a single instruction by using load/store multiple instructions. For example:

LDMIA r1, {r0, r2, r4} ; r0 := mem$_{32}$[r1]
; r2 := mem$_{32}$[r1+4]
; r4 := mem$_{32}$[r1+8]



Data transfer instructions (cont'd)

- The base address register (r1) has not been changed. We can update this pointer register by adding '!' after it:


LDMIA r1 ! , {r2-r9} ; r2 := mem$_{32}$[r1]
; ....
; r9 := mem$_{32}$[r1+28]
; r1 := r1 + 32

Data transfer instructions (cont'd)

- Example of moving 8 words from a source memory location to a destination memory location:

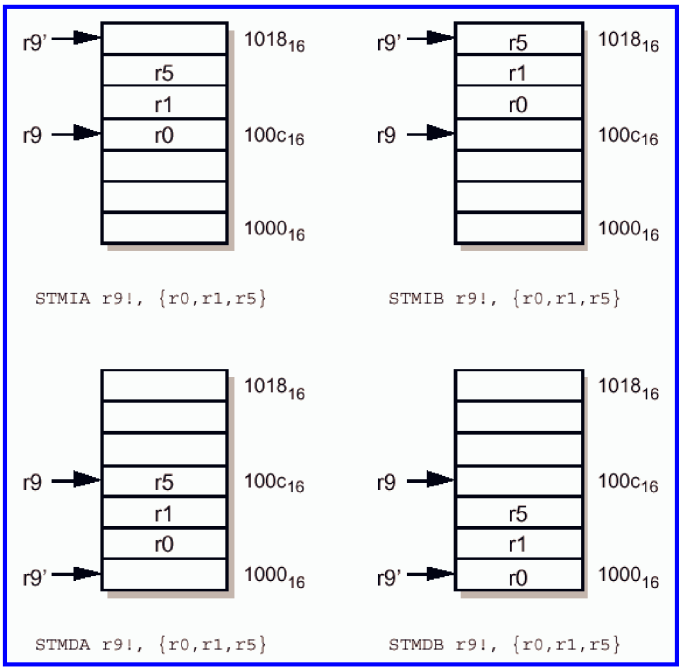
```
        ADR     r0, src_addr       ; initialize src addr
        ADR     r1, dest_addr      ; initialize dest addr
        LDMIA   r0!, {r2-r9}       ; fetch 8 words from mem
                                   ;    … and update r0 := r0 + 32
        STMIA   r1, {r2-r9}        ; copy 8 words to mem, r1 unchanged
```

- When using LDMIA and STMIA instructions we:
  1. **INCREMENT** the address in memory to load/store our data.
  2. The increment of the address occurs **AFTER** the address is used.
- In fact, one could use four different forms of load/store:

Increment -	After	LDMIA and STMIA
Increment -	Before	LDMIB and STMIB
Decrement -	After	LDMDA and STMDA
Decrement -	Before	LDMDB and STMDB

Data transfer instructions (cont'd)

Multiple register transfer addressing modes



Control flow instructions

- The basic branch instruction is:

```
          B         label     ; unconditionally branch to label
          ……
          ……
label     …….
```

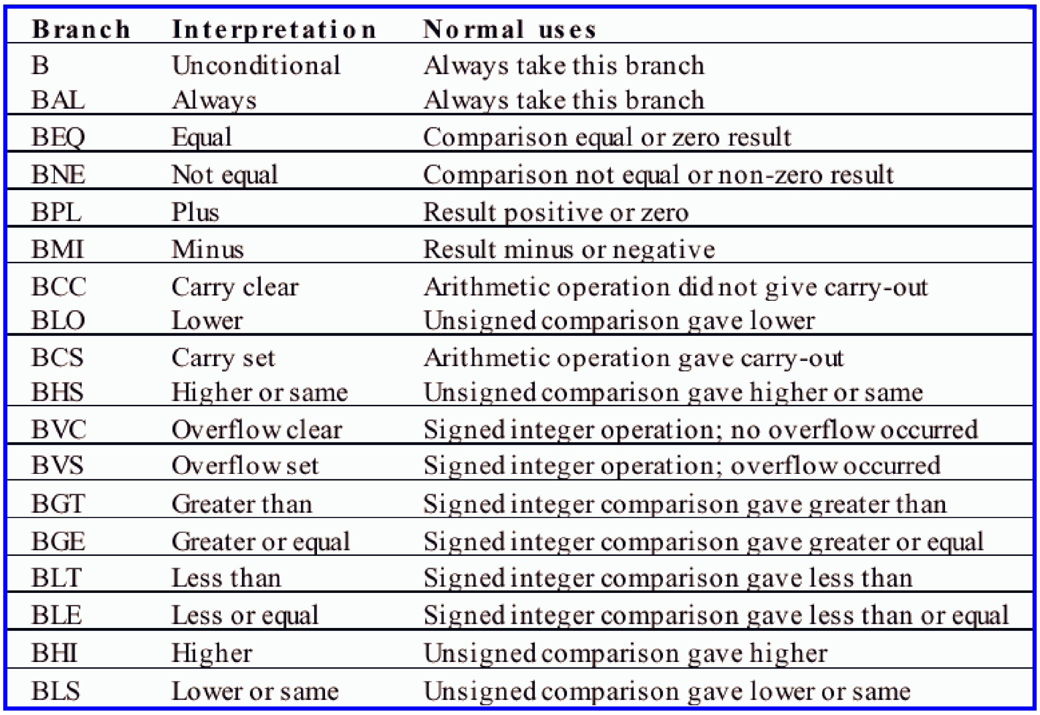
- Conditional branch instructions can be used to control loops:

```
          MOV       r0, #10             ; intialize loop counted r0
loop      …….                           ; start of body of loop
          …….
          SUB       r0, r0, #1          ; decrement loop counter
          CMP       r0, #0              ;   is it zero yet?
          BNE       loop                ;   repeat if r0 ≠ 0, else fall through
```

- The CMP instruction is a subtraction, which gives no results, except possibly changing conditional codes in the CPSR register.
- If r0=0, then Z bit in CPSR is set to 1, else Z bit is reset to 0.

Control flow instructions (cont'd)

ARM branch conditions



Branch	Interpretation	Normal uses
B	Unconditional	Always take this branch
BAL	Always	Always take this branch
BEQ	Equal	Comparison equal or zero result
BNE	Not equal	Comparison not equal or non-zero result
BPL	Plus	Result positive or zero
BMI	Minus	Result minus or negative
BCC	Carry clear	Arithmetic operation did not give carry-out
BLO	Lower	Unsigned comparison gave lower
BCS	Carry set	Arithmetic operation gave carry-out
BHS	Higher or same	Unsigned comparison gave higher or same
BVC	Overflow clear	Signed integer operation; no overflow occurred
BVS	Overflow set	Signed integer operation; overflow occurred
BGT	Greater than	Signed integer comparison gave greater than
BGE	Greater or equal	Signed integer comparison gave greater or equal
BLT	Less than	Signed integer comparison gave less than
BLE	Less or equal	Signed integer comparison gave less than or equal
BHI	Higher	Unsigned comparison gave higher
BLS	Lower or same	Unsigned comparison gave lower or same

Control flow instructions (cont'd)

- Conditional execution applies not only to branches, but to all ARM instructions.
- For example:

```
           CMP    r0, #5              ; if (r0 != 5) then
           BEQ    BYPASS
           ADD    r1, r1, r0          ;     r1 := r1 + r0 - r2
           SUB    r1, r1, r2
BYPASS  .....
```

- Can be replaced by a smaller and faster sequence:

```
                CMP      r0, #5              ; if (r0 != 5) then
                ADDNE  r1, r1, r0            ;     r1 := r1 + r0 - r2
                SUBNE  r1, r1, r2
BYPASS          .....
```

- Here the ADDNE and SUBNE instructions are executed only if Z= 0, i.e. the CMP instruction gives non-zero result.

Control flow instructions (cont'd)

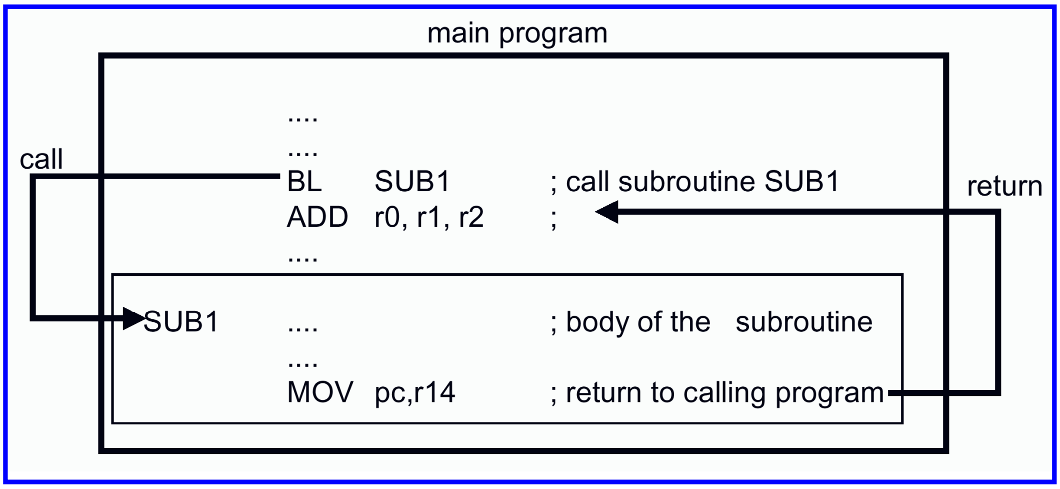
- Here is another very clever use of this unique feature in ARM instruction set (all instructions can be qualified by the condition codes).

```
if ((a=b) & (c=d)) then e := e + 1;
          CMP      r0, r1             ; r0 has a, r1 has b
          CMPEQ r2, r3                ; r2 has c, r3 has d
          ADDEQ r4, r4, #1            ; e := e+1
```

- Note how if the first comparison finds unequal operands, the second and third instructions are both skipped.
- The logical 'and' in the 'if' sentence is implemented by making the second comparison conditional.
- Conditional execution is only efficient if the conditional sequence is three instructions or fewer. If the conditional sequence is longer, we have to use branches.

Control flow instructions (cont'd)

- Subroutines allow you to modularize your code so that they are more reusable.
- The general structure of a subroutine in a program is:



```
        ....
        ....
        BL      SUB1          ; call subroutine SUB1
        ADD     r0, r1, r2    ;
        ....
SUB1    ....                  ; body of the   subroutine
        ....
        MOV     pc,r14        ; return to calling program
```

r14 is the link register used for storing the return address

Control flow instructions (cont'd)

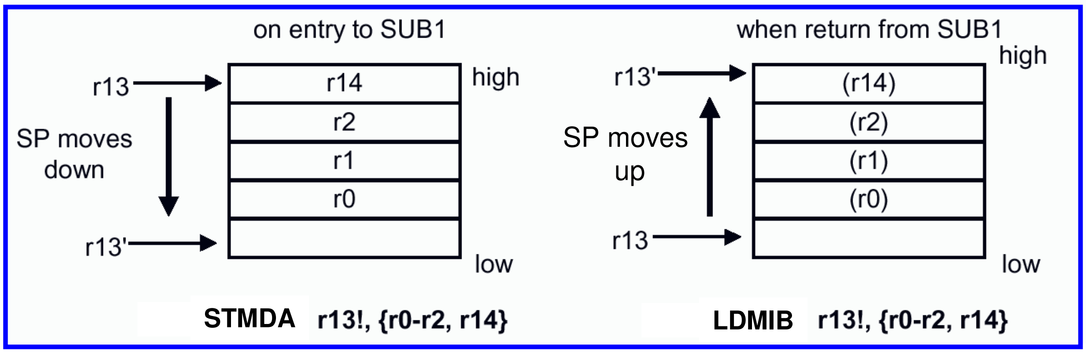
Nested subroutines

- Since the return address is held in register r14, we can not call a further subroutine without first saving r14.
- It is also a good software engineering practice that a subroutine does not change any register values except when passing results back to the calling program.
- This is the principle of information hiding: try to hide what the subroutine does from the calling program.
- In order to achieve these two goals we use the memory stack to:
  1. Preserve r14.
  2. Store and then retrieve the values of registers used inside the subroutine.

Control flow instructions (cont'd)

Preserve things inside a subroutine using the stack

```
      BL     SUB1
      .....
SUB1  STMDA  r13!, {r0-r2, r14}      ; push work & link registers
      ....
      BL     SUB2                    ; jump to a nested subroutine
      ...
      LDMIB  r13!, {r0-r2, r14}      ; pop work & link registers
      MOV    pc, r14                 ; return to calling program
```

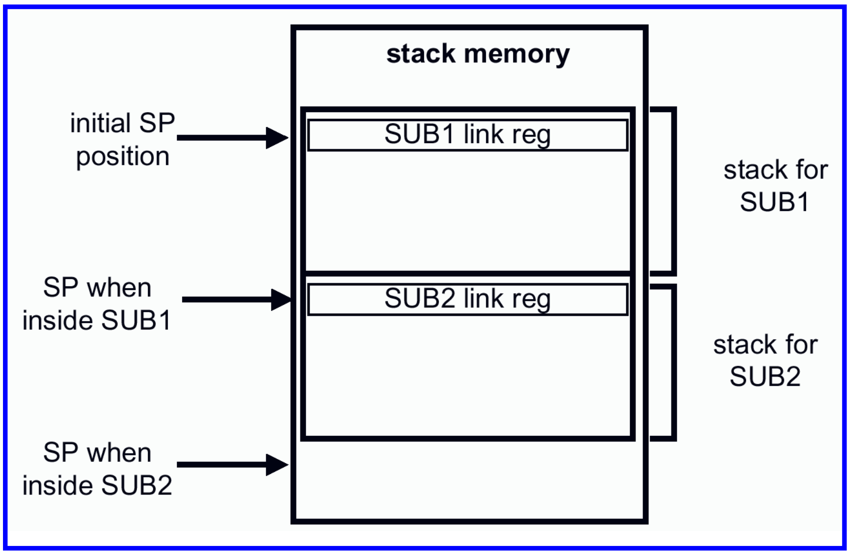


r13 is used as stack pointer

Control flow instructions (cont'd)

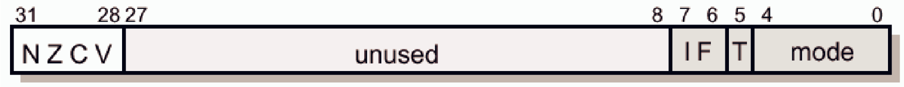
Effects of subroutine nesting

- SUB1 calls another subroutine SUB2. Assuming that SUB2 also saves its link register (r14) and its working registers on the stack, a snap-shot of the stack will look like:



Instruction set encoding

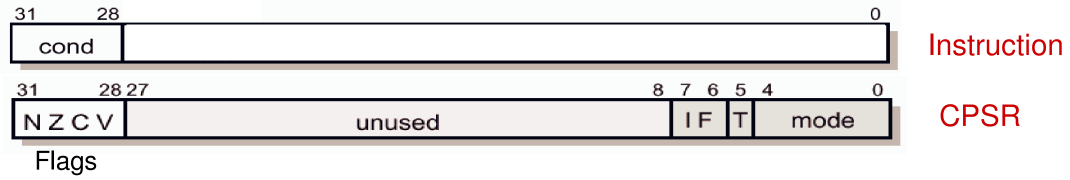
- ARM instructions are all 32-bits words aligned on 4-byte boundaries.
- Some ARMs can execute a compressed form of the instruction set where a subset of the full instruction set is encoded into 16-bit instructions (Thumb instruction set).
- Internally, all ARM operations are on 32-bit operands and the shorter data types (e.g. 8-bit) are only supported by data transfer functions. When a byte is loaded from memory (byte-addressed memory), it is extended to 32 bits for internal processing.
- Most programs operate in the user mode, however ARM has extra operating modes which are used to handle exceptions and software interrupts. The operating mode is defined by the bottom five bits of the CPSR.



CPSR[4:0]	Mode	Use	Registers
10000	User	Normal user code	user
10001	FIQ	Processing fast interrupts	_fiq
10010	IRQ	Processing standard interrupts	_irq
10011	SVC	Processing software interrupts (SWIs)	_svc
10111	Abort	Processing memory faults	_abt
11011	Undef	Handling undefined instruction traps	_und
11111	System	Running privileged operating system tasks	user

Instruction set encoding (cont'd)

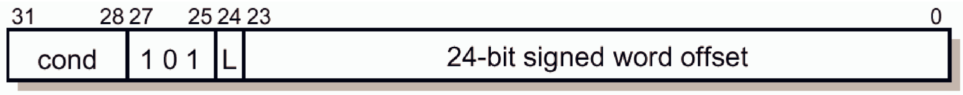
- ARM has extend the conditional execution to all of its instructions. The condition field occupies the top four bits of each 32-bit instruction.
- Each of the 16 values of the condition field caused the instruction to be executed or skipped according to the values of the flags in the CPSR.



Opcode [31:28]	Mnemonic extension	Interpretation	Status flag state for execution
0000	EQ	Equal / equals zero	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set / unsigned higher or same	C set
0011	CC/LO	Carry clear / unsigned lower	C clear
0100	MI	Minus / negative	N set
0101	PL	Plus / positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N equals V
1011	LT	Signed less than	N is not equal to V
1100	GT	Signed greater than	Z clear and N equals V
1101	LE	Signed less than or equal	Z set or N is not equal to V
1110	AL	Always	any
1111	NV	Never (do not use!)	none

Instruction set encoding (cont'd)

Branch instructions

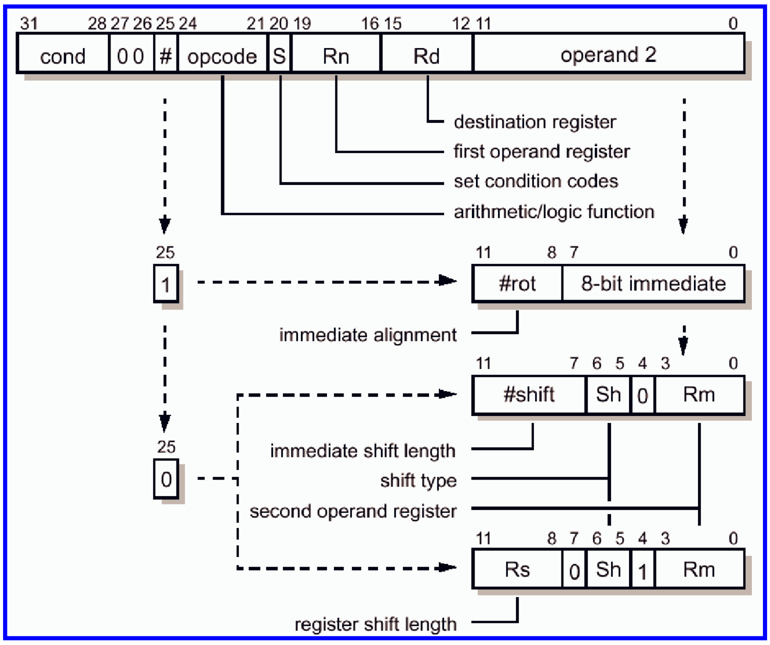


- Bits [31:28] are used to specify the condition under which the instruction is executed.
- Bits [25:27] identify that this is a B (branch) or BL (branch with link) instruction.
- The L bit (24) is set to 1 if it is a branch with link and is cleared to 0 if it is a single branch.
- The 24-bit signed word offset specifies the destination address.
- In order to compute the destination address of the branch instruction, the assembler adds the offset to the PC which contains the address of the branch instruction plus 8 bytes.
- In the case of a BL instruction (L = 1), the address of the instruction following the branch is moved into the link register (r14).
- This is used to perform a subroutine call and cause a return by copying the link register back to the PC.

Instruction set encoding (cont'd)

Data processing instructions

Use 3-address format: 1st operand is always register, 2nd operand is register, shifted register or immediate value, result is always a register.



S-bit controls condition code bits of CPSR:

N flag - set if result is negative (N is bit 31 of result).

Z flag - set if result is zero

C flag - set if there is a carry-out from ALU during arithmetic arithmetic operations.

V flag - set in an arithmetic operation if there is an overflow.

For immediate value second operand, only rotation is possible.

Shift type: in 2nd register shift can be logical, arithmetic or rotate.

Instruction set encoding (cont'd)

Opcode [24:21]	Mnemonic	Meaning	Effect
0000	AND	Logical bit-wise AND	Rd := Rn AND Op2
0001	EOR	Logical bit-wise exclusive OR	Rd := Rn EOR Op2
0010	SUB	Subtract	Rd := Rn - Op2
0011	RSB	Reverse subtract	Rd := Op2 - Rn
0100	ADD	Add	Rd := Rn + Op2
0101	ADC	Add with carry	Rd := Rn + Op2 + C
0110	SBC	Subtract with carry	Rd := Rn - Op2 + C - 1
0111	RSC	Reverse subtract with carry	Rd := Op2 - Rn + C - 1
1000	TST	Test	Scc on Rn AND Op2
1001	TEQ	Test equivalence	Scc on Rn EOR Op2
1010	CMP	Compare	Scc on Rn - Op2
1011	CMN	Compare negated	Scc on Rn + Op2
1100	ORR	Logical bit-wise OR	Rd := Rn OR Op2
1101	MOV	Move	Rd := Op2
1110	BIC	Bit clear	Rd := Rn AND NOT Op2
1111	MVN	Move negated	Rd := NOT Op2

Instruction set encoding (cont'd)

Multiply instructions

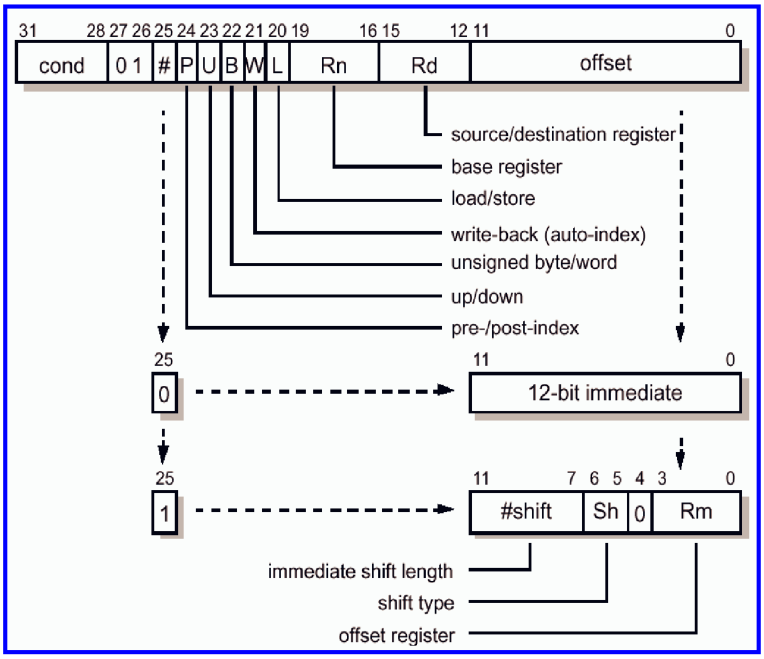
31 28	27 24	23 21	20	19 16	15 12	11 8	7 4	3 0
cond	0 0 0 0	mul	S	Rd/RdHi	Rn/RdLo	Rs	1 0 0 1	Rm

Opcode [23:21]	Mnemonic	Meaning	Effect
000	MUL	Multiply (32-bit result)	Rd := (Rm * Rs) [31:0]
001	MLA	Multiply-accumulate (32-bit result)	Rd := (Rm * Rs + Rn) [31:0]
100	UMULL	Unsigned multiply long	RdHi:RdLo := Rm * Rs
101	UMLAL	Unsigned multiply-accumulate long	RdHi:RdLo += Rm * Rs
110	SMULL	Signed multiply long	RdHi:RdLo := Rm * Rs
111	SMLAL	Signed multiply-accumulate long	RdHi:RdLo += Rm * Rs

- RdHi:RdLo is the 64-bit number formed by combined the 32 MSBs and the 32 LSBs of the result.
- The S bit controls the condition codes of the CPSR as with the other data processing instructions:
  - ✓ The N flag is set to the value of bit 31 of Rd for the case of 32-bit result, and to the bit 31 of RdHi for the case of long result.
  - ✓ The Z flag is set to 1 if Rd or RdHi or RdLo are zero.
  - ✓ The C flag is set to a meaningless value, and the V flag is unchanged.

Instruction set encoding (cont'd)

Data transfer instructions



P = 1 means pre-indexed, e.g. LDR r0, [r1,#4]
P = 0 means post-indexed, e.g. LDR r0, [r1], #4
B = 1 selects byte transfer (B = 0 is word transfer)
W = 1 is "write-back", e.g. LDR r0, [r1,#4]!

'offset' may be 12-bit unsigned immediate value
'offset' may also be a shifted register
The sign of the offset is determined by U. U=1 => add offset, U=0 => subtract offset
L=1 means 'load', L=0 means 'store'

Instruction set encoding (cont'd)

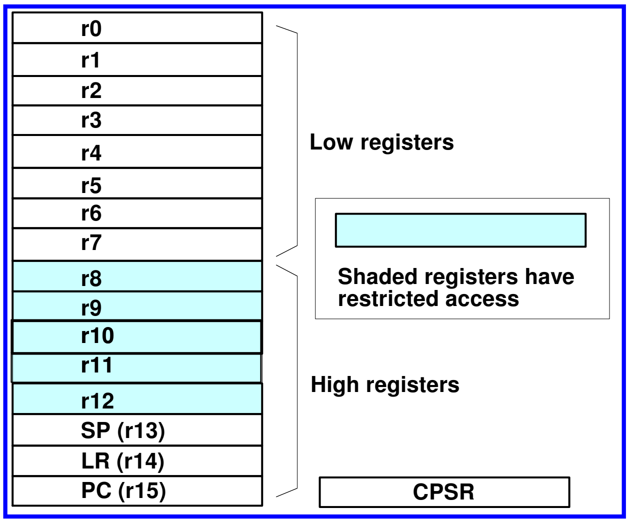
Software interrupts

- SWI (software interrupts) are used mainly for communication with external devices, e.g. monitor.
- Various SWs are available in the ARM processor.
- The instruction format is:

31 28	27 24	23 0
cond	1 1 1 1	24-bit immediate (SWI number)

Thumb instruction set

- ARM Thumb instruction set addresses the issue of code density.
- It is a compressed form of a subset of the standard ARM instruction set. Thumb instructions are mapped onto the ARM instructions, and the processor uses a decompression scheme in the ARM instruction pipeline in order the Thumb instructions to be executed as standard ARM instructions within the processor.
- Thumb instruction set operate on a restricted view of the ARM registers.



r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
SP (r13)
LR (r14)
PC (r15)

Low registers

Shaded registers have restricted access

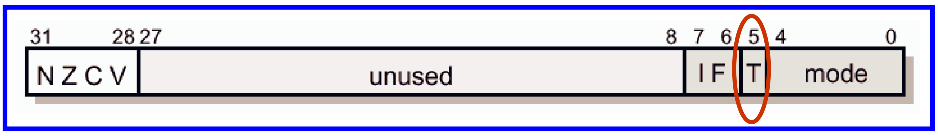
High registers

CPSR

- There is full access to the eight low registers (r0-r7).
- r13-r15 are used for special purposes (stack pointer, link register and program counter.
- Only few instructions are allowed to use the high registers (r8-r12).
- All Thumb instructions are 16-bit.
- Most Thumb instructions are executed unconditionally.
- Many Thumb data processing instructions use as destination register one of the source registers.
- Thumb instruction formats are less regular (as a result of dense encoding).

Thumb instruction set (cont'd)

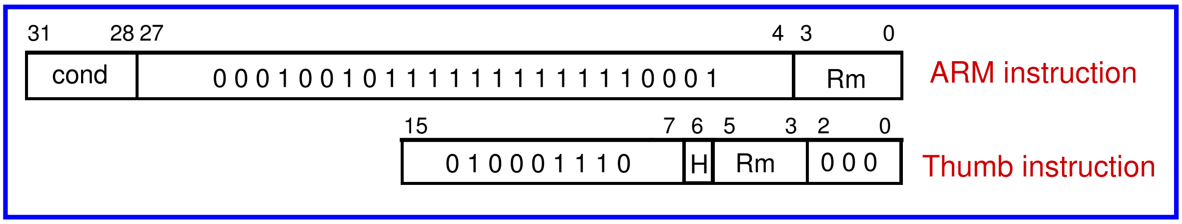
- If T in the CPSR is set to 1, the processor interprets the instruction stream as 16-bit Thumb instructions. Otherwise it interprets the stream as standard ARM instructions.



31 28 27 8 7 6 5 4 0

N Z C V | unused | I F | T | mode

- The normal way to switch between Thumb and standard ARM operation is by executing a BX (branch and exchange) instruction. This instruction is available in both modes of operation. The bottom bit of Rm is copied into the T bit of the CPRS, and the PC is switched to the address given in the remainder of the register.



31 28 27 4 3 0

cond | 0 0 0 1 0 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 1 | Rm — ARM instruction

15 7 6 5 3 2 0

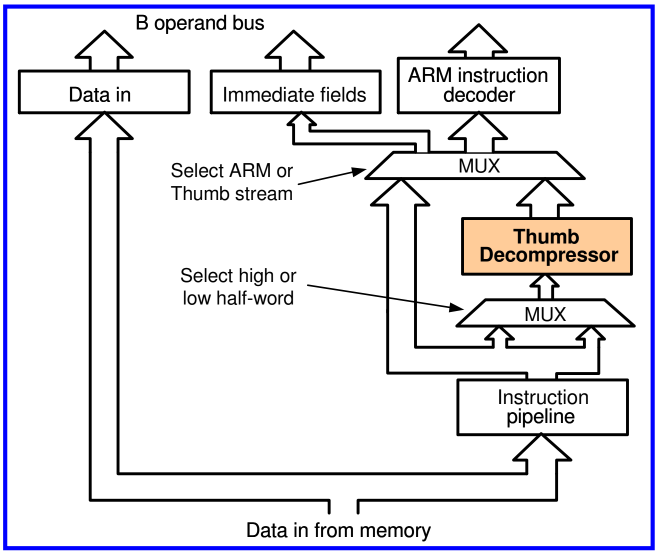
0 1 0 0 0 1 1 1 0 | H | Rm | 0 0 0 — Thumb instruction

Thumb instruction set (cont'd)

- Although ARM supports a shift operation together with the ALU operation in a single instruction, the Thumb instruction set separates shift and ALU operations into two different instructions.
- In standard ARM instruction set the shift operation is used as an operand modifier, while in Thumb is used as an individual opcode.
- Thumb instructions include a subset of the standard instructions covering the most commonly used operations required by the compiler.
- The selection of those to include and those to leave out was based on a detailed understanding of the needs of typical application programs.
- In Thumb instructions, a more complex encoding is used due to the availability of less encoding bits (16 bits in comparison with 32 bits of the standard ARM instruction set), and the restricted use of the system's registers.

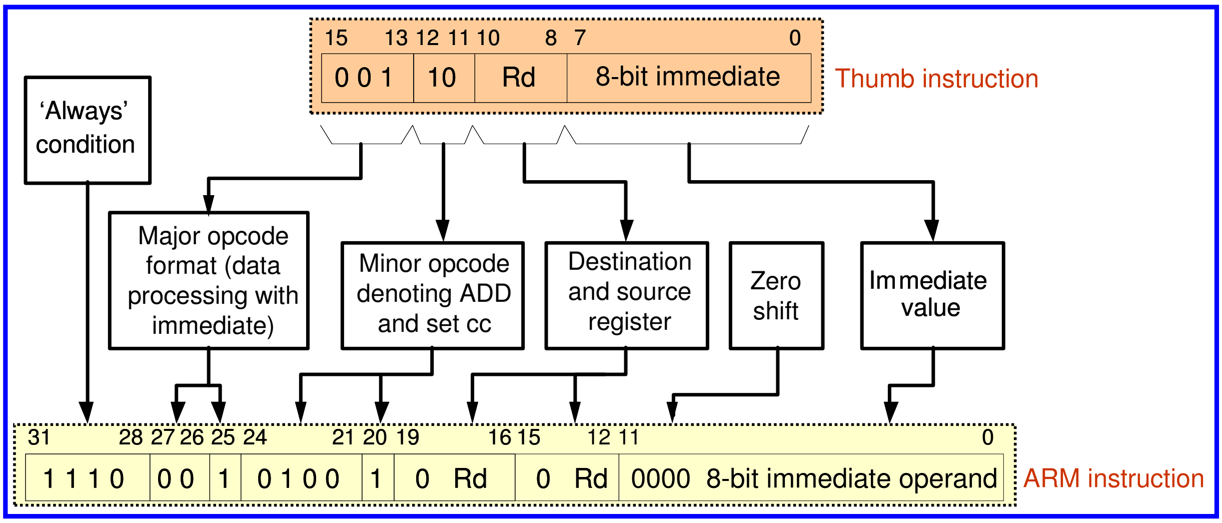
Thumb instruction set (cont'd)

- Thumb instruction set can be incorporated into an ARM processor with some changes to the processor logic.
- The major addition is the Thumb instruction decompressor in the instruction pipeline, which is a logic circuit that translates the Thumb instructions to its equivalent ARM instructions.



Thumb instruction set (cont'd)

- Example: Mapping of Thumb ADD instruction with immediate operand to the equivalent standard instruction.
- Since the only conditional Thumb instructions are branches, the condition ‘always’ is used during the translation of all Thumb instructions.
- The Thumb 2-address format is mapped into the ARM 3-address format by replicating a register specifier.

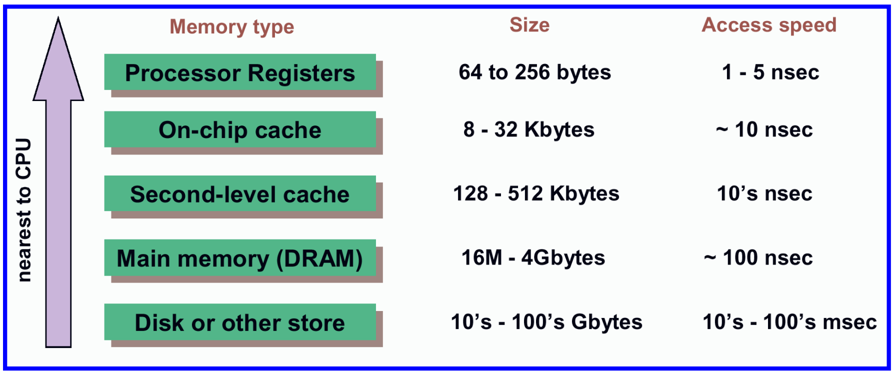


Thumb instruction set (cont'd)

- Thumb instructions are 16-bit long and encode the functionality of an ARM instruction in half number of bits.
- However, since Thumb instructions has less info than a standard instruction, a particular program will require more Thumb instructions than standard ARM instructions.
- Statistics from typical applications:
  - ✓ Thumb code requires 70% of the ARM code space.
  - ✓ Thumb code uses 40% more instructions.
  - ✓ ARM code is 40% faster for 32-bit memory, while Thumb code is 45% faster for 16-bit memory (since always memory is organised in Bytes, 32 or 16-bit memory accounts for the size of word transferred in each cycle).
  - ✓ Thumb code uses 30% less external memory.
- Thus, where performance is the important parameter, a system should use 32-bit memory and run ARM code, while where cost and power consumption are more important, 16-bit memory and Thumb code is a better choice.
- An ARM system may use Thumb code for non-critical routines to save power and memory requirements (e.g. user interface in a mobile telephone where real-time DSP functions require the full-power of the standard ARM).

Cache memory support

- ARM processor access memory during the fetching of the instructions and during loading and storing of data. The processor is much faster than the main memory.
- Cache memories (on-chip) attempt to bridge the processor-memory performance gap.
- They are small and high-speed memories that keep recently referenced instructions and data close to the processor, based on the expectation that they will be referenced again soon.
- For parallel access during fetching and storing/loading we use separate instruction and data caches.



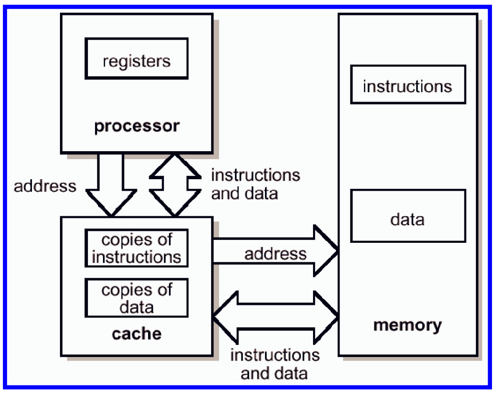
nearest to CPU

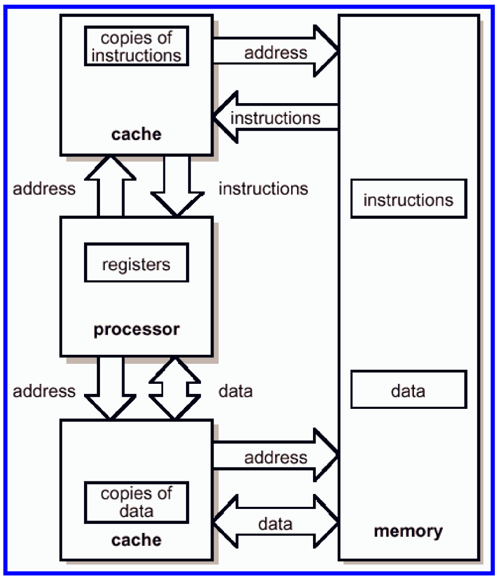
Memory type	Size	Access speed
Processor Registers	64 to 256 bytes	1 - 5 nsec
On-chip cache	8 - 32 Kbytes	~ 10 nsec
Second-level cache	128 - 512 Kbytes	10's nsec
Main memory (DRAM)	16M - 4Gbytes	~ 100 nsec
Disk or other store	10's - 100's Gbytes	10's - 100's msec

Memory hierarchy

Cache memory support (cont'd)

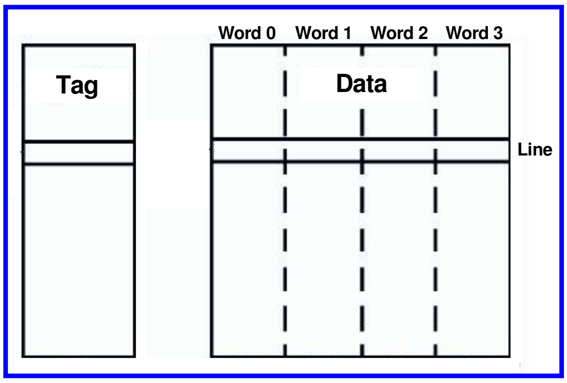
- A 'HIT' in the instruction cache (or in the data cache) has as result the instruction fetch stage (or the data transfer stage) to take only one cycle.
- A 'MISS' causes a stall in the instruction fetching for possibly many cycles until the instruction is retrieved from the main memory or a stall to the load/store instruction execution until the data transfer from/to the main memory. In addition, the next instructions in the pipeline are stalled.
- The problem with the use of caches is that they introduce uncertainty in terms of latency for accessing the main memory, which is a problem for safe scheduling in real-time systems.





Cache memory support (cont'd)

- How to know if a data item is in the cache ?
- A cache line is typically more than one word.
- In direct-mapped cache a line of data is stored along with an address tag (containing the top bits of the address which are not used to select within a line).
- Example: 4KBytes Cache with M = 4 words per line and N = (4096/16) = 256 lines.
- A mapping of the memory addresses to the cache memory is needed.
- The coding of the 256 lines needs 8 bits, while the coding of the 4 words within each line need 2 additional bits, leaving 22 bits for cache tag.



Direct-mapped cache memory organization

Indexes calculations:

Line index: (address / M) *mod* N

Word index: address *mod* M

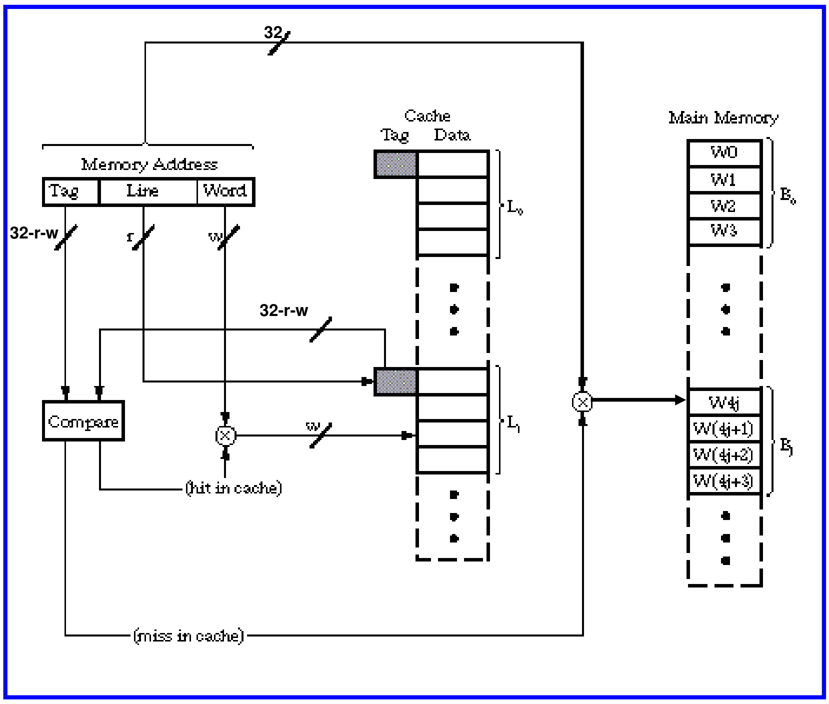
Cache tag	Line index	Word index
22	8	2

Address mapping used to access a cache entry

Cache memory support (cont'd)

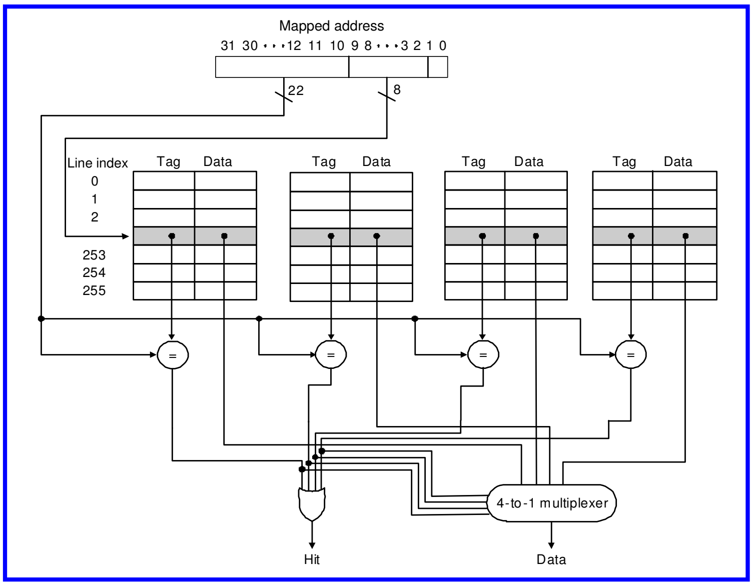
- The top address bits are compared with the stored tag bits, and if they are equal, the item is in the cache (HIT).
- Line and word index bits are used to find the correct line and word of the data to be used by the data path of the processor.
- When a MISS occurs, data cannot be read from the cache. A slower read from the main memory will take place.

Direct-mapped Cache memory operation



Cache memory support (cont'd)

- More complex organizations consist of multiple direct-mapped cache schemes operating in parallel (set-associative cache).
- For example in a 4-way set-associative cache, an address mapped to the cache may find its data in either of the 4 direct-mapped schemes, so each address may be stored in either of 4 places. This decreases the MISS rate in the cache memory.



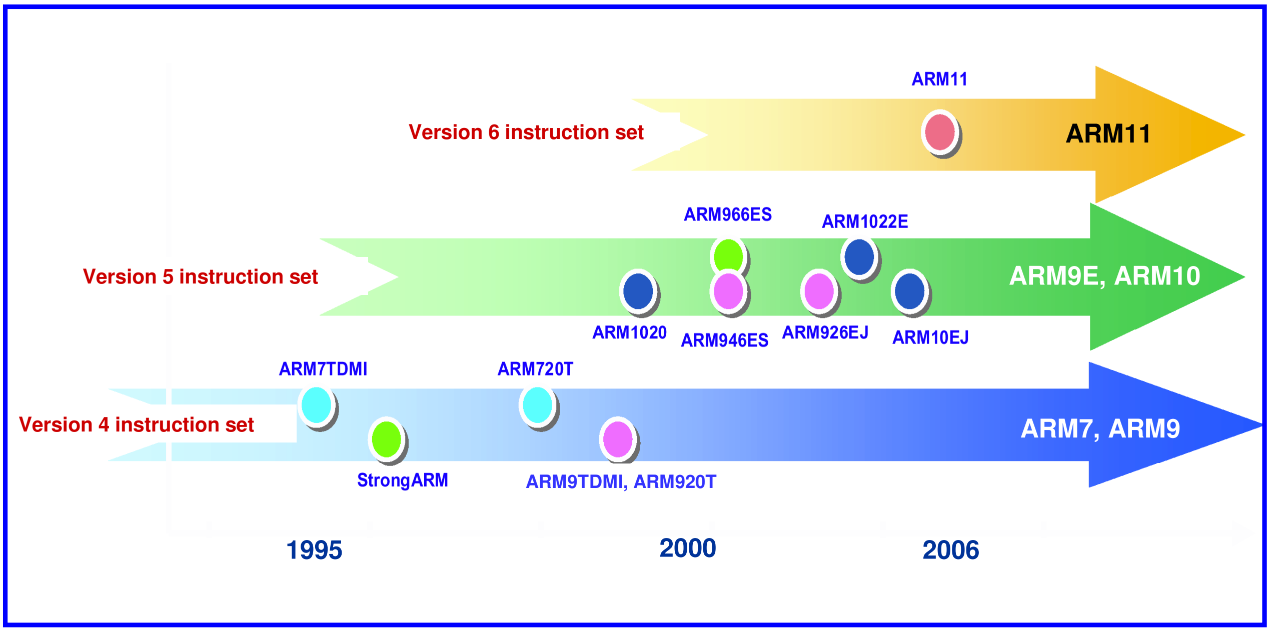
Cache memory support (cont'd)

- ARM710T and ARM720T:
  - ✓ Contain a unified 8 KBytes cache memory.
  - ✓ It is organized as a four-way set associative cache (four direct-mapped cache schemes operating in parallel).
  - ✓ Consists of 512 lines of 4 words each.
- ARM920T:
  - ✓ Contain two 16 KBytes caches (one for instructions and a second for data).
  - ✓ They are organized as eight-way set associative caches.
  - ✓ Consist of 512 lines of 8 words each.
- Processors of the ARM10E family contain two 16 KBytes caches (ARM1022E) or two 32 KBytes caches (ARM1020E).

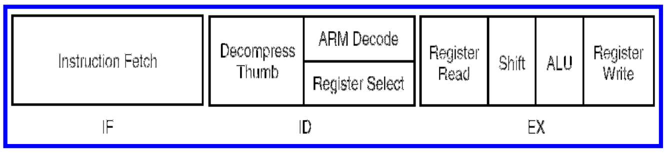
Evolution of the ARM architecture

- The successive ARM families represent a significant shift in the pipeline design.
- ARM7 family has a simple pipeline architecture with only three pipeline stages.
- ARM9 and ARM9E increases the number of stages from three to five, while ARM10 to six and ARM11 to eight.
- Addition and enhancement of cache memories and MMUs. Only a couple of ARM7 processors contain unified caches, while ARM9-11 contain separate data and instruction caches.
- There are often enhancements at the instruction set by inserting arithmetic instructions which are suitable for DSP applications, at the processor structure by inserting additional arithmetic blocks (e.g. extra adders for address calculation in load-store instructions, or other enhancements such as the use of branch prediction (static or dynamic).

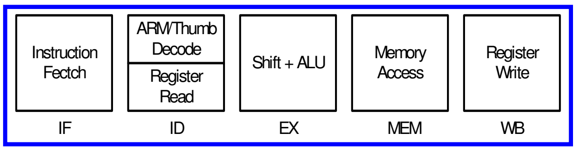
Evolution of the ARM architecture (cont'd)



ARM9 architecture



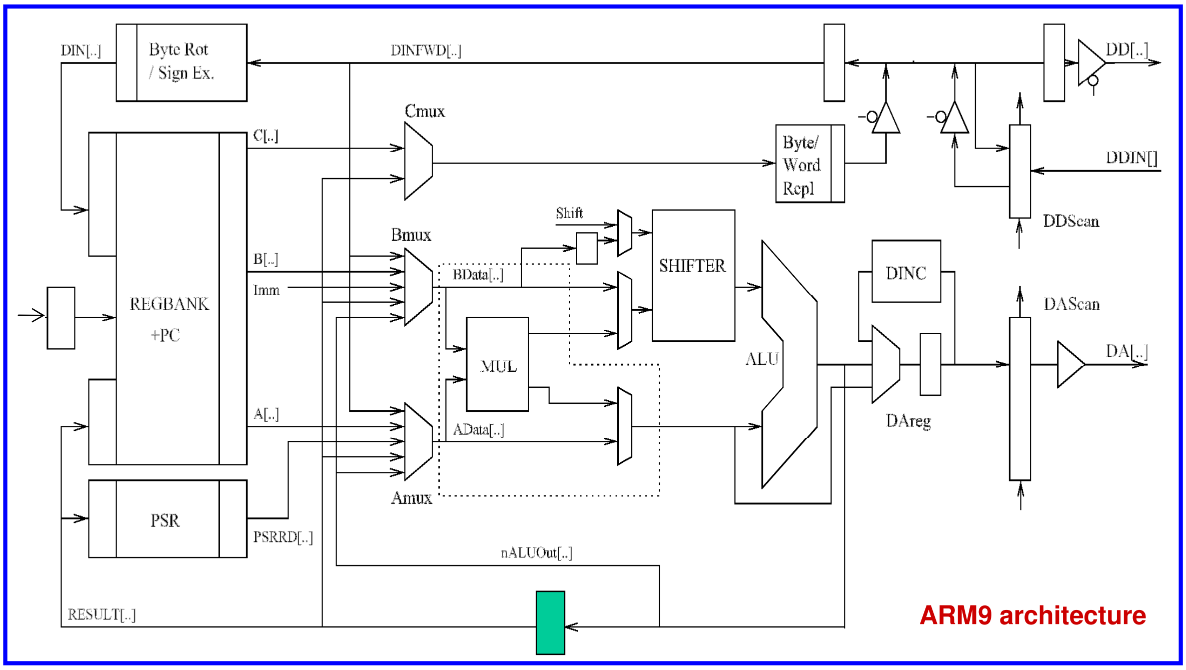
ARM7 pipeline– three stages



ARM9 pipeline – five stages

- In **ARM9**, the 'register read' phase is moved to the decode stage.
- The execution stage is split to 3 stages (one for arithmetic operations, a second to perform memory accesses that remains idle in case of data processing instructions, and a third to write the result to the register file).
- These changes result to a more balanced pipeline (in terms of time), and allow a faster clock signal.
- The data accesses do not have to compete with instruction fetches due to the fact that they use different ports to/from the memory.
- In addition, a "result forwarding" technique is implemented, so the results from the ALU can be feedback immediately to be used by the following instructions. This avoid us not to wait for results to be written back to the register bank and read from the register bank.

ARM9 architecture (cont'd)

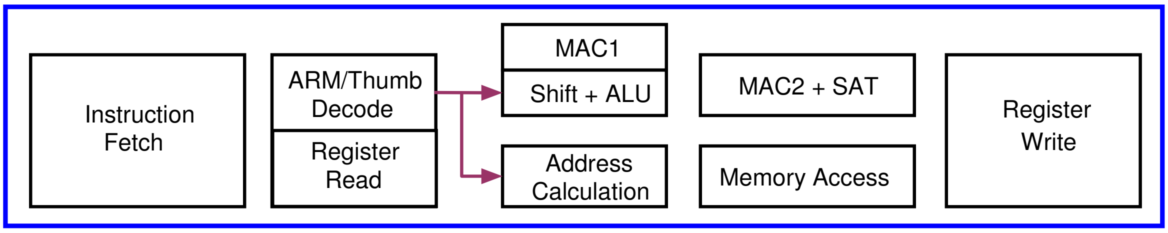


ARM9 architecture

Benchmark programs show that ARM9 needs about 80% of the ARM7 cycles

ARM9E architecture

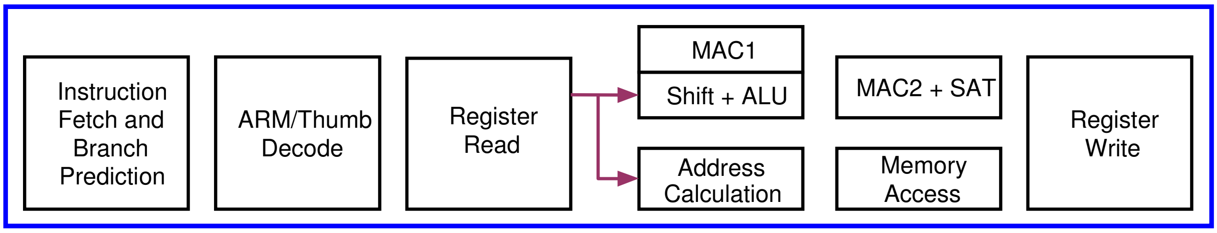
- As ARM9, the ARM9E processor implements a 5-stage pipeline and uses two ports to/from the memory.
- Introduces a separate adder for multiply-accumulate instructions instead of using the main ALU to do the final addition. This helps the execution of instructions of the ARM version's 5 instruction set, enabling DSP enhancements.
- Saturation arithmetic operations that round the multiplication results to the closest fixed-point number, are supported to minimize the calculation error in DSP algorithms.
- In order to speed-up the memory access stage, another adder dedicated to address computation is introduced.



ARM9E pipeline

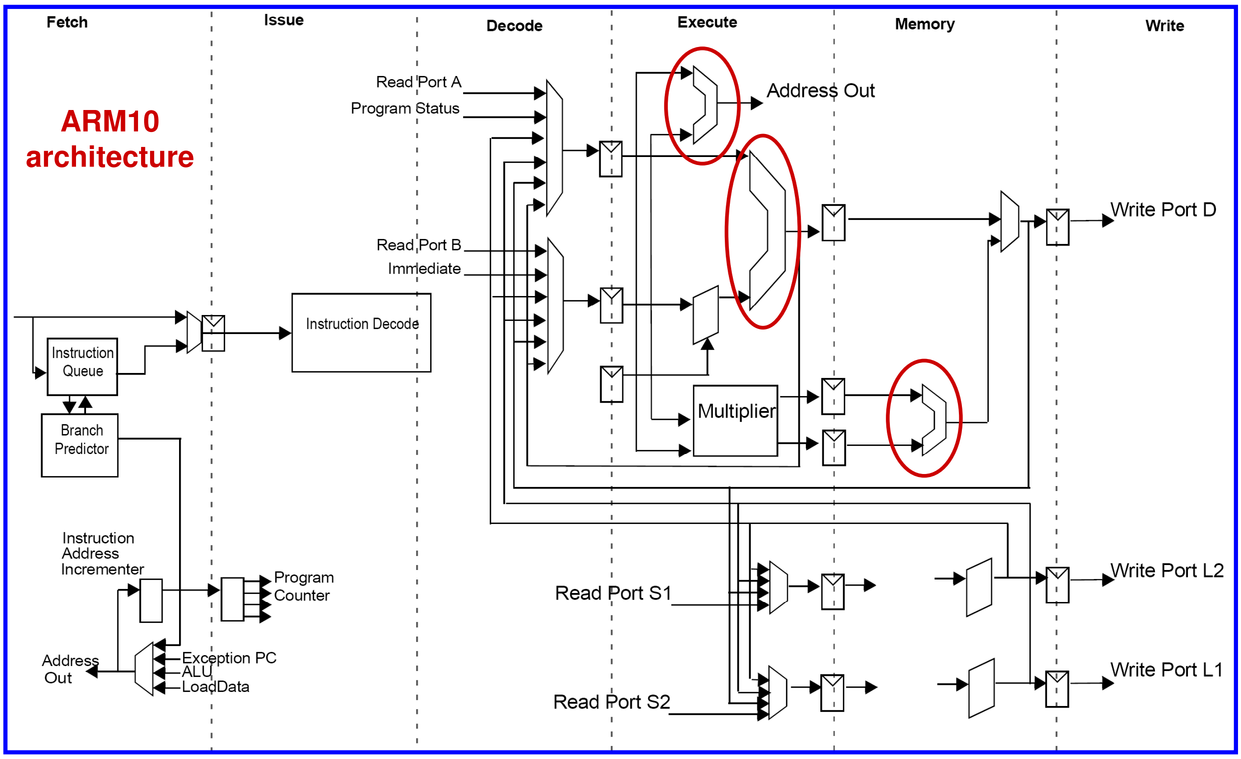
ARM10 architecture

- Processors of ARM10 family have a six-stage pipeline, and has 64-bit instruction and data buses (two instructions can be fetched on each cycle).
- This enables the introduction of static branch prediction, by using the heuristic "backward-taken/forward-not-taken" in order to eliminate the penalty of branches for loops that execute many times.
- As in ARM9E, the data processing part of the pipeline is decoupled from the memory access pipeline (two more adders are used again, one to calculate the address and a second to support the multiplier).
- The last improvement introduced in ARM10 pipeline is the separation of instruction decoding into a single stage.



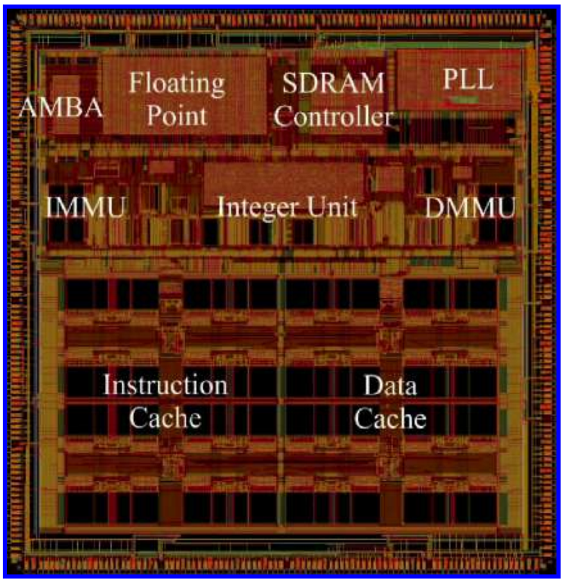
ARM10 pipeline

ARM10 architecture (cont’d)



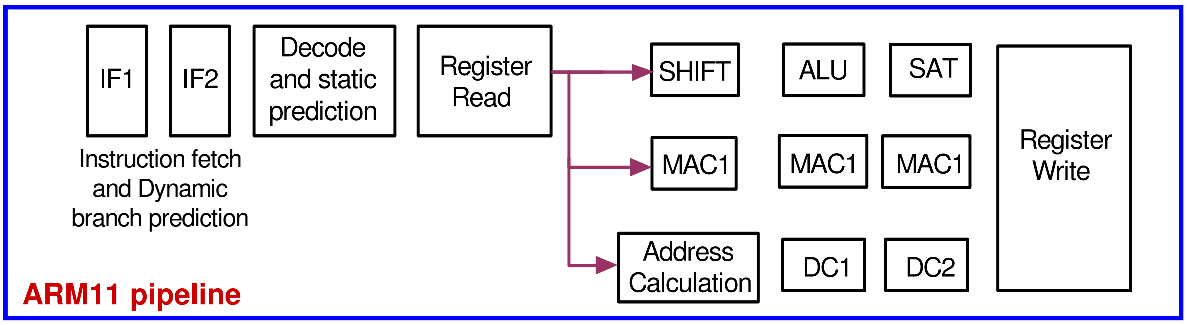
ARM10 architecture (cont’d)

ARM1020E processor with cache memory support
(Includes 32KB instruction and data cache memories, as well as a floating-point co-processor)



ARM11 architecture

- ARM11 uses eight pipeline stages and introduces two main changes to the pipeline: the shift operation is separated into a single stage, and both instruction and data cache accesses are distributed across two pipeline stages.
- Dynamic branch prediction is introduced for the first time through a branch target base (BTB) containing statistics for the branches.
- If the BTB does not contain information for a given branch, the branch is predicted in the decode stage using a static heuristic as in ARM10.
- Decoupling of ALU, MAC and load/store pipelines in order to improve the performance.
- Enhancements in the instruction set (version 6) to support extra DSP operations.



ARM11 pipeline

ARM11 exhibits 40% performance improvements compared to previous versions

Conclusions

- The basic ARM processor (ARM7) was fully analyzed:
  - ✓ Operation, architecture, units, registers.
  - ✓ Instructions (way of execution, decoding etc.).
  - ✓ Compressed instruction set (thumb).
- ARM are the most widely used 32-bit embedded RISC processors. Optimized for cost and low-power applications, provide the low power consumption, small size, and high performance needed in portable, embedded applications.
- ARM provides processor cores with cache memory support.
- The evolution of the ARM architecture was also presented (ARM9, ARM9E, ARM10 and ARM11).

6a. Application-specific instruction-set processors (ASIPs) design

University of Thessaly

Department of Computer and Communication Engineering

ASIPs in embedded systems

- Basic challenge in embedded systems design:

  How to effectively design first-time-right complex systems that meet multiple design constraints?

- To do that it is important to maximize the flexibility/programmability of the target system architecture moving as much functionality as possible to embedded software.

- General purpose processors may not be able to deliver the performance (speed, energy consumption) required by the application, and they may be prohibitively expensive or inefficient.

- Thus, the embedded systems industry has already shown an increasing interest in Application-Specific-Set Processors (ASIPs).

What are ASIPs ?

- ASIPs are processors designed for a particular application or family of applications.
- An ASIP is designed to exploit special characteristics of the target application in order to meet performance, cost and energy requirements.
- These processors are smaller and simpler than their general-purpose counterparts, able to run at higher clock frequencies and with more energy efficiency.
- Obtaining best results requires proper decisions at the instruction set, architectural and the memory organization levels.
- They are considered as a balance between two extremes: ASICs and general-purpose processors. Since an ASIC is designed for a specific behavior, it is difficult to make any changes at a later stage. ASIPs offer the required flexibility at lower cost than general purpose processors.

Benefits of ASIP solutions

- Maintain a level of flexibility and programmability through their own instruction set, and perform the required (specific) computations in fewer cycles.
- Overcome the problems of conventional processors:
  - ✓ Fixed-level of parallelism which may prove inefficient for real-time applications of high computational complexity.
  - ✓ High energy consumption.
  - ✓ Time-critical tasks that require the incorporation of dedicated hardware modules.
- More customized than general purpose processors or DSPs which are just for simple control and specific DSP operations. They provide the required computations without unnecessary generality, in a higher speed and by using smaller silicon area.

Benefits of ASIP solutions (cont'd)

- Performance

  ASICs high

  ASIPs

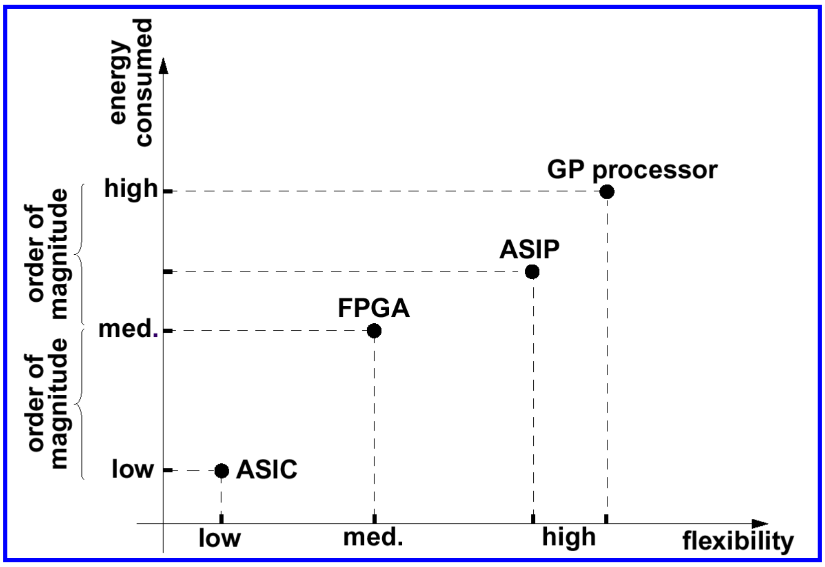
  GP processors low

- Flexibility

  GP processors high

  ASIPs

  ASICs low



What makes an ASIP specific ?

What can we specialize in a processor ?

1. Instruction-set (IS) specialization:

- Exclude instructions which are not used:
  - Reduces instruction word length (fewer bits needed for encoding).
  - Keeps control unit and data path simple.

- Introduce new instructions, which are specific to the application: combinations of arithmetic instructions (e.g. multiply-accumulate), small algorithms (e.g. encoding/decoding, filters) etc.
  - This reduces code size, and leads to reduced memory size, power consumption and execution time.

What makes an ASIP specific ? (cont'd)

2. Functional units and datapath specialization:

Once an application specific IS is defined, this IS can be implemented by using a specific data path and specific functional units.

- Adaptation of word length.
- Adaptation of registers number.
- Adaptation of functional units (specialized units for arithmetic operations, and complex units to perform certain sequences of computations).

What makes an ASIP specific ? (cont'd)

3. Memory specialization:

- Number and size of memory banks.
- Number and size of access ports.
  - They both influence the degree of parallelism in memory access.
  - Having several smaller memory blocks (instead of one big) increases speed, and reduces energy consumption.
  - However, sophisticated memory structures may increase cost.
- Cache configuration:
  - separate instruction/data?
  - associativity
  - cache size
  - line size

Depends very much on the characteristics of the application.

Large impact on performance and energy consumption.

What makes an ASIP specific ? (cont'd)

4. Interconnect specialization:

- Interconnect of functional modules and registers.
- Interconnect to memory and cache.
  - How many internal buses ?
  - What kind of protocol ?
  - Additional connections increase the potential of parallelism.

5. Control specialization:

- Centralised control or distributed ?
- Pipelining ?
- Hardwired or microprogrammed ?

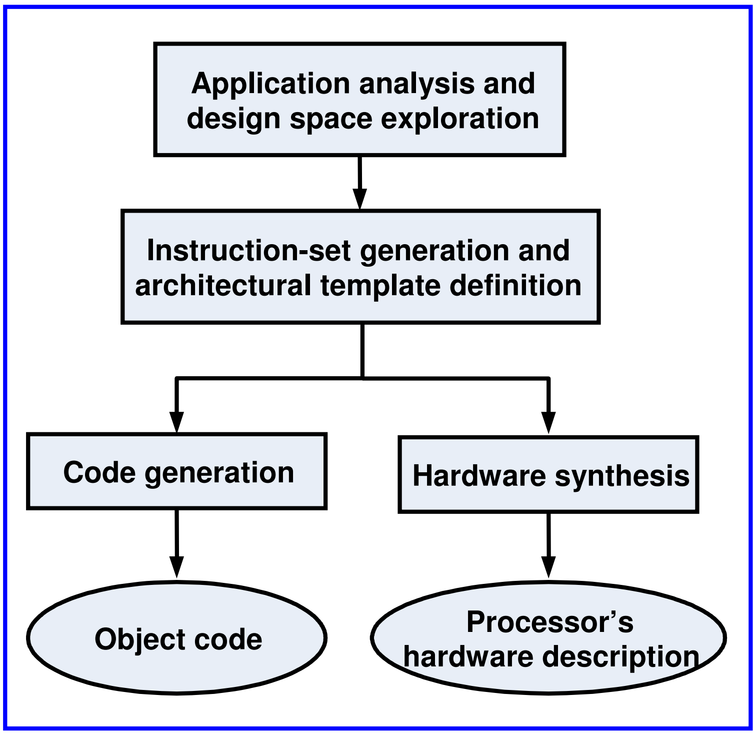
Basic requirements for ASIP design

- Given an application or set of applications, the design of an instruction set and architecture requires the following things in order to meet the design constraints of area, performance and energy consumption:
  - ✓ Definition of the design space of both instruction set and architectures to be explored.
  - ✓ Understanding of the relationships between application characteristics and design (mapping of application on the defined architecture).
  - ✓ Development of tools (estimator, compiler, assembler or code generator, debugger) for application analysis, efficient design space exploration, processor architecture selection and code generation.

Main steps for ASIP design

- Application analysis and design space exploration.
- Instruction set generation: generation of an instruction set that optimizes several parameters: instruction set size, cycles count, cycle time, required gates count etc.
- Architectural template definition: define number and type of functional units, storage elements, interconnect resources, pipelining and parallelism etc.
- Code generation: systematic mapping of the application into assembly code with the generated instruction set.
- Architecture synthesis: this procedure synthesizes the processor architecture (generates processor's hardware description), which implements the generated instruction set.

ASIP design flow



Application analysis and design space exploration

Instruction-set generation and architectural template definition

Code generation

Hardware synthesis

Object code

Processor's hardware description

Application analysis

- Input in the ASIP design process is an application or a family of applications.
- The target application(s) along with its test data and design constraints are analyzed.
- It is essential to analyze the application in order to get the desired characteristics and requirements, which can guide the hardware synthesis as well as the instruction set generation.
- The original application description is translated in a high-level language and profiling is applied in order to get the desired information.
- Major task is to extract the proper requirements (micro-operations), which will lead the instruction set and micro-architecture design procedure.

Application analysis (cont'd)

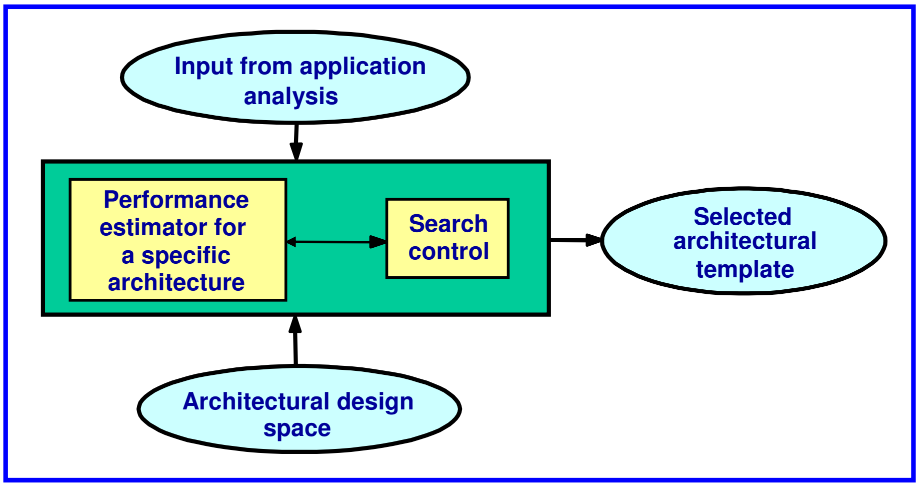
- Application analysis and profiling output may includes:
  - ✓ Number of operations and functions.
  - ✓ Frequency of individual instructions and sequence of contiguous instructions.
  - ✓ Average basic block sizes.
  - ✓ Data types and their access methods.
  - ✓ Ratio of address computation instructions to data computation instructions.
- Application analysis serves as a guide for the subsequent steps in ASIP synthesis.

Architectural design space exploration

- It is important to decide about a processor architecture suitable for target application.
- Determination of a set of architecture candidates for specific application(s) given the design constraints.
- Estimation of performance, hardware cost and energy consumption, and selection of the optimum architecture, are included in this phase of the ASIP design.
- Performance estimation can be based on a simulator with energy consumption estimation capabilities.

Architectural design space exploration (cont'd)

Typical architecture exploration environment


Input from application analysis
Performance estimator for a specific architecture
Search control
Selected architectural template
Architectural design space

The selection process typically can be viewed to consist of a search technique over the design space driven by a performance estimator. Such a simulator is configured for a particular architecture and produce the corresponding performance and energy consumption estimates.

Architectural template definition

- Assume for example that the target application belongs in the multimedia field. Thus it requires high performance and low energy consumption for efficient manipulation of large amount of data in real time.
- Architectures with different configurations in terms of hardware resource selection and interconnection are explored in order to examine the execution performance and power tradeoffs.
- For this reason a parametric processor model is defined and analysed based on pre-characterized hardware units.
- Different values should can be assigned to the parameters while keeping design constraints into consideration.
- The parameters that meet the design constraints and exhibit the optimum results, are used for the definition of the processor's architectural template.

Architectural template definition (cont'd)

Architectural parameters

- Number and type of functional units
- Storage elements
- Interconnect resources
- Pipelining
- Number of pipeline stages
- Instruction-level parallelism
- Memory hierarchy and addressing support
- Latency of functional units and operations

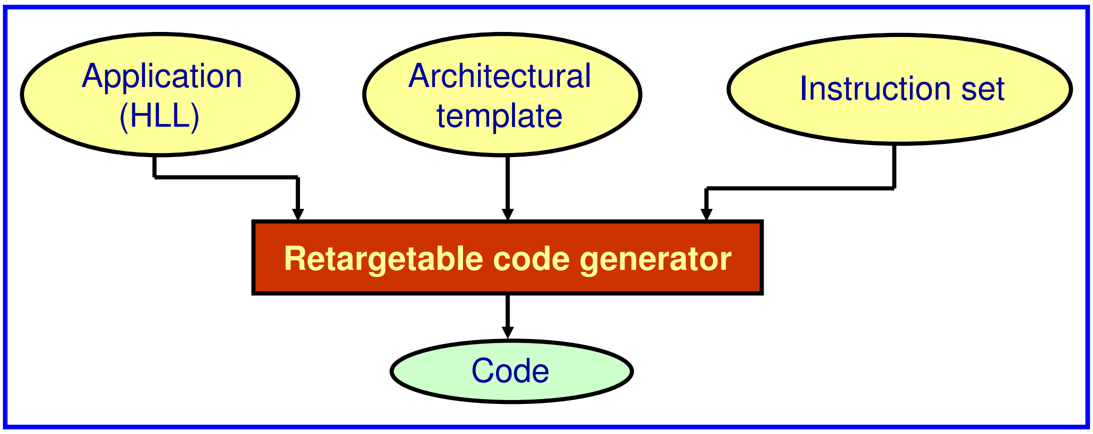
Instruction set generation

- Instruction set is synthesized for a particular application based on the application requirements, quantified in terms of the required micro-operations and their frequencies.

- The instruction set generation process should utilize:
  - ✓ A model for the architecture.
  - ✓ An objective function that establishes a metric for the robustness of the solution (both instruction set and architecture).
  - ✓ Design (hardware resource and timing) constraints.

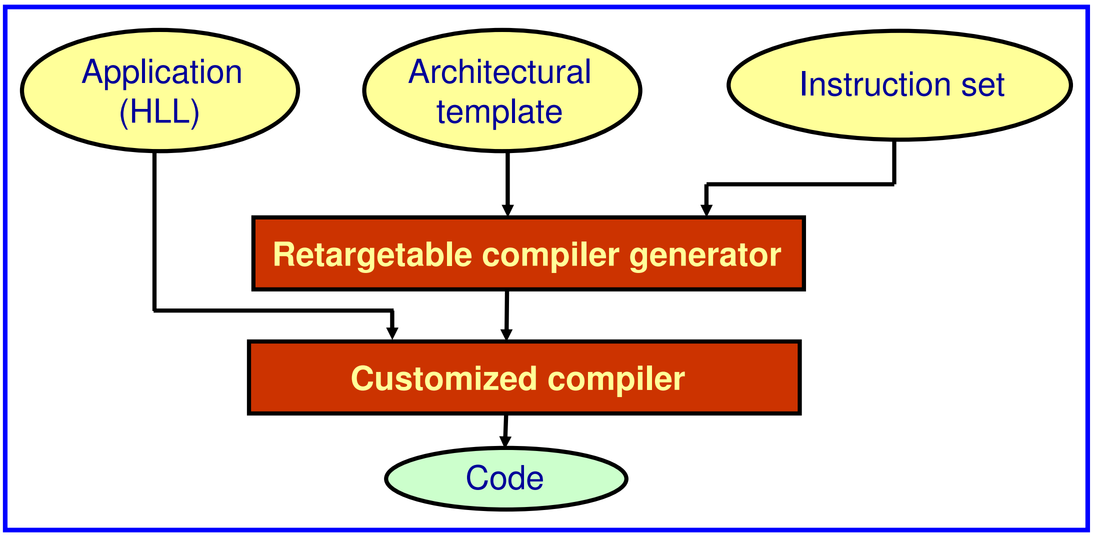
- Instruction selection is based on heuristic rules: starting from a general instruction set that covers all the operations required by the application, modifications are performed (include, exclude or combine instructions) only if the objective function (performance) is improved by a predefined percentage.

Code generation



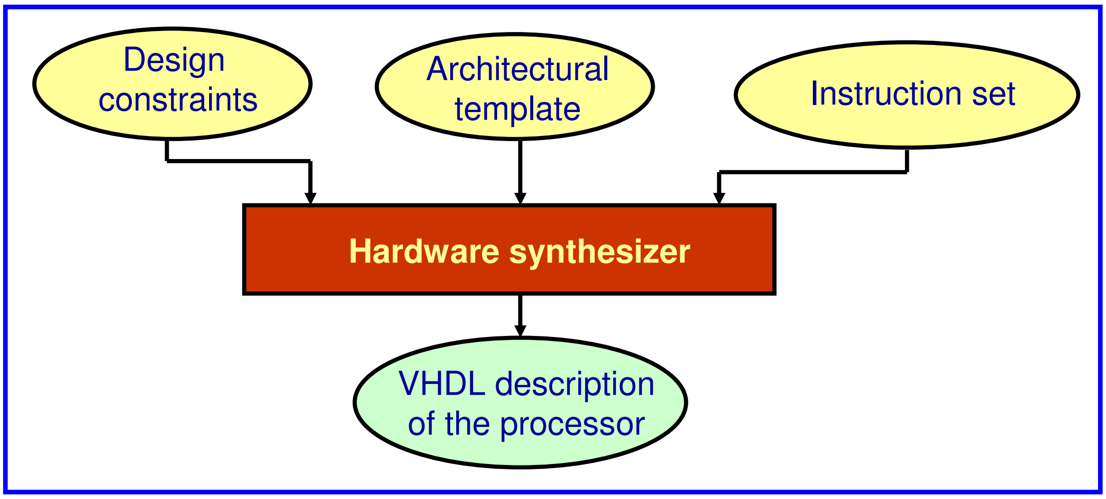
The retargetable code generator takes as inputs the architecture template, the instruction set and the application written in a HLL, in order to generate the object code.

Code generation (cont'd)



The retargetable compiler generator takes as inputs the architecture template and the instruction set and generate a customized compiler which accepts the application programs and produces the object code.

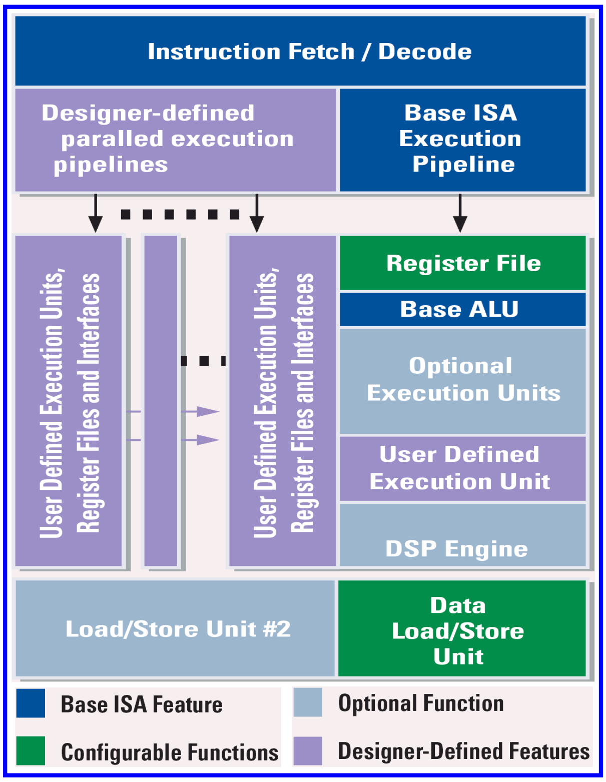
Hardware synthesis



- Hardware synthesis refers to the generation of the HDL descriptions for the processor modules.
- Automatic generation of RTL descriptions, for both simulation and logic synthesis purposes.

ASIP example: Tensilica Xtensa LX2 processor

- Configurable, extensible and synthesizable processor for rapid implementation of complex SoC designs.
- Base architecture: 32-bit ALU, 68 registers, 80 instructions, compressed 16- or 24-bit instruction encoding.
- Selection and configuration of predefined processor functions (configurability).
- Optional predefined execution units: 32-bit multiplier, 16-bit MAC, DSP engine, floating point unit.
- Explorer to analyze the application and find options that will enhance performance.
- Specific language (Verilog-like) to describe new execution units, and processor generator to add them to the processor (extensibility).
- Designer-selectable 5- or 7-stage pipeline: option of adding two cycles for access memories with long access times.


Instruction Fetch / Decode
Designer-defined paralled execution pipelines
Base ISA Execution Pipeline
User Defined Execution Units, Register Files and Interfaces
User Defined Execution Units, Register Files and Interfaces
Register File
Base ALU
Optional Execution Units
User Defined Execution Unit
DSP Engine
Load/Store Unit #2
Data Load/Store Unit
Base ISA Feature
Optional Function
Configurable Functions
Designer-Defined Features

Conclusions

- ASIPs are processors designed for a particular application or family of applications.
- An ASIP is designed to exploit special characteristics of the target application in order to meet performance, cost and energy requirements.
- ASIPs are considered as a balance between two extremes: ASICs and general-purpose processors, and offer the required flexibility at higher speed, lower energy consumption and lower cost than general purpose processors.
- The main steps in a typical methodology for ASIP design are:
  - ✓ Application analysis.
  - ✓ Design space exploration.
  - ✓ Instruction set and architectural template generation.
  - ✓ Code generation.
  - ✓ Hardware synthesis.

6b. Very long instruction word (VLIW) processors

University of Thessaly

Department of Computer and Communication Engineering

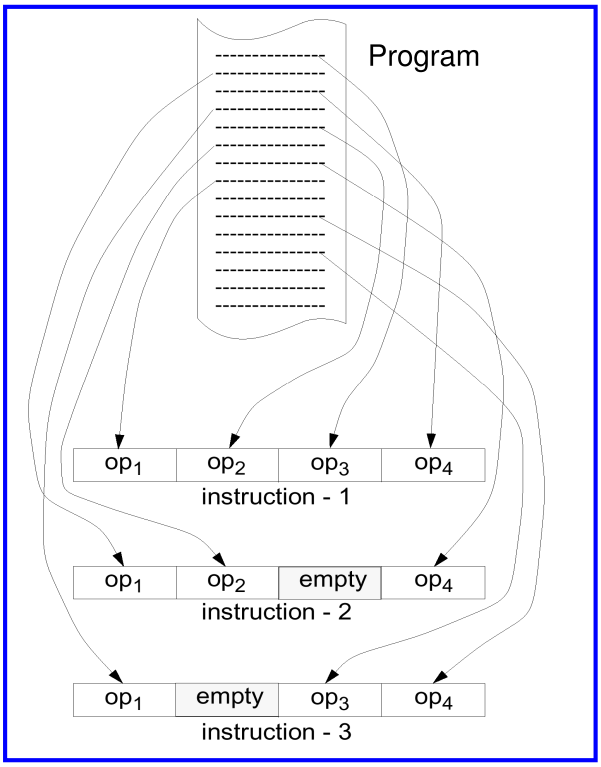
VLIW processors in embedded systems

- The competitors in embedded processor market ranging from vendors implementing variations of traditional embedded processor architectures (mainly RISCs) such as ARM and MIPS to vendors introducing application specific instruction set architectures, such as ARC and Tensilica.
- At the same time, the complexity of embedded applications increases considerably, and it is not uncommon to find many hundreds of lines of HLL code in embedded products (e.g. mobile phones).
- As a result, today there is a new direction to use VLIW processors in embedded systems that can be application specific and offer high performance, in order to perform computation intensive functions.
- This direction is based not only on the increase of the complexity of embedded applications, but also on the high evolution of the IC fabrication technology and compilers' technology.
- VLIW architectures are characterized by instructions that each specifies several independent operations, while RISC instructions typically specify one operation, and CISC instructions typically specify several dependent operations.

What are VLIW processors ?

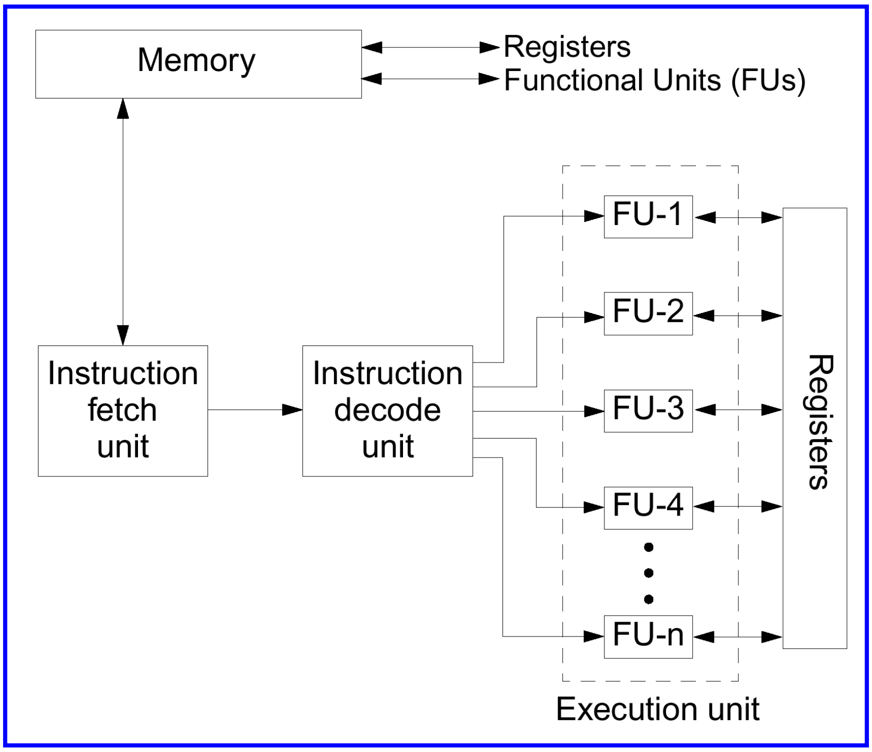
- VLIW processors are high-performance processors, that use multiple execution units and have the ability to execute multiple operations simultaneously.
- VLIW architectures rely on compile-time detection of parallelism: the compiler analyses the program and detects operations that can be executed in parallel, so such operations are packed into one “long” instruction.
- This is in contrast with superscalar processors (traditional parallel computers) in which hardware is responsible to detect parallelism between operations, leading in high complexity of the hardware.
- In VLIW machines, after one instruction has been fetched, all the corresponding (independent) operations are issued in parallel.
- The compiler can analyse the whole program in order to detect parallel operations and no specific hardware is needed for run-time detection of parallelism.

What are VLIW processors ? (cont’d)

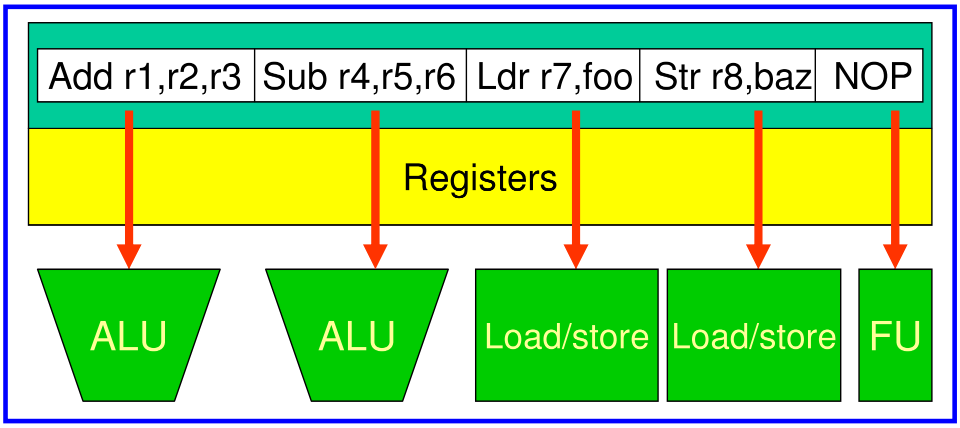


Detection of operations parallelism and their packaging into instructions are performed by the compiler off-line (static scheduling).

Architecture of VLIW processors

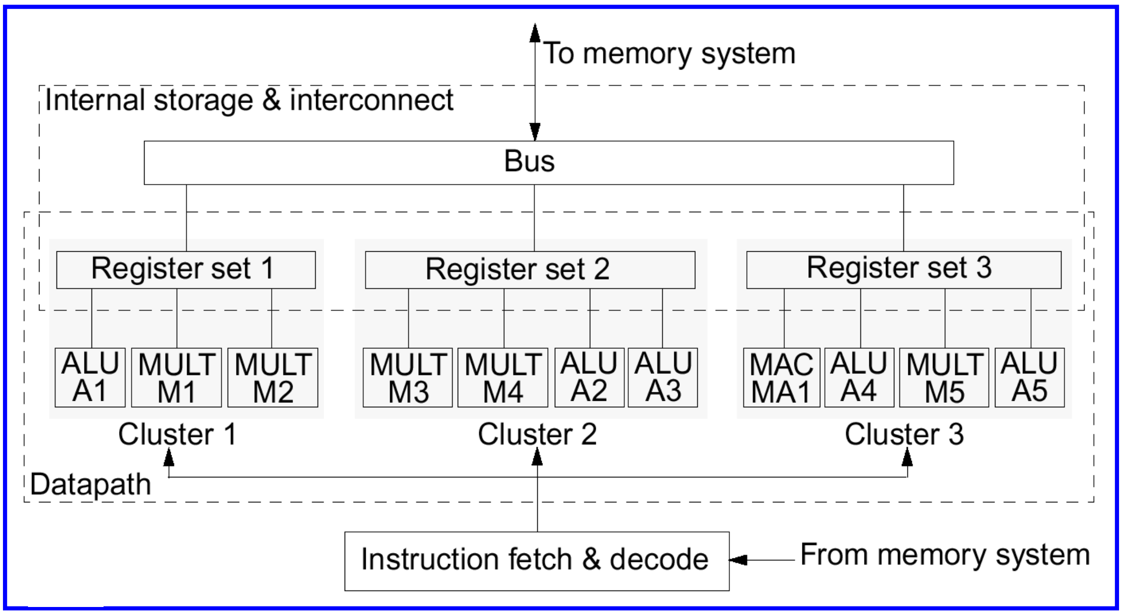


Architecture of VLIW processors (cont'd)



- Traditionally the datapath has a large single register bank shared by all functional units.
- In order to increase parallelism, we have to increase the number of functional units.
- Then, the internal storage and communication between functional units and registers becomes dominant in terms of area, delay and power.
- There is internal bandwidth limitation.

Architecture of VLIW processors (cont'd)

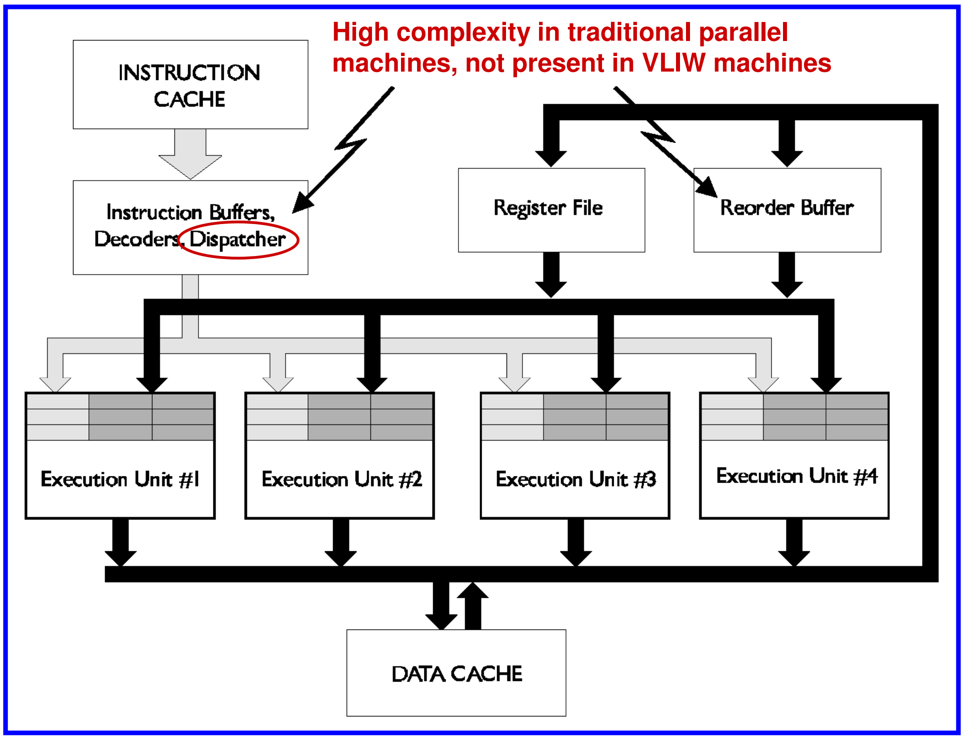


- A solution is to restrict the connectivity between functional units and registers, so that each functional unit can read/write from/to a subset of registers (clustering).

VLIW processors' advantages

- The number of functional units can be increased without needing additional sophisticated hardware to detect parallelism, as in traditional parallel machines (superscalar). So, VLIW machines requires less silicon area, lower cost and smaller design/testing effort and time.
- VLIW require more sophisticated compilers than traditional architectures in order to be able to extract parallelism and to keep the instructions full.
- In such a way the complexity is paid only once (when the compiler is written) instead of every time the chip is fabricated.
- Improvements to the compiler can be made after chips has been fabricated, while improvements in superscalar machines require changes to the hardware, which naturally incurs all the expenses of chip design and fabrication.
- In VLIW machines, the compiler detects parallelism based on a global analysis of the whole program, while in traditional parallel computers there is a need to dynamically examine of large window of execution (set of instructions that is considered for execution at a certain time) in order to know possible data, resource or control dependencies between instructions that affect parallel execution.

VLIW processors’ advantages (cont’d)



High complexity in traditional parallel machines, not present in VLIW machines

Typical superscalar architecture

VLIW processors’ advantages (cont’d)

- The dispatcher examine a window of instructions contained in a buffer, and decides which one can be passed to the execution units for parallel execution.
- More execution units require wider windows and a more complex dispatcher.
- To avoid waiting for conditional branches to be resolved, parallel computers implement branch prediction. With branch prediction, the processor makes an early guess about the outcome of the branch and begins looking for parallelism along the predicted path.
- To be able to undo the effects of a branch execution in the case of a misprediction, a hardware structure called a reorder buffer is employed. This structure keeps track of all the results produced by instructions that have recently been executed or that have been dispatched to execution units but have not yet completed.
- The reorder buffer provides a place for results and when a conditional branch is, in fact, resolved, the results of the executed instructions can be either dropped from the reorder buffer (branch mispredicted) or written from the buffer to the register bank (branch predicted correctly).
- The two above hardware resources are not needed in VLIW machines, due to the presence of sophisticated compilers.

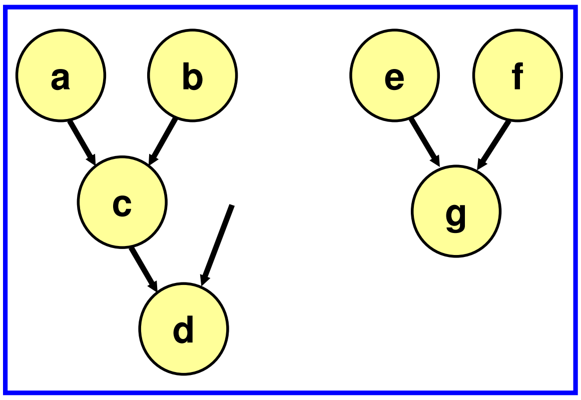
VLIW processors' problems

- Large number of registers needed in order to keep all functional units active (i.e. to store operands and results).
- Large data transport capacity is needed between functional units and registers and between registers and memory.
- High bandwidth between instruction cache and fetch unit. Example: one instruction with 7 operations, each 24 bits leads to 168 bits per instruction.
- Large code size mainly due to unused operations (wasted bits in the instruction word).
- Incomputability of binary code: if additional functional units are introduced, the number of operations possible to execute in parallel is increased, the instruction word changes and the old binary code cannot run on the new version of the processor.
- Sophisticated compilers (and thus more expensive) are needed.

VLIW compilers

- In order to keep all the functional units busy, compilers for VLIW processors have to be aggressive in parallelism detection.
- In general, static scheduling of independent operations and packaging of them into long instructions is followed.
- Loop unrolling is used by VLIW compilers in order to increase the degree of parallelism in loops: several iterations of a loop are unrolled and handled in parallel.
- Trace scheduling is applied to conditional branches: the compiler tries to predict which sequence is the most likely to be selected and schedules operations so that this sequence is executed as fast as possible. Compensation code is added in order to keep the program correct.

Static scheduling of independent operations



Data flow graphs of a program showing data dependencies between operations

a	b	e
f	c	nop
d	g	nop

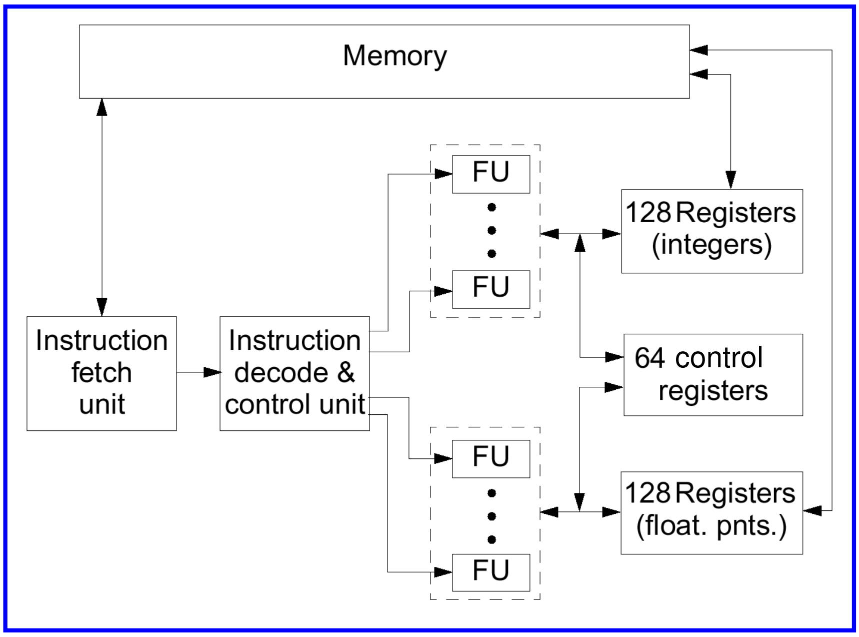
Scheduling and packaging of operations in instructions

Example 1: Itanium processor

- Itanium (by Intel and HP) is not a pure VLIW architecture, but many of its features are typical for VLIW processors.

- Typical VLIW features:
  - ✓ Instruction-level parallelism fixed at compile-time.
  - ✓ Very long instruction word (128 bits).

Itanium processor’s architecture



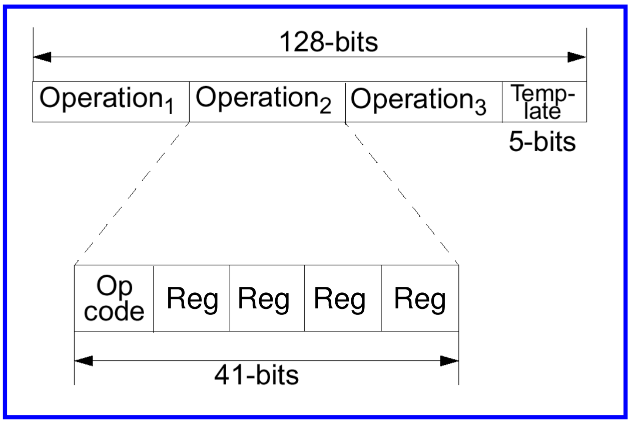
128 integer registers, 128 FP registers and 8 branch registers of 64-bits each.

64 control registers of 1-bit each.

15 functional units:
4 integer ALUs,
4 multimedia ALUs,
2 FP units,
2 load/store units,
3 branch units.

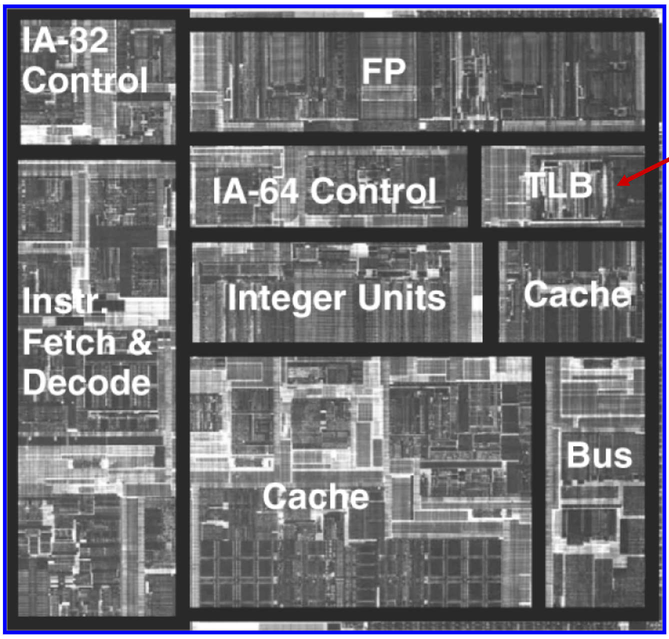
10-stage pipeline.

Itanium processor’s instruction format



- Three operations per instruction word. This does not mean that maximum three operations can be executed in parallel. The three operations in the instruction are not necessarily to be executed in parallel.
- The template indicates what can be executed in parallel. The encoding in the template shows which of the operations in the instruction can be executed in parallel. The template connects also to neighbouring instructions, so operations from different instructions can be executed in parallel.

Itanium processor’s photograph

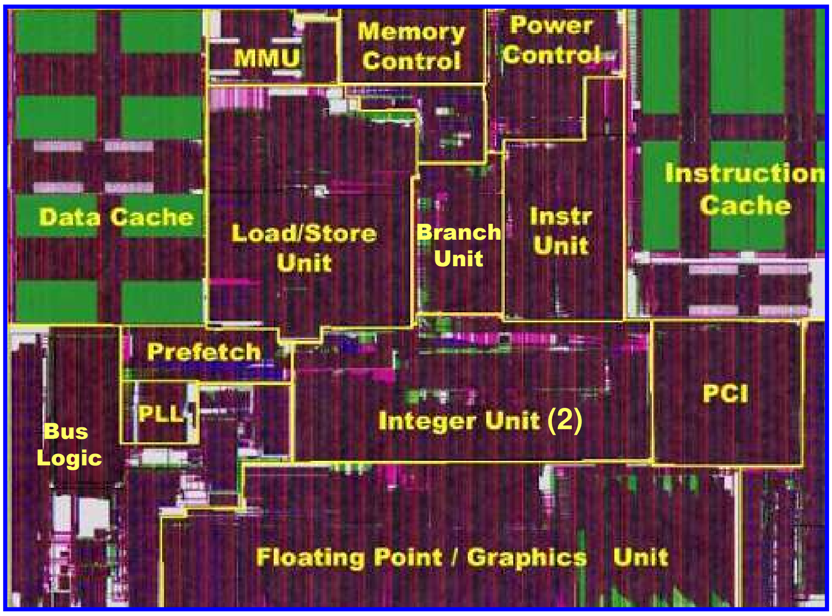


Translation Look-aside Buffer (between I-Cache and processor’s pipeline)

Example 2: Crusoe processor

- Crusoe (by Transmeta) is a pure VLIW architecture.
- Features:
  - ✓ Instruction-level parallelism fixed at compile-time.
  - ✓ Very long instruction word (128 bits): contain up to four RISC-like instructions of 32-bits each.
  - ✓ Two integer units, one floating-point unit, a memory (load/store unit) and a branch unit.
  - ✓ 64 integer and 64 floating-point registers of 128-bits each.
  - ✓ Includes the code morphing software: a dynamic translation tool that compiles instructions of x86 architecture (CISC) into instructions of VLIW (the processor has to dedicate some of its cycles to running the translation).

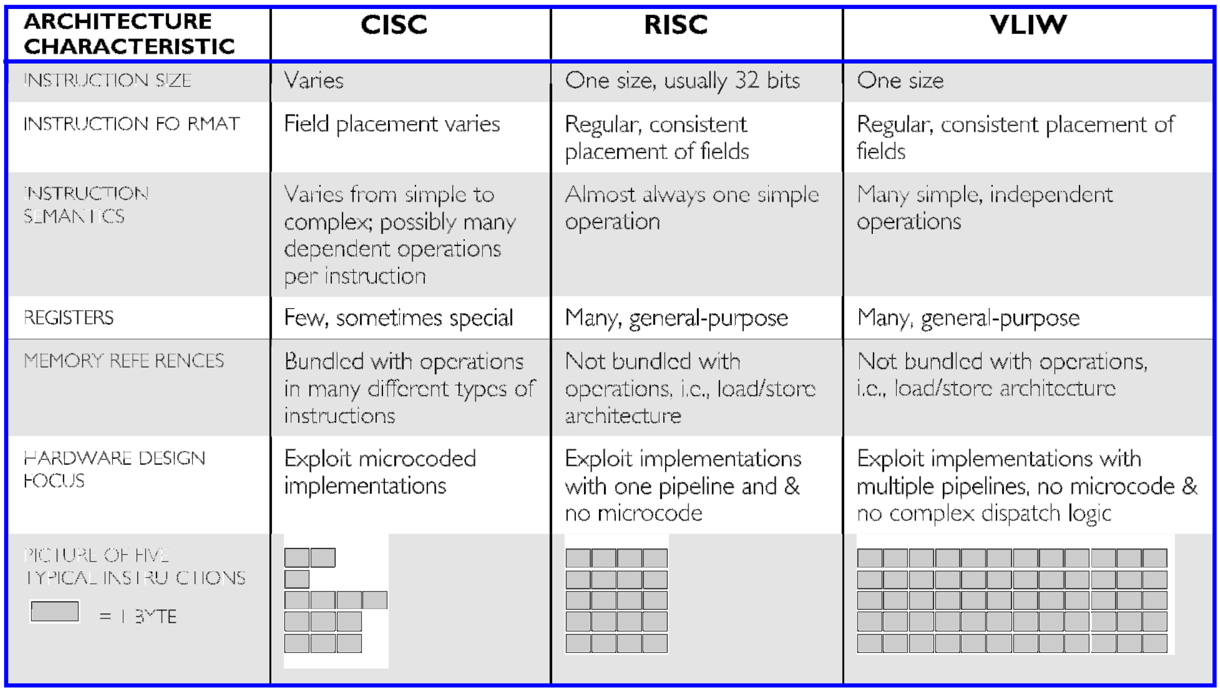
Crusoe processor's photograph



Example 3: C62x processors

- C62x (by Texas Instruments) are VLIW-based DSP processors for multimedia applications.
- Features:
  - ✓ Instruction-level parallelism fixed at compile-time.
  - ✓ Very long instruction word (256 bits): contain up to eight 32-bits instructions.
  - ✓ Eight independent functional units: six ALUs and two multipliers.
  - ✓ Uses load-store architecture.
  - ✓ 64 general purpose registers of 32-bits each.
  - ✓ Contains two execution clusters with 32 registers each.
  - ✓ Implements an 11-stages pipeline and all instructions are conditional.

VLIW vs. CISC vs. RISC



ARCHITECTURE CHARACTERISTIC	CISC	RISC	VLIW
INSTRUCTION SIZE	Varies	One size, usually 32 bits	One size
INSTRUCTION FORMAT	Field placement varies	Regular, consistent placement of fields	Regular, consistent placement of fields
INSTRUCTION SEMANTICS	Varies from simple to complex; possibly many dependent operations per instruction	Almost always one simple operation	Many simple, independent operations
REGISTERS	Few, sometimes special	Many, general-purpose	Many, general-purpose
MEMORY REFERENCES	Bundled with operations in many different types of instructions	Not bundled with operations, i.e., load/store architecture	Not bundled with operations, i.e., load/store architecture
HARDWARE DESIGN FOCUS	Exploit microcoded implementations	Exploit implementations with one pipeline and & no microcode	Exploit implementations with multiple pipelines, no microcode & no complex dispatch logic
PICTURE OF FIVE TYPICAL INSTRUCTIONS ▭ = 1 BYTE	▭▭ ▭ ▭▭▭▭ ▭▭▭ ▭▭▭	▭▭▭▭ ▭▭▭▭ ▭▭▭▭ ▭▭▭▭ ▭▭▭▭	▭▭▭▭▭▭▭▭▭▭▭▭ ▭▭▭▭▭▭▭▭▭▭▭▭ ▭▭▭▭▭▭▭▭▭▭▭▭ ▭▭▭▭▭▭▭▭▭▭▭▭ ▭▭▭▭▭▭▭▭▭▭▭▭

Conclusions

- VLIW architectures avoid hardware complexity (that exists in traditional parallel architectures) by relying exclusively on the compiler for parallelism detection.
- Not having to deal with parallelism detection, VLIW processors can have a high number of functional units. This, however, generates the need for a large number of registers and high communication bandwidth.
- In order to keep the large number of functional units busy, compilers for VLIW processors have to be aggressive in parallelism detection.
- Static instruction scheduling, loop unrolling and trace scheduling are used by compilers in order to increase the degree of parallelism in application programs.
- Several modern processors (Itanium, Crusoe, C62x) have typical VLIW features such as instruction-level parallelism at compile-time and long instruction words.

7a. System-on-chip design and prototyping platforms

University of Thessaly

Department of Computer and Communication Engineering

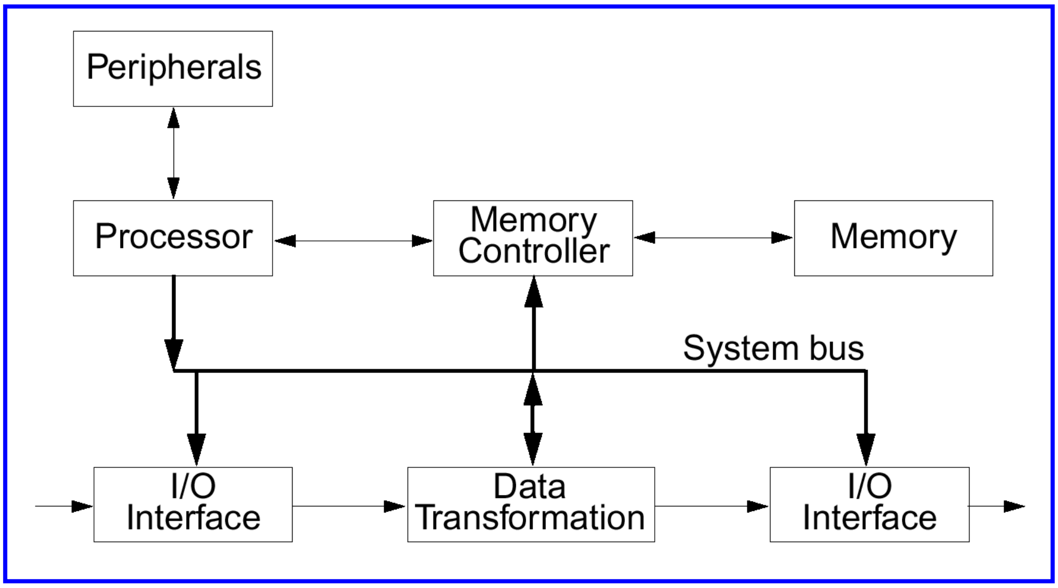
What is System-on-Chip (SoC) ?

- System-on-chip is an integrated circuit that implements most or all of the functions of a complete electronic system, which solves an embedded application.

- It is a heterogeneous system: may include hardware and software parts, control and data-processing functionality, digital and analog parts etc.

- Contains more than a single processor: memory modules, custom circuitry, I/O peripherals, A/D or D/A converters etc.

A typical SoC structure



Basic components of a SoC

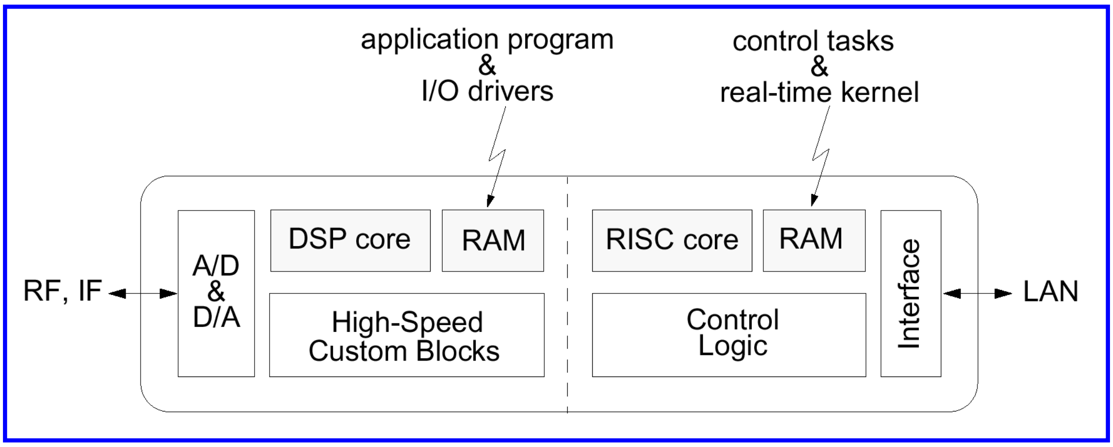
- A microprocessor and its memory subsystem:
  - ✓ Could be an 8-bit microcontroller core up to 64-bit RISC.
  - ✓ The memory subsystem can be single or multi-levelled.
- A datapath with interfaces to the external system:
  - ✓ The external interfaces can be bus drivers, Ethernet interfaces, A/D – D/A converters, electro-mechanical converters etc.
- Blocks performing transformations on data received from the external systems:
  - ✓ Could be implemented by custom hardware and/or by DSP cores.
- Interface logic to peripherals.

Basic components of a SoC (cont'd)

- Programmable processor cores:
  - ✓ Algorithms and protocols become increasingly complex, and this makes their implementation in hardware difficult
  - ✓ Modern processors are fast as a result of their sophisticated design.
  - ✓ Upgrading the software implementation is easy (flexibility).
- Custom hardware is still quite useful:
  - ✓ High performance for time-critical task.
  - ✓ Low energy consumption.

Typical SoC example

SoC structure for the implementation of a typical wireless telecommunication application.



Why do we need System-on-Chips ?

FPGA
From PCB to SoC

- System-on-chips now are technologically possible: today's chips can contain up to 100 million transistors (according to the Moores Law, approximately every 18 months the number of transistors on a single chip doubles).
- Higher performance: fast data transfer compared to board multi-chip designs.
- Lower energy consumption: multi-chip designs need additional drivers and interfaces for inter-chip and inter-board connections.
- Reduced size: components connected on a PCB can now be integrated onto a single chip.
- Lower cost.
- Increased reliability and design security.

Problems in SoC design

- Increased system complexity mainly due to the heterogeneity (analog along with digital parts, processors along with custom hardware, different memory types etc.), and due to the integrated nature.
- This creates problems to the design phase (expertise in different design areas is needed and the integration is a quite demanding task), and to the technology (several processes have to be incorporated onto a single die).
- Traditional hardware design methodologies does not work.
- Increased verification requirements.
- Design productivity vs. time-to-market pressure.
- Solutions to overcome complexity, low design productivity and time-to-market pressure is to use advanced SoC design methodologies and tools and mainly to re-use IP (Intellectual Property) blocks in our SoC design.

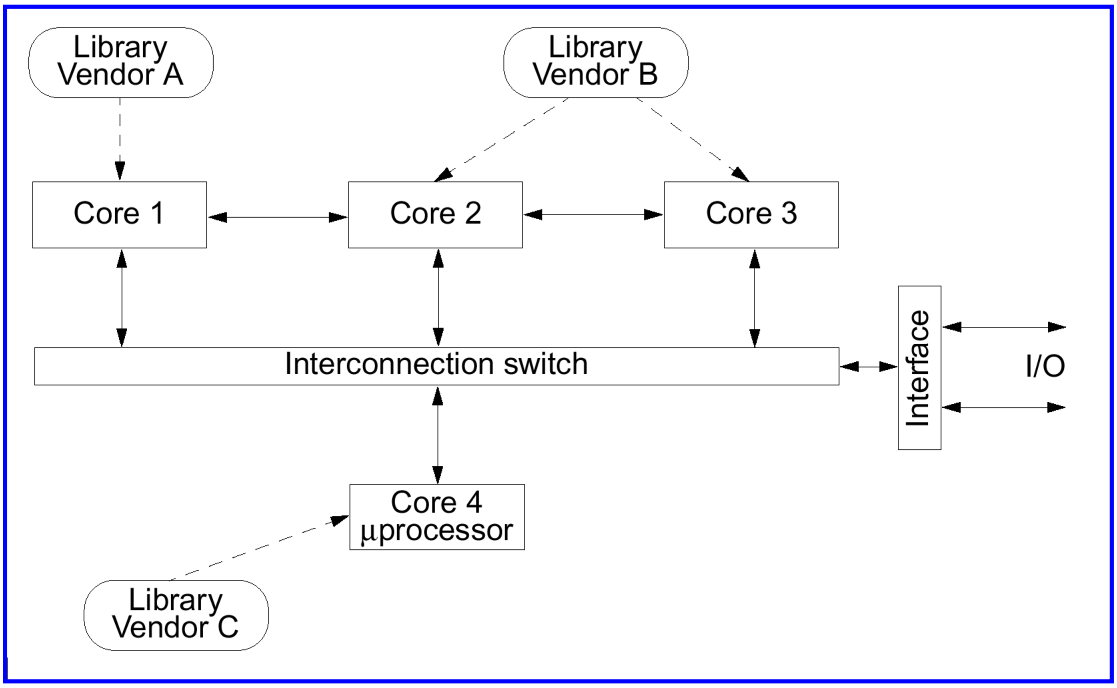
IP-based SoC design

- IP-based design is the process of composing a new system by reusing existing components.

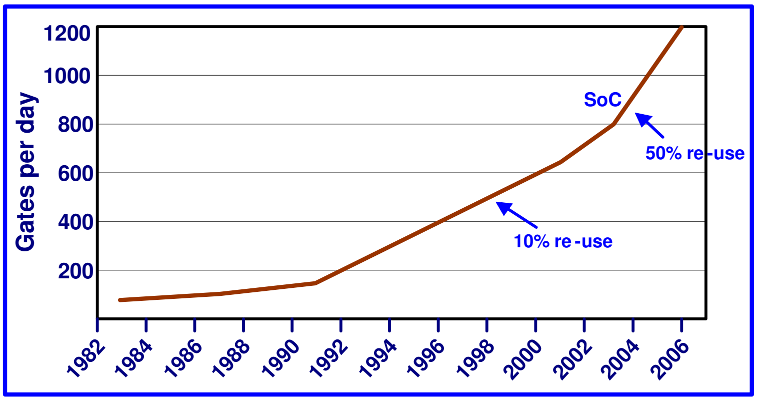
- Possible IP blocks to be used:
  - ✓ Microprocessors (ARM, MIPS, PowerPC, SPARC etc.)
  - ✓ Interfaces (USB, PCI, UART etc.)
  - ✓ Encoder and decoders (JPEG, MPEG, Viterbi etc.)
  - ✓ Memories (SRAM, Flash etc.)
  - ✓ Microcontrollers (HC11 etc.)
  - ✓ DSPs (TI, Oak etc.)
  - ✓ Transformers (FFT, IFFT etc.)
  - ✓ Networking blocks (Ethernet, ATM etc.)
  - ✓ Encryption blocks (DES, AES etc.).

- The increasing need of SoCs is forcing design houses and vendors to develop high-quality IP blocks (a new industry has been developed !).

IP-based SoC design (cont'd)



SoC design productivity and cost



Cost of 50M transistor SoC	
• Gates:	12.5 M
• Productivity (gates/day):	1200
• Total engineer months:	~ 520
• Engineer cost per month:	~ 8.5 K€
• Total personnel cost:	~ 4.5 M€
• Additional NRE (masks, CAD):	~ 4 M€
• Total cost:	~ 8.5 M€

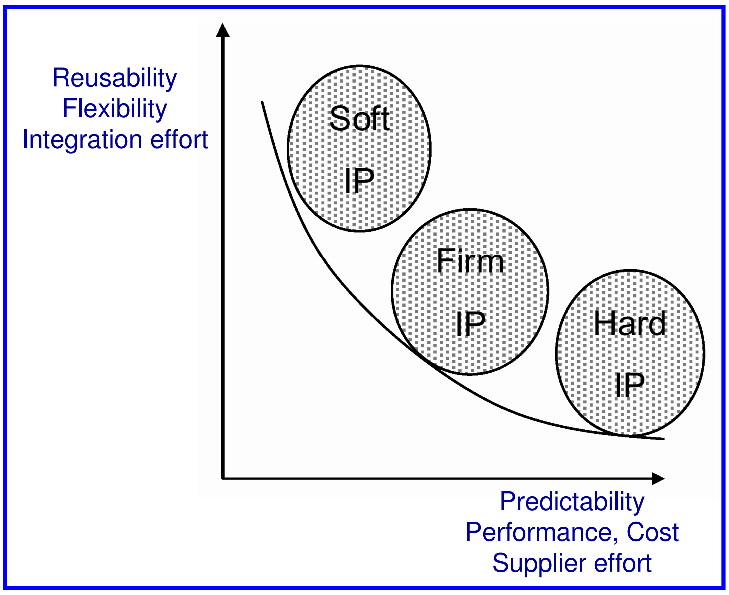
Main issues in IP blocks design

- How to specify an IP block for a reuse library: functionality, timing information, interface properties, achieved speed, power consumption etc.

- Much effort has to be allocated for:
  - ✓ Specification, simulation, estimation and exploration.
  - ✓ Integration (interfaces definition and implementation).
  - ✓ Verification and testing for many operating conditions and inputs.

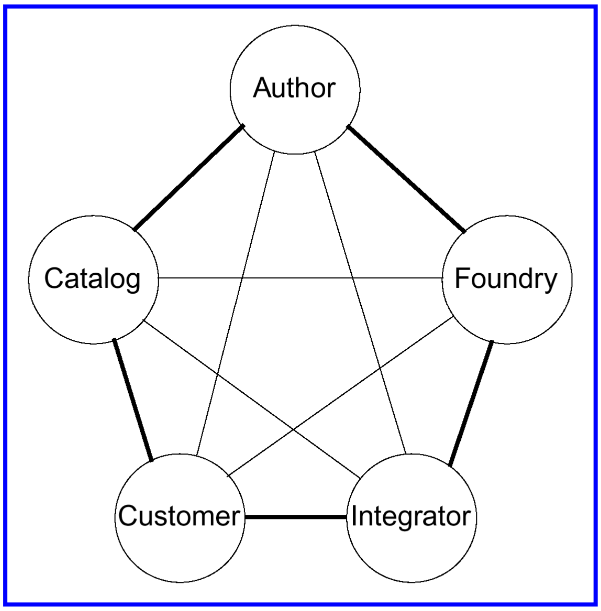
Types of IP blocks

- Hard IP blocks:
  Fully designed, placed and routed by the supplier. It is offered as a completely validated layout with definite timing characteristics and offers fast integration but low flexibility.
- Firm IP blocks:
  Technology-mapped gate-level netlist and offers flexibility during place and route, but with lower predictability.
- Soft IP blocks:
  Synthesizable RTL or behavioral descriptions. Require much effort for integration/verification, but offers maximal flexibility.

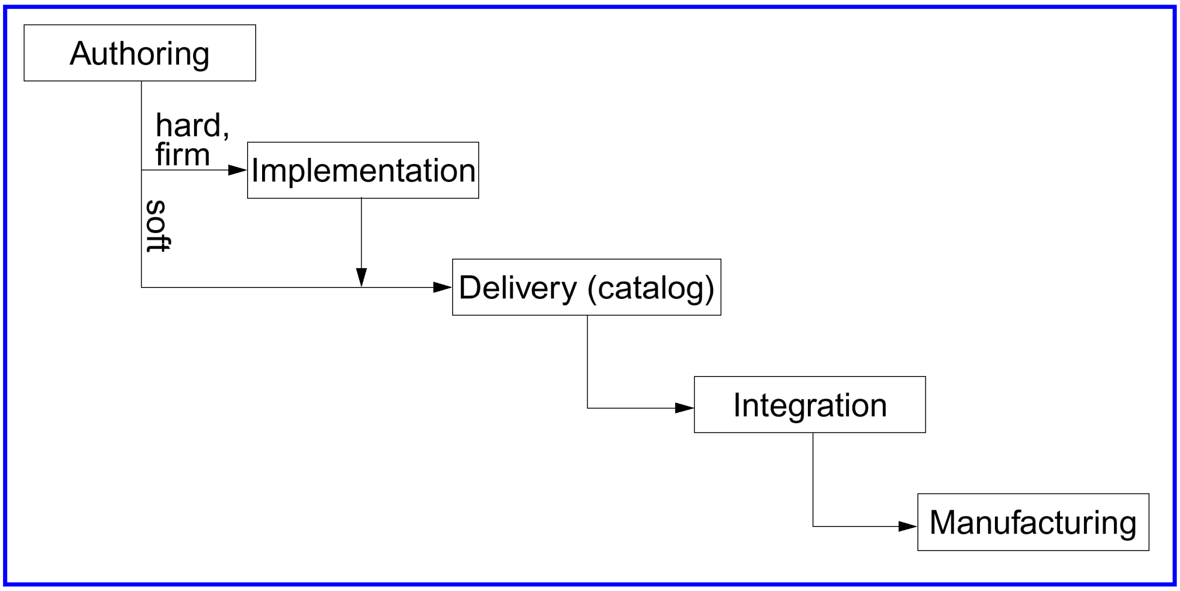


Players in IP blocks design

- Author: creator of the IP block
- Foundry: manufacturer
- Catalog: enables customers to find and select the IP bock.
- Integrator: designs the SoC for the customer.



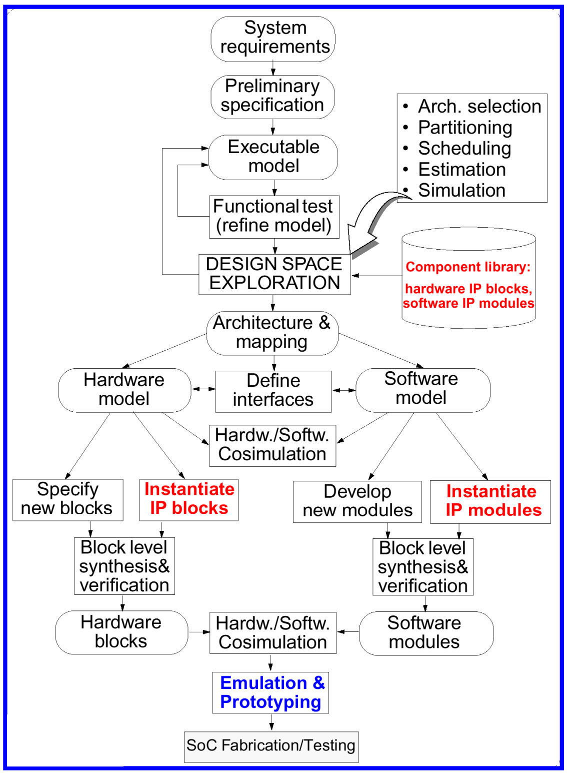
IP block life cycle



IP block life cycle (cont'd)

- Authoring: Find the idea, create a model, perform high-level validation, create simulation models to be delivered with the IP block.
- Implementation: create hard of firm IP blocks.
- Delivery: Extract design characteristics (functionality, interfacing, timing, power consumption, technology-related aspects, compatibility aspects etc.) and put the block in an IP catalog containing adequate search mechanisms.
- Integration: instantiate the block, build interfaces (interface logic synthesis), verify (by simulation).
- Manufacturing: Fabrication and testing of the integrated circuit composed by the implemented IP blocks.
- Note, that IP-based design is also a solution for improving software design productivity. Reuse techniques and libraries of predefined software modules (classes, functions) are not new in software.

SoC design flow



System requirements

Preliminary specification

Executable model

Functional test (refine model)

DESIGN SPACE EXPLORATION

- Arch. selection
- Partitioning
- Scheduling
- Estimation
- Simulation

Component library: hardware IP blocks, software IP modules

Architecture & mapping

Hardware model

Define interfaces

Software model

Hardw./Softw. Cosimulation

Specify new blocks

Instantiate IP blocks

Develop new modules

Instantiate IP modules

Block level synthesis& verification

Block level synthesis& verification

Hardware blocks

Hardw./Softw. Cosimulation

Software modules

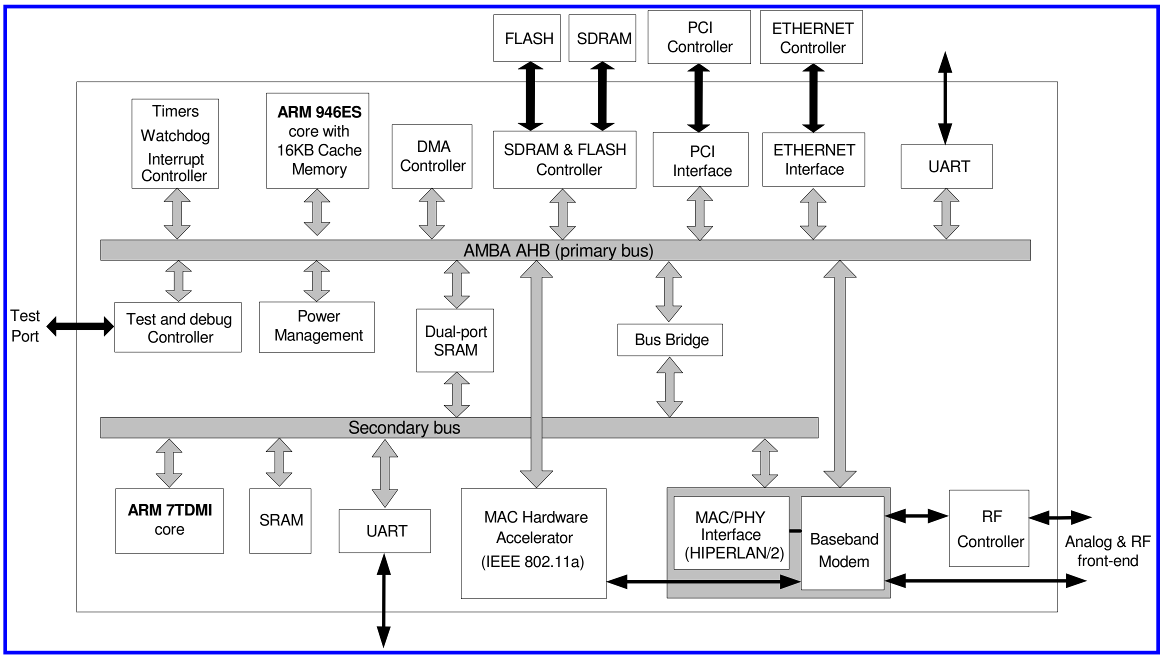
Emulation & Prototyping

SoC Fabrication/Testing

Example: Wireless LAN SoC

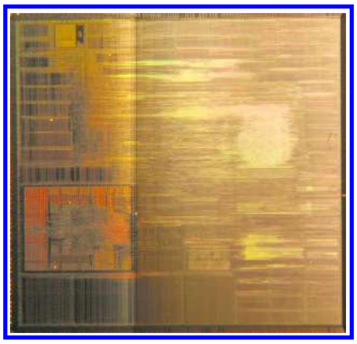
- Flexible architecture to implement the digital part of the physical layer functionality of two wireless LAN standards (5 GHz band): HIPERLAN/2, IEEE 802.11a.
- Implements the operations CL and MAC/DLC of the HIPERLAN/2 and the lower-MAC layer of the IEEE 802.11a standard.
- Contains two embedded microprocessors:
  - ✓ ARM946E-S for the implementation of the high layers of the HIPERLAN/2 standard.
  - ✓ ARM7TDMI for the implementation of the lower-MAC layer of the HIPERLAN/2 standard and for controlling the baseband modem (transmitter/receiver) of the system.
- Also, includes a MAC hardware accelerator (custom block) that implements critical functionality of the MAC layer of the IEEE 802.11a.
- Various peripherals: test and debug controller, power controller, Ethernet and PCI interfaces, SDRAM controller, DMA controller, UARTs.

Example: Wireless LAN SoC (cont'd)



Example: Wireless LAN SoC (cont'd)

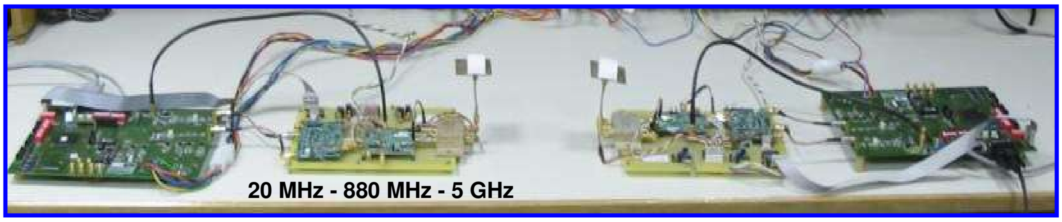
Process technology	0.18 μm CMOS
Supply voltage	1.8 V (core), 3.3 V (I/O pads)
Operating frequency	80 MHz (some modem's blocks in 40 MHz)
Average power consumption	554 mW
Equivalent gates count	4,400,000
Transistors count	> 17,500,000
Pins count	456 (about 130 are for testing/debugging purposes)
Packaging	BGA 35mm x 35mm
Chip area	9.408mm x 9.408mm ≈ 88.5 sq. mm
Core area	8.576 mm x 8.576 mm ≈ 73.5 sq. mm
Area occupied by logic	43 sq. mm
Area occupied by memories	30.5 sq. mm
Total length of interconnections	47 m (six metal layers)



Design	INTRACOM TELECOM
Fabrication	ST

EASY

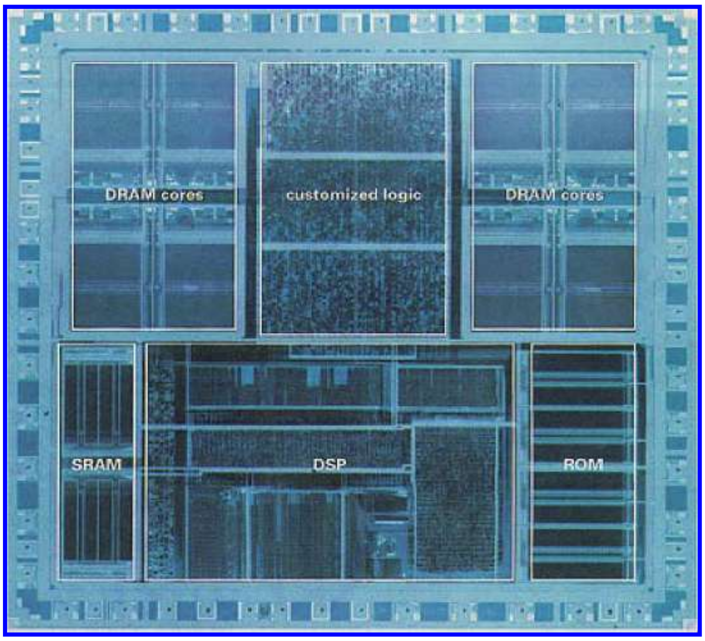
Example: Wireless LAN SoC (cont'd)



Second example: Audio processing SoC

Audio processing SoC (Siemens):

- 16-bit DSP
- Custom logic: 15,000 gates
- SRAM, ROM memory modules
- 1 Mbit DRAM



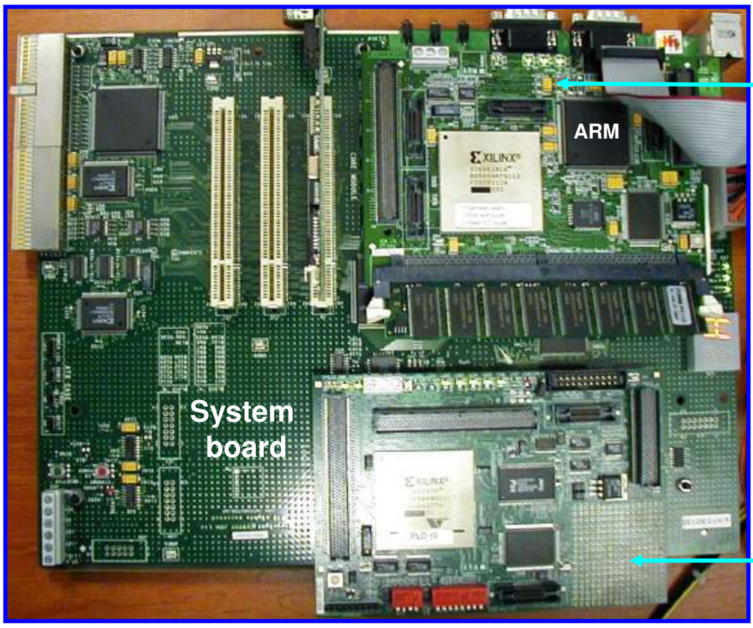
Prototyping platforms

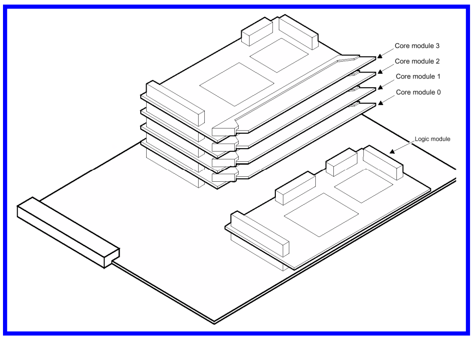
- Prototyping platforms are used after the simulation (or as an alternative to the simulation) in order to prototype the SoC design into an FPGA-based board, before its fabrication.

- Main characteristics of prototyping platforms:
  - ✓ Offer an accurate representation of the design since they are actual implementations and not just simulations.
  - ✓ Faster than simulations: they can reproduce a problem after several seconds of execution, while in HDL simulation environments things are much slower.
  - ✓ However, they are not exact replicas of the final SoC, and they cannot run at the same frequency with the real silicon SoC.
  - ✓ The design can be mapped relatively quickly (hours or days).
  - ✓ Debugging support is usually included.
  - ✓ However, tasks such as design partitioning (to the available FPGAs), clock tree routing, bus handling and memory mapping are complex and difficult.
  - ✓ Can be expensive for large designs, and sometimes they can lead to resource bottleneck (a group have to wait for another group to finish using the platform).

Example: ARM integrator platform

Contains a system board and up to 5 modules. The system board provide the AMBA bus and other system functionality. Core modules allows the presence of ARM cores in the prototype and logic modules provide user programmable logic elements (FPGAs).

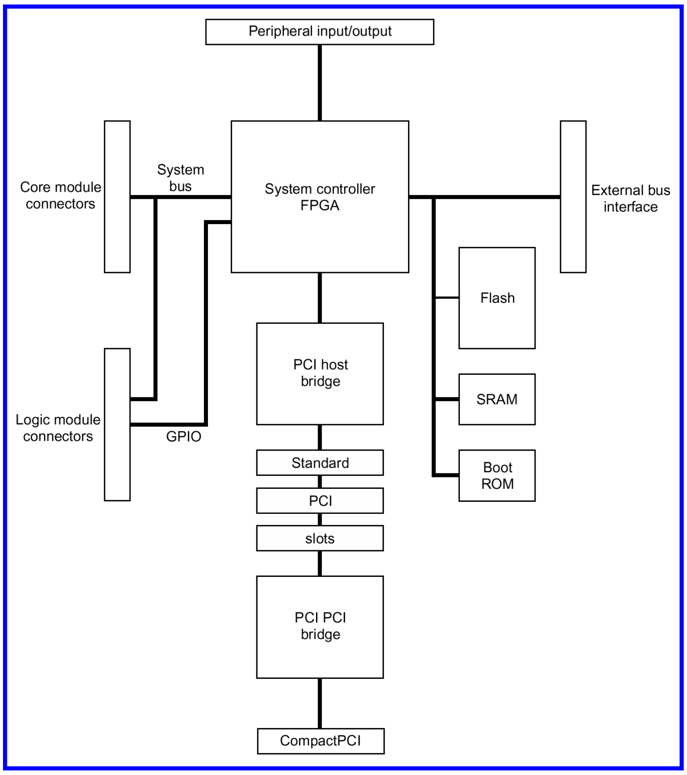




ARM integrator platform: System board

Major features of the system board:

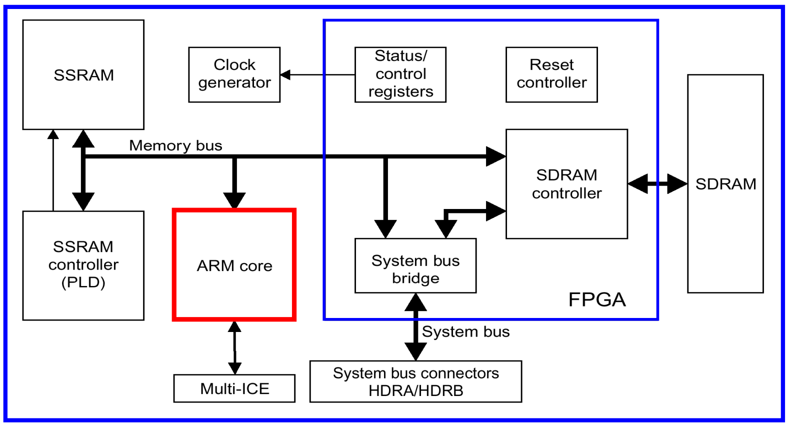
- System control FPGA which implements:
  - ✓ System bus interface to core and logic modules.
  - ✓ System bus arbiter.
  - ✓ Interrupt controller.
  - ✓ Peripheral input and output controllers.
  - ✓ Timers.
  - ✓ Reset controller.
  - ✓ System status and control registers.
- Clock generator.
- 32MB flash memory.
- Boot ROM.
- SRAM.
- System expansion, supporting core and logic modules (up to 5 in total).
- PCI bus interface.



ARM integrator platform: Core module

The major components on the core module are as follows:

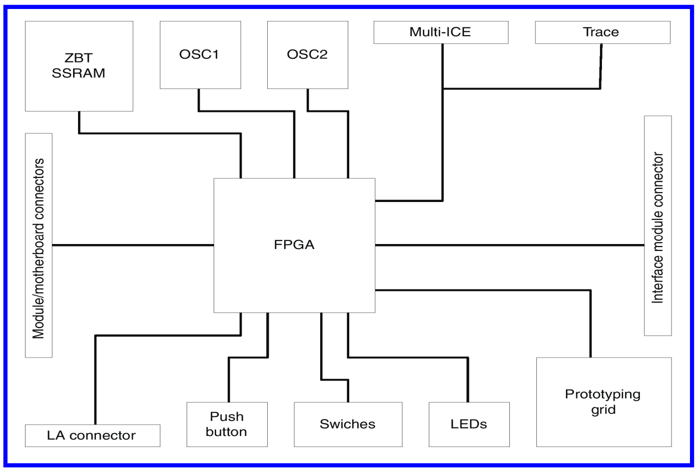
- ARM core (7 or 9 family).
- FPGA which implements:
  - ✓ SDRAM controller.
  - ✓ System bus bridge.
  - ✓ Reset controller.
  - ✓ Interrupt controller.
  - ✓ Status & control registers.
- SSRAM (256KB) and plugged SDRAM.
- SSRAM controller.
- Clock generator.
- System bus connectors.
- Debugging connectors.



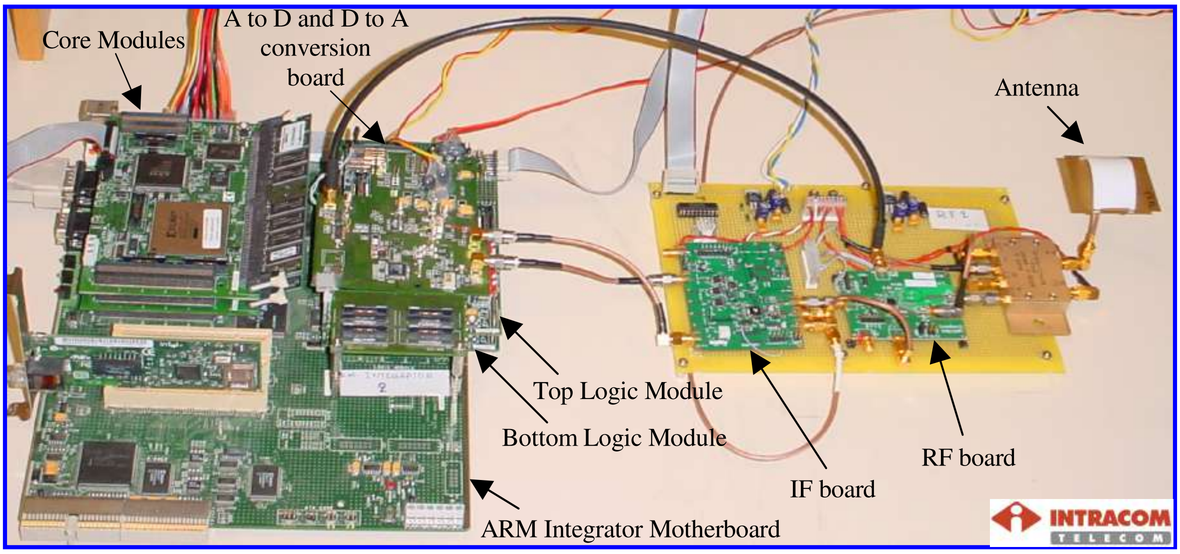
ARM integrator platform: Logic module

The logic module comprises of:

- Xilinx or Altera FPGA.
- Configuration PLD and flash Memory for storing FPGA configurations.
- 1MB SSRAM.
- Clock generators and reset sources.
- Switches.
- LEDs.
- Prototyping grid.
- Debug connectors.
- System bus connectors to a system board or other modules.



Wireless LAN System-on-Chip prototyping

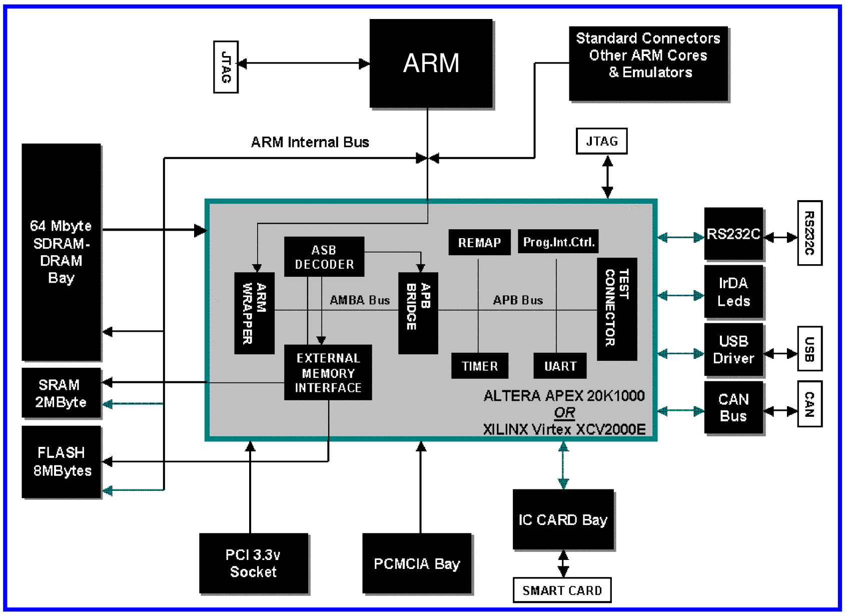


- Two ARM7TDMI core modules (one implementing the upper layers of the protocol and the second controlling the baseband modem and implementing the lower MAC protocol layer).
- Two logic modules with XILINX Virtex E 2000 FPGAs implementing the baseband modem functionality. The average FPGA utilization was 87%.

Second example: Carmen platform

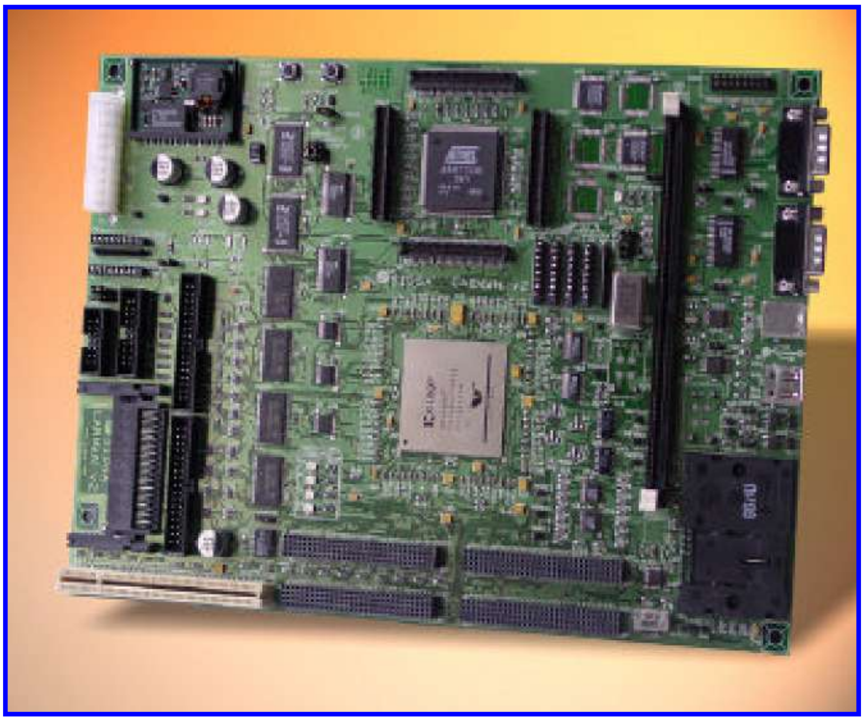
Major components:

- ARM processor (7 or 9)
- FPGA (XILINX Virtex XCV2000E or ALTERA APEX20K1000).
- 30 MHz system clock.
- 64MB SRAM, expandable SDRAM up to 256MB.
- 8Mbyte FLASH and 2MB high-speed SRAM.
- Multiple I/O interfaces: PCMCIA, PCI, USB
- RS232 etc.
- AMBA bus components.
- RAM & FLASH controllers.
- Timers.
- UART.
- Interrupt controller.



CARMEN platform by SIDSA

Second example: Carmen platform (cont'd)



Conclusions

- With the current technology and in the context of increasingly complex applications and strong market pressure, system-on-chip is a natural approach for several embedded applications.
- Programmable components provide the necessary flexibility and custom hardware is needed for time-critical tasks implementation and for low power consumption.
- The real bottleneck in SoC design is productivity.
- Solutions are the IP-based design (blocks reuse), and the improvement of existing design methodologies and tools.
- Prototyping platforms (e.g. ARM Integrator, Carmen) are used after the co-simulation (or as an alternative to the simulation) in order to prototype and test the SoC design into an FPGA-based board, before its fabrication.

7b. Reconfigurable systems

University of Thessaly

Department of Computer and Communication Engineering

Reconfigurable computing

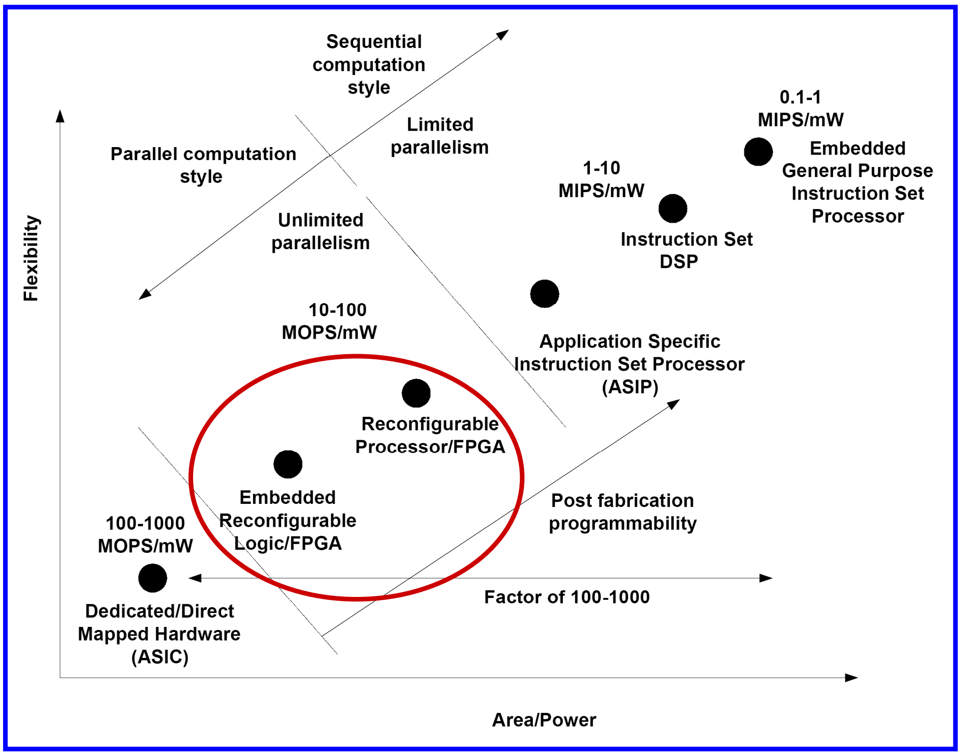
- Reconfigurable computing refers to systems incorporating some form of hardware programmability, that customizes how the hardware is used using a number of physical control points.

- These control points can be changed periodically in order to execute different applications using the same hardware.

- Since, the contradictory requirements of modern applications for both flexibility and implementation efficiency, cannot be satisfied by conventional instruction-set processors and application-specific circuits, reconfigurable hardware offers a good balance between implementation efficiency and flexibility.

- This is because the reconfigurable hardware combines post-fabrication programmability with the parallel computation style of application specific circuits, which is more efficient in comparison to the sequential computation style of instruction-set processors.

Reconfigurable hardware


Sequential computation style
Limited parallelism
Parallel computation style
Unlimited parallelism
Flexibility
0.1-1 MIPS/mW
Embedded General Purpose Instruction Set Processor
1-10 MIPS/mW
Instruction Set DSP
Application Specific Instruction Set Processor (ASIP)
10-100 MOPS/mW
Reconfigurable Processor/FPGA
Embedded Reconfigurable Logic/FPGA
100-1000 MOPS/mW
Dedicated/Direct Mapped Hardware (ASIC)
Post fabrication programmability
Factor of 100-1000
Area/Power

Reconfigurable hardware (cont'd)

- There are additional reasons for using reconfigurable resources in System-on-Chip (SoC) design.
- The increasing non-recurring engineering (NRE) costs push designers to use the same SoC in several applications and products for achieving low cost per chip.
- The presence of reconfigurable resources allows the fine tuning of the chip for different products or product variations.
- Also, the increasing complexity in future designs adds the possibility of using design flows, which can require costly and slow redesign of the chip. In this way:
  - ✓ Reconfigurable elements are often homogenous arrays, which can be pre-verified to minimize the possibility of design errors.
  - ✓ Post-manufacturing programmability of reconfigurable elements allows correction of problems later than the fixed hardware.

Types of reconfiguration

- Logic reconfiguration.
- Instruction-set reconfiguration.
- Static reconfiguration or dynamic reconfiguration.
- Full or partial reconfiguration.
- Fine-grained, medium-grained and coarse grained reconfiguration.

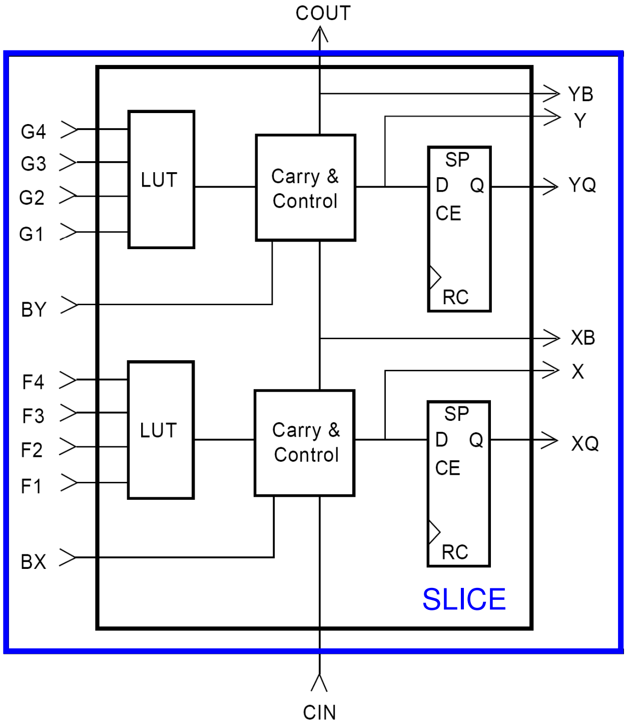
Logic reconfiguration

- A typical block for logic reconfiguration contains a look-up table (LUT), an optional D flip-flop (latch) and additional combinational logic.
- The LUT allows any logic function to be implemented, providing generic logic.
- The latch can be used for pipelining reasons, registers for holding logic values or any other situation where clocking is required.
- The additional combinational logic is usually ‘carry logic’ used to speed up carry-based computations (e.g. additions).
- The logic blocks located at the periphery of the reconfigurable device (I/O blocks) can be of different architecture dedicated to I/O operations.
- In addition to operating as a function generator, each LUT can provide RAM functionality.
- Furthermore, two or more logic blocks can be combined to implement more complex functions.

Logic reconfiguration (cont’d)

Example of basic logic block (Xilinx Virtex FPGA):

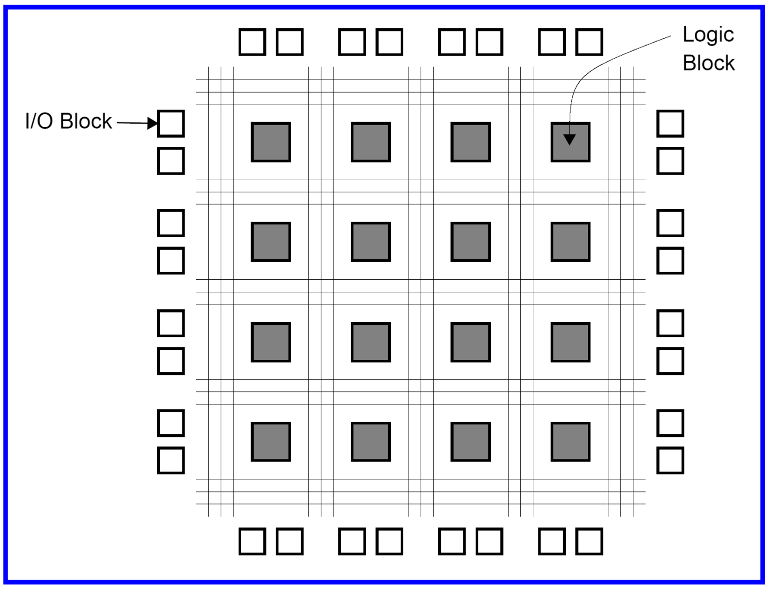
- Each FPGA slice contains two basic reconfigurable logic blocks.
- The 4-bit look-up table (LUT) is implemented with a multiplexer whose select lines are the inputs of the LUT and whose inputs are constants.



Logic reconfiguration (cont'd)

- The logic blocks are grouped to matrices overlaid with an interconnection network of wires.
- The reconfiguration of the logic blocks is achieved by using bits from an SRAM memory to control the state of the transistors within the LUTs.
- The functionality is modified by downloading bit stream of reconfiguration data onto the hardware.

Basic architecture of a reconfigurable logic device (FPGA)



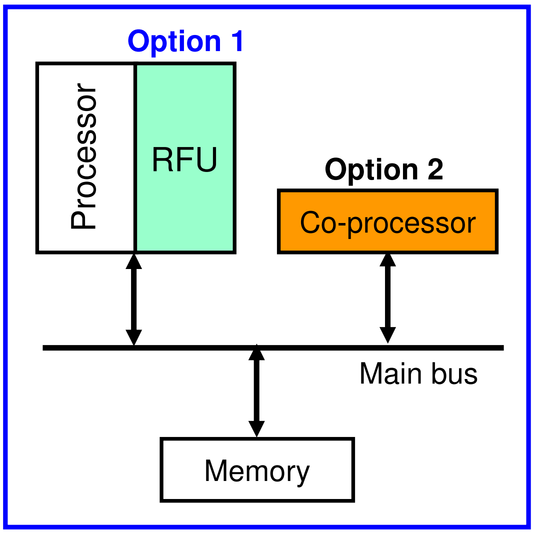
Instruction-set reconfiguration

- The concept of instruction-set reconfiguration refers to architectures consisting of microprocessor and reconfigurable logic.
- The key benefit is the combination of software flexibility with hardware efficiency.
- One promising approach is the use of reconfigurable instruction-set processors (RISP), which have the capability to adapt their instruction set to the application being executed through a reconfiguration in their hardware.
- Through the adaptation, specialized hardware accelerates the execution of the application.
- By moving the execution of some application tasks to the reconfigurable part of the processor, a remarkable improvement in performance can be achieved.
- One important issue is the type of interface between the microprocessor and the reconfigurable logic.

Instruction-set reconfiguration (cont'd)

- **Option 1**: Use of a reconfigurable functional unit (RFU) inside the processor. The instruction decoder issues instructions to the RFU as it is one of the functional units of the processor. The communication cost is very small and the speed improvement is significant.

- **Option 2**: The reconfigurable logic is placed next to the processor (operating as a co-processor). Communication is performed by using a protocol.



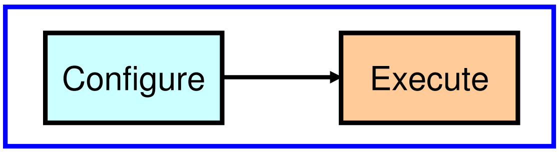
Static reconfiguration

- Static reconfiguration (often referred as compile-time reconfiguration) is the simplest and most common approach for implementing applications with reconfigurable logic.

- It involves hardware changes at a relatively slow rate, and consists of a single system-wide configuration.

- Prior the execution of an application, the reconfigurable resources are loaded with their respective configurations, and during the execution of the operation, the reconfigurable resources will remain in the same configurations (i.e. remain static) throughout the end of application execution.

- Advantages: Higher performance than pure software implementation, lower cost than specific hardware.

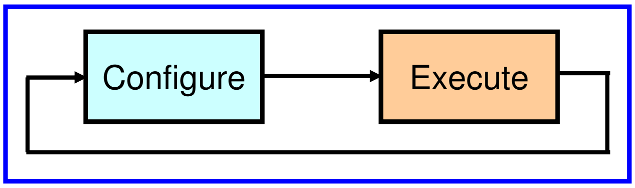


Static reconfiguration (cont'd)

- In order to reconfigure a statically reconfigurable architecture, the system has to be halted while the reconfiguration is in progress and then restarted with the new configuration.
- Traditional FPGA architectures are primarily statically programmed devices, allowing only one configuration to be loaded at a time.
- This type of FPGAs is programmed using a serial stream of configuration information (stored in an SRAM), requiring a full reconfiguration if any change is needed.

Dynamic reconfiguration

- Whereas static reconfiguration allocates logic for the duration of an application, dynamic reconfiguration (often referred as run-time reconfiguration) uses a dynamic allocation scheme that re-allocates hardware at run time (i.e. during execution of the application).
- The physical hardware is smaller than the sum of required resources. With dynamic reconfiguration we swap the number of configurations in and out of the actual hardware, as they are needed.
- Problems: Divide the algorithms into time-exclusive segments that do not need to run concurrently and manage the transmission of intermediate results from one configuration to the next.
- Advantages: The benefits of static reconfiguration are remained, and we can achieve an efficient trade-off between time and space (cost).

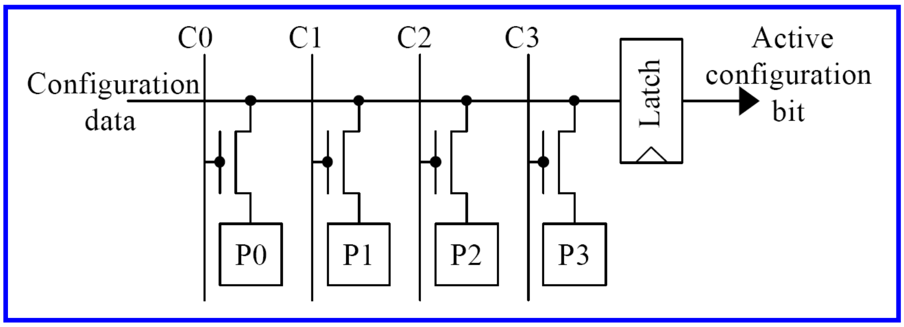


Dynamic reconfiguration (cont'd)

- There are two different configuration memory styles that can be used with dynamic reconfigurable systems.
- Single context device is a serially programmed device that requires a complete reconfiguration in order to change any of the programming bits.
- Multi-context device has multiple layers of programming bits, each of which can be active at a different point in time.
- In order to implement run-time reconfiguration onto a single context device (FPGA), the different full configurations must be grouped into layers within the configuration memory, and each layer is swapped in and out of the FPGA as needed.
- Although, in single context devices, the reconfiguration of the hardware is simple, there is a high-overhead when only a small part of the configuration memory needs to be changed.
- Because in such devices only full reconfigurations are allowed, a good partitioning of the different configurations between layers is essential.

Dynamic reconfiguration (cont'd)

- Multi-context architectures include multiple memory bits for each programming bit location.



- One layer of configuration information can be active at a given moment, but the device can quickly switch between different layers (contexts) of already-programmed configurations.
- However, this method requires more area than single context structures, since there must be many storage units per programming location.

Partial reconfiguration

- In some cases, configurations do not occupy the full reconfigurable hardware, or only a part of a configuration requires modification.
- In both of these situations, a partial reconfiguration of the reconfigurable resources is desired, rather than the full reconfiguration supported by the serial architectures (programmed using serial streams of reconfiguration information).
- Partially reconfigurable architectures use addresses (like a RAM device) to specify the target location of the configuration data, allowing the selective reconfiguration of the reconfigurable recourses.
- The undisturbed portions of the reconfigurable resources may continue execution, allowing the overlap of computation (execution) with reconfiguration.
- Attention is required in order to manage the transmission of data between the unchanged and changed portions of the reconfigurable resources.
- Partially, run-time reconfigurable architectures can allow complete reconfiguration flexibility (Xilinx 6200) or may require a full array column to be reconfigured at once (Xilinx Virtex).

Granularity of building blocks

- Granularity refers to the level of manipulation of data in reconfigurable devices.
- There are three types of granularity: fine-grain which correspond to bit-level manipulation of data, medium-grain manipulating data with varying number of bits and coarse-grain which implies word-level operations.
- In fine-grained architectures, the basic programmable building block usually consists of a combinational network and a few flip-flops.
- Each logic block can be programmed into a simple logic function (e.g. full-adder), and the blocks are connected through an interconnection network.
- Commercially available FPGAs (Xilinx, Altera etc.) are based on fine-grained architectures.
- Although highly flexible, these systems exhibit low efficiency in terms of speed and area.

Granularity of building blocks (cont'd)

- Reconfigurable systems which use logic blocks of larger granularity are categorized as medium-grained.
- For example, Garp architecture has been designed to perform arithmetic computations with up to four 2-bit inputs, while Chess architecture operates on 4-bit values with each of its cells acting as a 4-bit ALU.
- The major advantage of medium-grained systems, in comparison with fine-grained systems, is that they better utilize the chip area, since they are optimized for specific operations.
- However, a drawback is the high overhead that is inserted when synthesizing operations which are incompatible with the simplest logic block architecture.

Granularity of building blocks (cont'd)

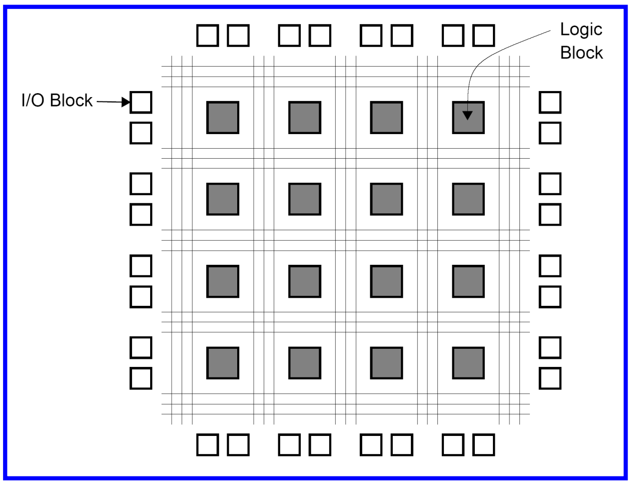
- Coarse-grained architectures are intended for the implementation of tasks dominated by word-width operations.
- Because the used logic blocks are optimized for 'large' computations, they will perform these operations much more quickly (and by consuming less area) than a set of smaller cells connected to form the same structure.
- However, they exhibit low flexibility.
- For example, RaPiD device is composed of 16-bit adders, multipliers and registers. If only 1-bit values are required, then the use of this device has an unnecessary area and speed overhead, as all 16 bits are computed.
- Such architectures can be much more efficient than fine-grained and medium-grained architectures, for implementing functions closer to their basic word size.

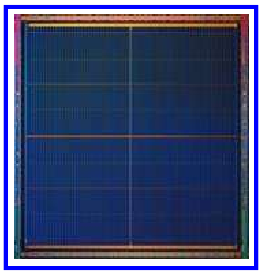
Reconfigurable devices

- FPGAs (Filed-Programmable Gate Arrays):
  - ✓ Currently represent the most popular segment of reconfigurable devices.
  - ✓ They can be reconfigured in seconds either statically or dynamically/partially.
  - ✓ Advantages: react to last minute design changes in order to correct errors or upgrade functions, useful for prototyping ideas before implementation and for meeting time-to-market deadlines.
- Integrated circuit devices with embedded reconfigurable resources:
  - ✓ Represent an alternative to FPGAs.
  - ✓ They are based on a combination of a programmable CPU and a reconfigurable array of word-level data path units.
  - ✓ Also, called hardware-software design platforms, they increases the productivity and the success probability, and reduces the design time in comparison with custom SoC designs.
- Embedded reconfigurable cores: reconfigurable cores are embedded in a single SoC/ASIC.

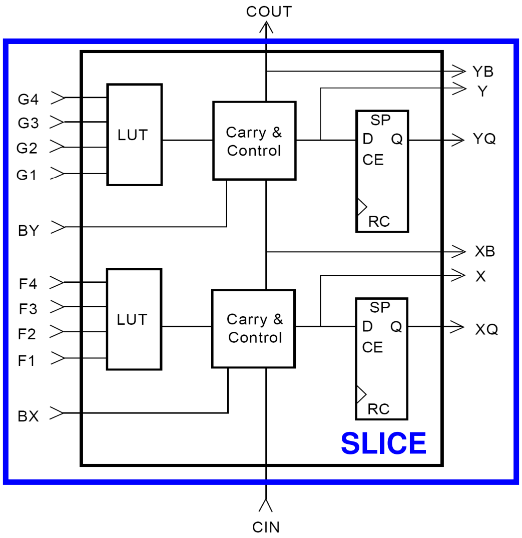
FPGAs

- In FPGAs, the logic blocks are grouped to matrices overlaid with an interconnection network of wires.
- The reconfiguration of the logic blocks is achieved by using bits from an SRAM memory to control the state of the transistors within the LUTs.
- The functionality is modified by downloading the reconfiguration data onto the hardware.



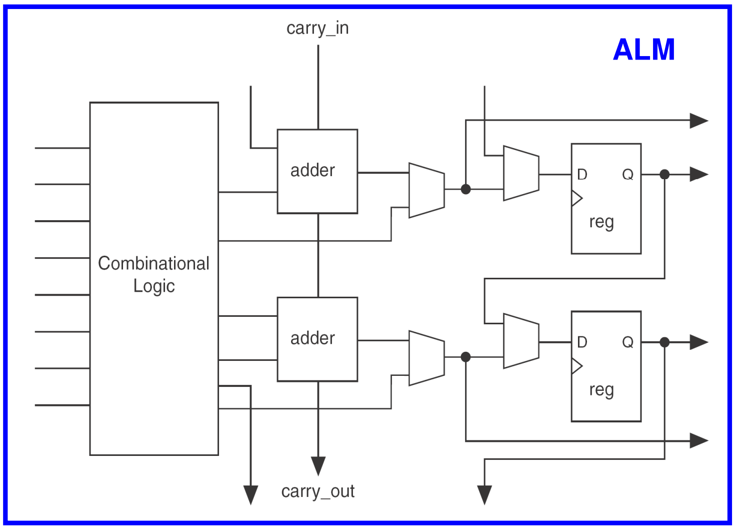


Xilinx Virtex 4 FPGA



- Virtex 4 is one of the new generation FPGA from Xilinx.
- Each configurable logic block (CLB) is made up of 4 slices.
- Each slice contains 2 LUTs, 2 storage elements, carry look-ahead circuitry and few multiplexers.
- Each CLB has internal fast interconnect and connects to a switch matrix to access general routing resources.
- Includes enough RAM resources for running complex applications, has logic density up to 200K logic slices and can achieve clock rates up to 500MHz.
- Fine-grained architecture with embedded word-level modules (multipliers), supporting dynamically (partially) reconfiguration.
- Technology: 90nm CMOS with 1.2 Volts supply voltage.

Altera Stratix II FPGA

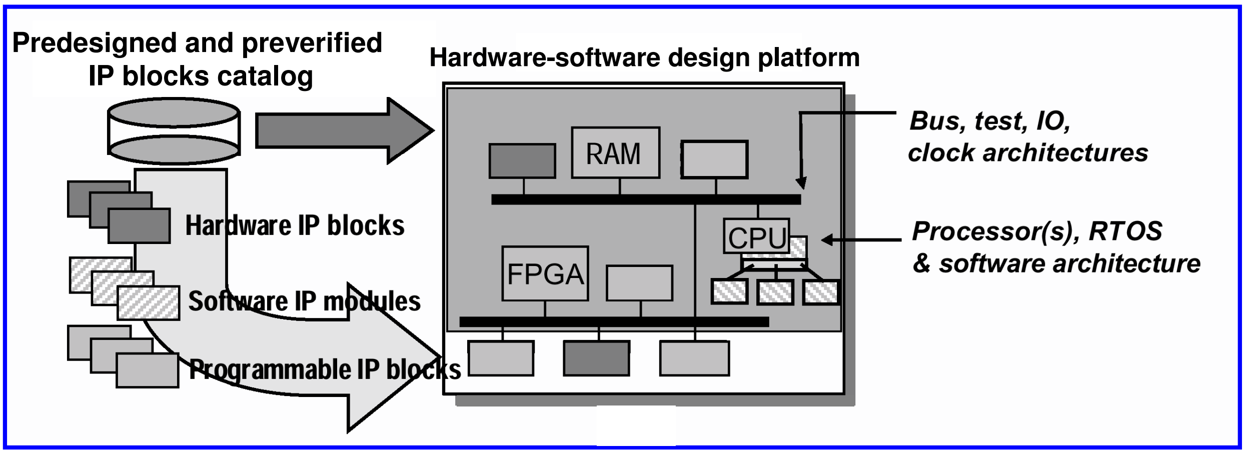


- Similar logic structure consisting of logic array blocks (LAB).
- Each LAB contains 8 adaptive logic modules (ALM).
- Each ALM contains 2 LUTs of 4-inputs and 4 LUTs of 3 inputs, 2 storage elements, carry look-ahead circuitry and few multiplexers.
- Each LAB has internal fast interconnect and connects to the rest array.
- Includes enough RAM resources for running complex applications, has logic density up to 80K ALMs and can achieve clock rates up to 500MHz.
- Fine-grained architecture with embedded word-level modules (multipliers), supporting static (serially or parallel) reconfiguration.
- Technology: 90nm CMOS with 1.2 Volts supply voltage.

IC devices with embedded reconfigurable resources

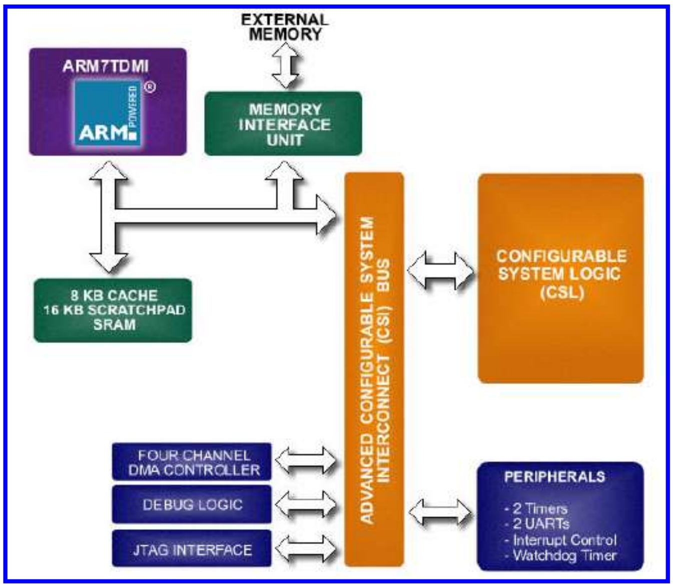
- Custom SoC designs have very high cost, making them practical only when we have production of a very large number of chips (millions).
- IC devices with embedded reconfigurable devices (also called hardware-software design platforms) have quite less cost, providing developers with an alternative solution to custom SoCs and standard processors with external peripherals.
- Hardware-software design platform is a stable SoC architecture for a target application or family of applications that is based on a combination of a programmable CPU and a reconfigurable array of data-path units.
- It can be extended and customized relatively fast and easy.
- The use of such platforms increases the productivity and the success probability, and reduces the design time.
- Derivative designs (implementing similar applications) can be easily created by using software or hardware modifications.
- Diverse applications each requires a different platform (it is difficult to use the same platform for a telecom application where the control functionality is dominant and for a multimedia application where the data-processing tasks are dominant).

Hardware-software platforms design concept


Predesigned and preverified IP blocks catalog
Hardware-software design platform
Hardware IP blocks
Software IP modules
Programmable IP blocks
RAM
CPU
FPGA
Bus, test, IO, clock architectures
Processor(s), RTOS & software architecture

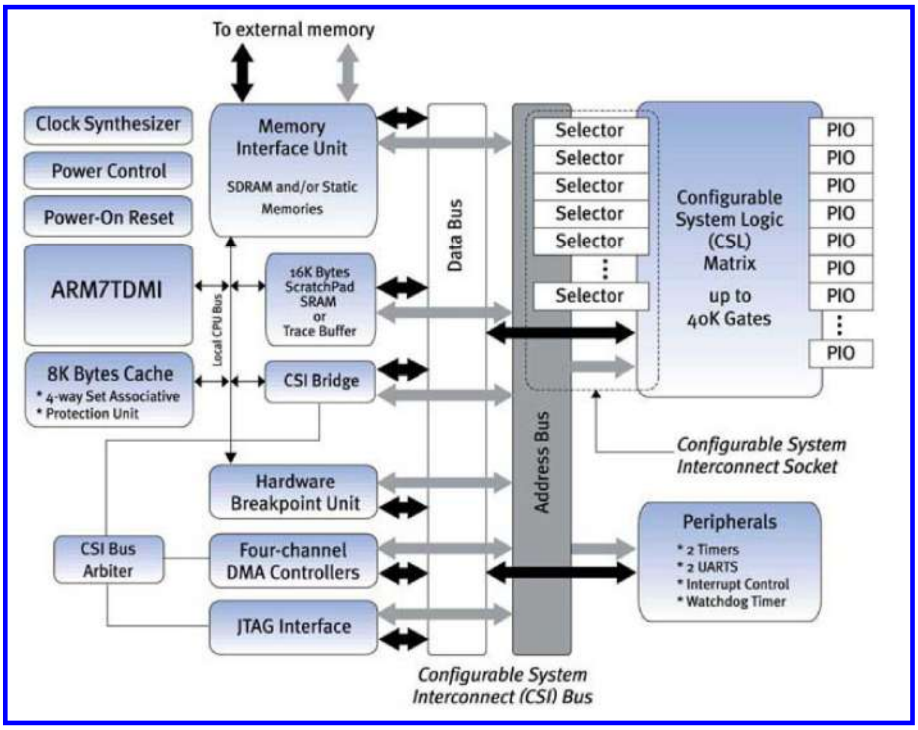
Triscend A7 platform

- Triscend A7 is a 32-bit configurable system-on-chip.
- Combines a 32-bit ARM7TDMI embedded processor core with a flexible Configurable System Logic (CSL) matrix (statically configurable), a robust memory subsystem, a high-performance custom internal bus, and other system peripheral functions onto a single chip.
- Triscend also offers development tools that integrates third-party EDA and processor development tools for rapidly development of a SoC covering specific application needs.



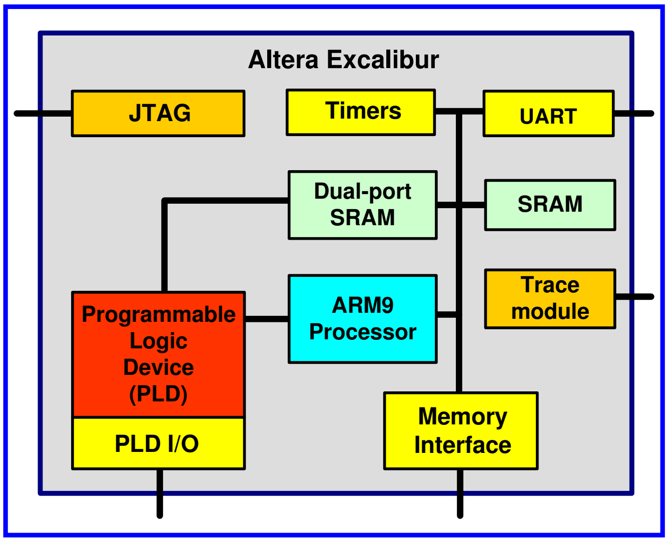
Triscend A7 platform (cont'd)

Detailed architecture



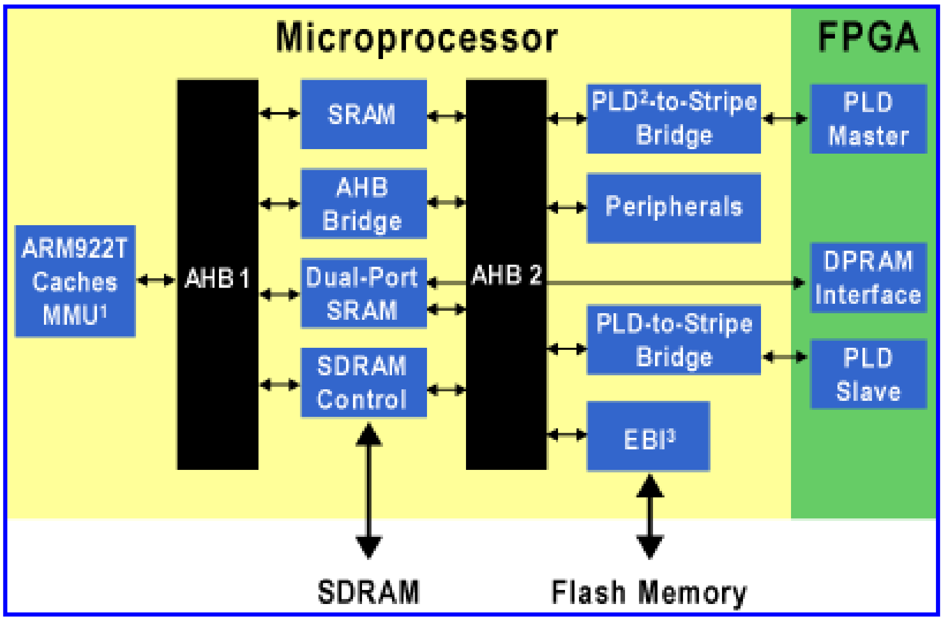
Altera Excalibur platform

- Excalibur platform integrates a 32-bit ARM922T embedded processor with associated caches and a memory management unit.
- Contains single and dual-port RAM modules, memory controllers, several peripherals (UART, timers, interrupt controller) and debugging modules. Supports the AMBA bus architecture by containing two buses with dedicated bus bridges.
- The FPGA section contains Altera's PLD devices (static, fine-grained architecture), which can contain 4,160 to 38,400 logic elements or 100K to 1M gates. FPGAs with such densities provide access to a wide range of IP functional blocks and interfaces.



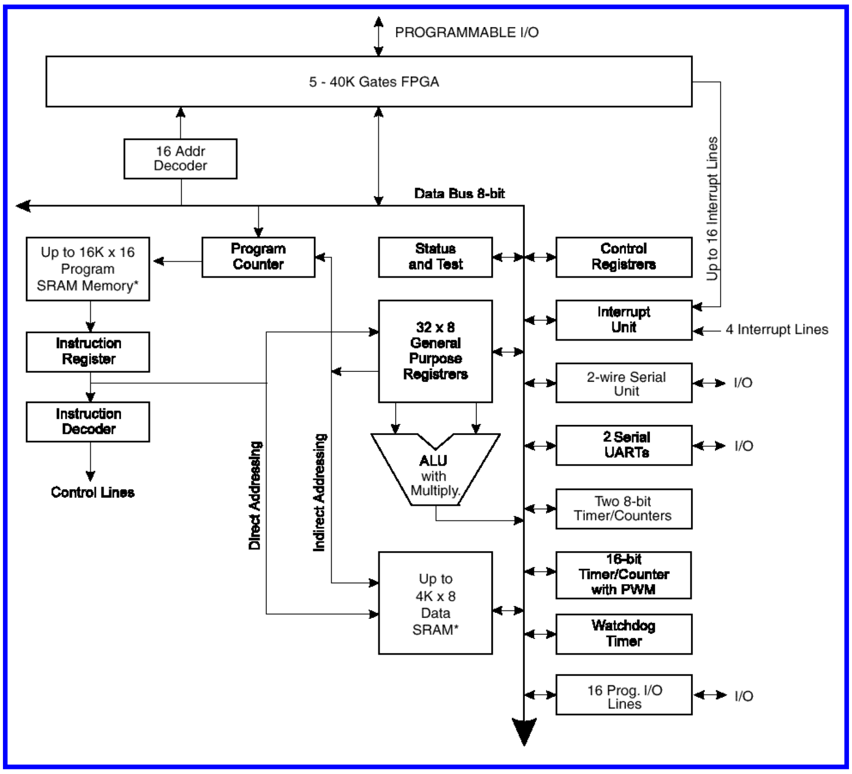
Altera Excalibur platform (cont'd)

Detailed architecture



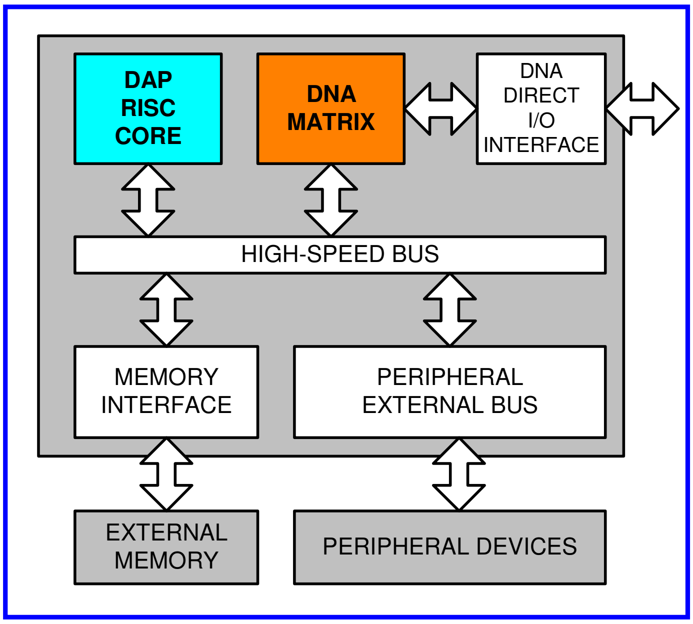
Atmel FPSLIC platform

- Atmel's Field Programmable System-level IC integrates the Atmel's AVR 8-bit RISC processor with an Atmel's AT40K FPGA, and several peripherals.
- The FPGA part is based on a fine-grained architecture and allows dynamic full or partial reconfiguration.



IPflex DAPDNA-2 platform

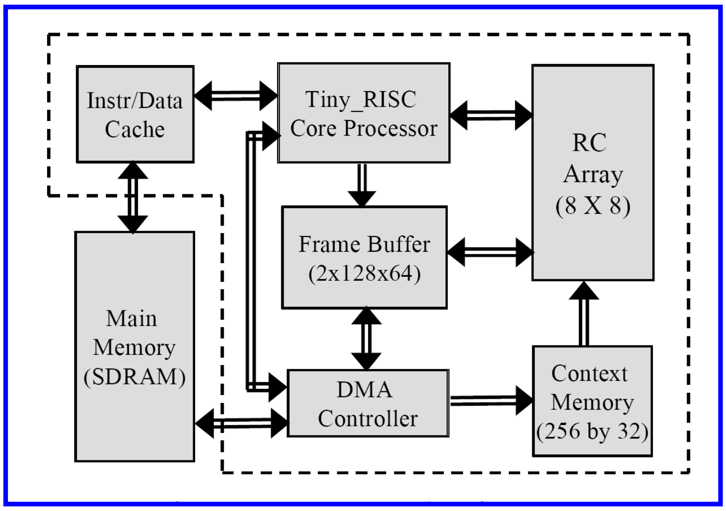
- IPpflex platform combines the IPflex's DAP RISC core (32-bit with 8KB data cache and 8KB instruction cache) with the DNA matrix-based device and several interfaces.
- The DNA matrix is a coarse-grained architecture, allows dynamic reconfiguration and contains 376 processing elements comprised of computation units, memory and counters.



MorphoSys platform

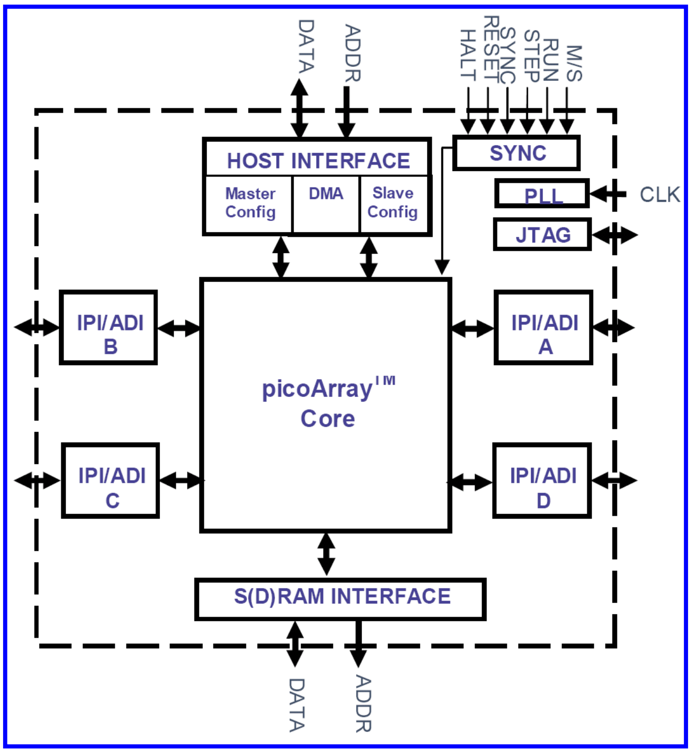
- MorphoSys platform contains a Tiny RISC processor that is a 4-stage pipeline, MIPS-like RISC machine with 16 32-bit registers, 32-bit ALU/shift unit and on-chip data cache memory
- The reconfigurable array consists of an 8x8 matrix of Reconfigurable Cells (RC).
- Each RC comprises an ALU-Multiplier, a shift unit, input multiplexers, and a register file with five 16-bit registers.
- The array is based on a coarse-grained architecture, that allows dynamic reconfiguration.

Academic approach:
University of California at Irvine



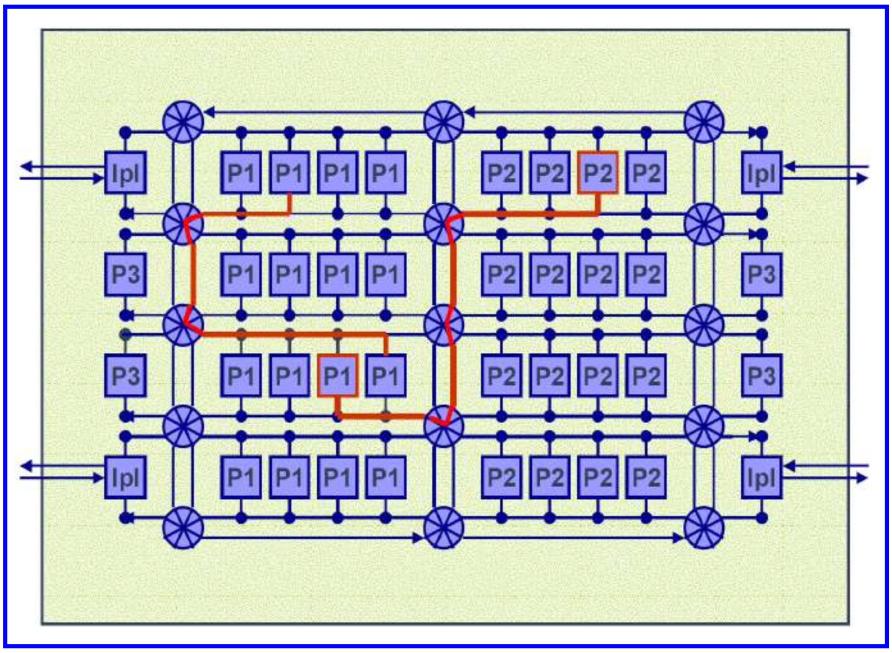
picoChip platform

picoChip platform consists of an array of RISC-like processors (picoArray) and peripherals (external microprocessor interface, external memory interface, interfaces allowing multiple arrays to be connected together).



picoChip platform (cont'd)

The array contains 322 elements; 308 processors (16-bit architecture with 3-way LIW and local memory) and 14 co-processors, all connected by programmable interconnect modules.



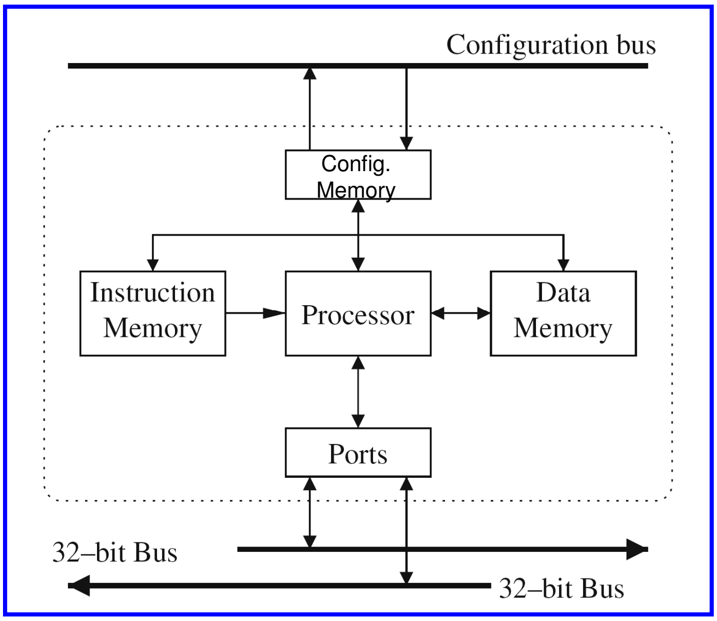
picoChip platform (cont'd)

Processor types in the array

Processor type	Description	Number of processors	Memory per processor
STANDARD	1. ALU 2. ALU & memory access unit 3. MAC unit	240	768 Bytes
MEMORY	1. ALU. 2. ALU & memory access unit. 3. Multiply unit	64	8,704 Bytes
CONTROL	1. ALU 2. ALU & memory access unit. 3. Multiply unit	4	65,536 Bytes
FAU (Function Acceleration Units)	Units for implementing specific functions that are useful for digital signal processing (e.g. trellis operations for FEC decoding etc.)		

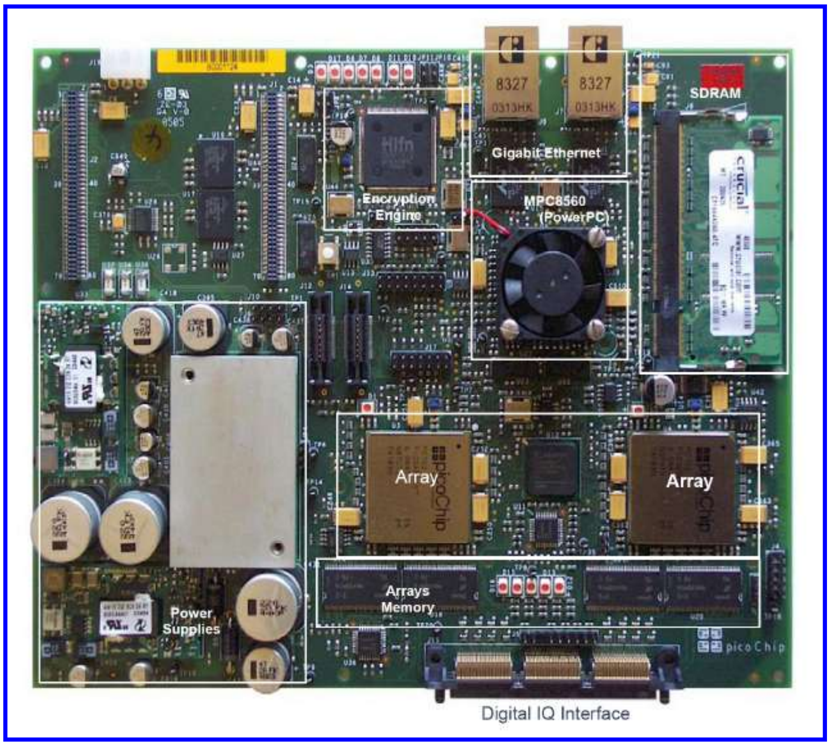
picoChip platform (cont’d)

Processor structure



picoChip platform (cont’d)

- The flexible nature of the picoArray technology allows the implementation of several communications standards (IEEE 802.11 - wireless LAN, IEEE802.16 - outdoor wireless).
- The physical layer is implemented on the arrays, while the MAC layer of the standards is implemented on a PowerPC external processor.
- An encryption engine implementing basic standards is also available.



Embedded reconfigurable cores

- Embedded reconfigurable cores are hardware cores with programmable capabilities that can be embedded in several SoCs and ASICs.
- Embedded reconfigurable cores provide:
  - ✓ Hardware flexibility in implementing multiple applications.
  - ✓ Lower power consumption than the programmable processors.
  - ✓ Lower hardware cost.
- For example, the array of reconfigurable cells contained in the MorphoSys platform is available as an autonomous IP block for integration in SoCs or ASICs.

Conclusions

- Reconfigurable hardware offers a good balance between implementation efficiency and flexibility, by combining characteristics of both instruction-set processors and application-specific circuits.
- Reconfigurable hardware is also used for achieving lower cost and lower design effort and time than specific circuits, and better performance (speed and power consumption) than instruction-set processors.
- Several types of reconfiguration can be used: logic reconfiguration, instruction-set reconfiguration, static reconfiguration or dynamic reconfiguration, full or partial reconfiguration, fine-grained, medium-grained or coarse grained reconfiguration.
- Several reconfigurable devices are available: FPGAs, integrated circuit devices with embedded reconfigurable resources or hardware-software design platforms, and embedded reconfigurable cores.

8. Communication in embedded systems

University of Thessaly

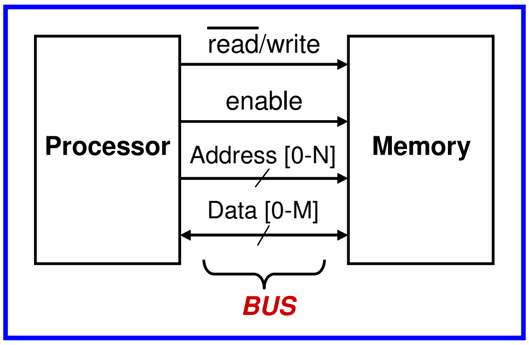
Department of Computer and Communication Engineering

Embedded system functionalities

- Processing:
  - ✓ Used to transform data.
  - ✓ Implemented using programmable processors and custom hardware blocks.
- Storage:
  - ✓ Used to maintain data.
  - ✓ Implemented using memory modules.
- Communication:
  - ✓ Used to transfer data between processors, custom hardware blocks, peripherals and memories within a system.
  - ✓ Implemented using system buses.
  - ✓ Examples: Common forms of communication are when a processor read or writes a memory or when a processor reads or writes a peripheral’s register.

What is a bus ?

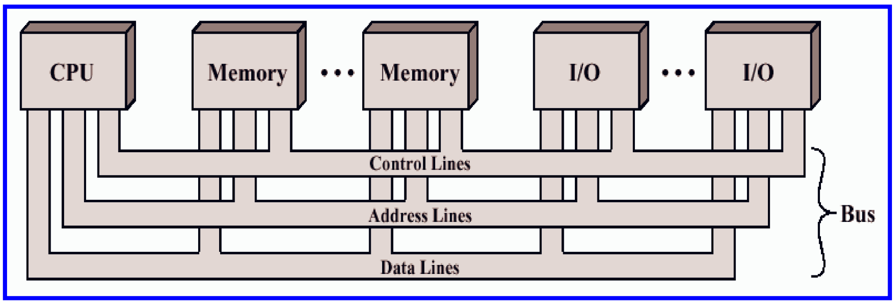
- A bus consists of wires connecting two or more blocks (processors, memories, custom blocks etc.).
- Each wire may be unidirectional (read/write, enable, address) or bi-directional (data).
- One line may represent a set of wires (address, data).
- A bus has an associated protocol describing the rules for transferring data over its wires.



Simple processor-memory bus

What is a bus ? (cont'd)

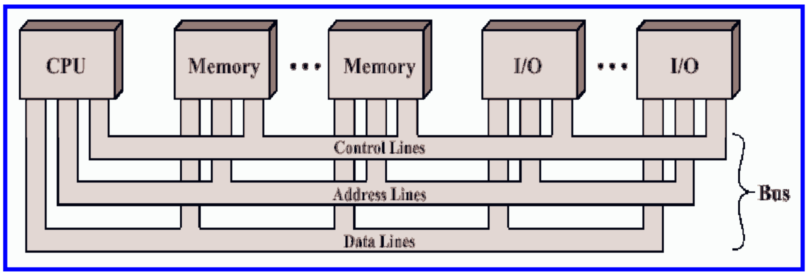
- A system bus may support the following transfers:
  - ✓ Memory to processor: the processor reads instructions and data from memory.
  - ✓ Processor to memory: the processor writes data to memory.
  - ✓ I/O (or custom block) to processor: the processor reads data from the I/O module (e.g from the register of the I/O module).
  - ✓ Processor to I/O: the processor writes data to the I/O device.
  - ✓ I/O to or from memory: I/O module allowed to exchange data directly with memory without going through the processor (DMA).



System bus

Bus signals

- Address lines and data lines.
- Control lines:
  - ✓ Memory write: data on the bus written into the addressed location.
  - ✓ Memory read: data from the addressed location placed on the bus.
  - ✓ I/O write: data on the bus output placed to the addressed I/O port.
  - ✓ I/O read: data from the addressed I/O port placed on the bus.
  - ✓ Bus REQ: indicates a module needs to get control of the bus.
  - ✓ Bus GRANT: indicates that the requesting module has been granted bus control.
  - ✓ Interrupt REQ: indicates that an interrupt is pending.
  - ✓ Interrupt ACK: Acknowledges that pending interrupt has been recognised.
  - ✓ Reset: initialises everything connected to the bus.
  - ✓ Clock: on a synchronous bus everything is synchronized to this signal.

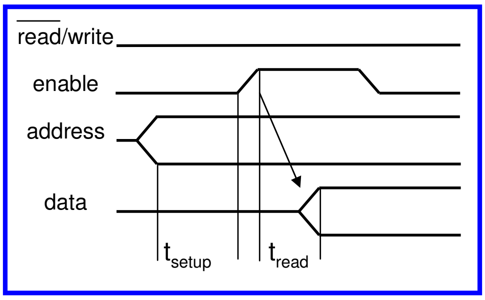


System bus

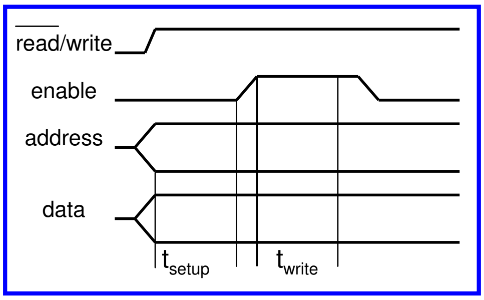
Timing diagrams

- The most common way to describe a hardware bus protocol is through timing diagrams.
- In a timing diagram control signals are either high or low, while data or address lines can be either invalid (a single horizontal line) or valid (two horizontal lines).
- For the simple bus (processor-memory), the processor must set the read/write signal to low for a read operation to occur.
- When the enable signal is high, triggers the memory to put data on the data lines after t_{read}, and the processor must place the address on the address lines at least t_{setup} before setting the enable line high.
- For a write operation the read/write signal is set to high.
- The data must be valid t_{setup} before the set of the enable signal that triggers the memory to accept the data from the data lines after t_{write}.

Read protocol

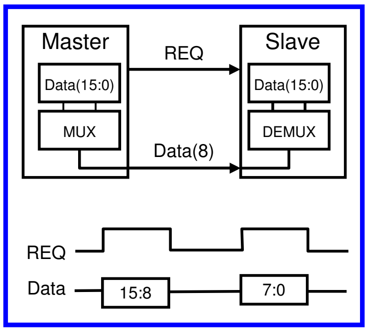


Write protocol

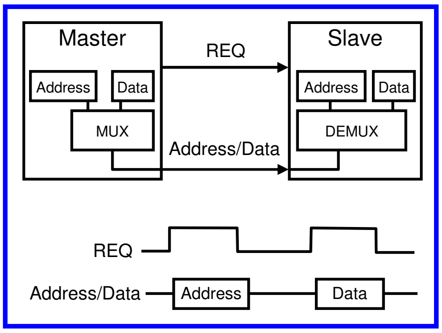


Basic protocol concepts

- A bus protocol typically involves two actors: master and slave. The master initiates the data transfer & the slave responds to the initiation request. Usually, in a processor-memory bus, master is the processor and slave is the memory (i.e. the memory cannot initiate a data transfer).
- In each data transfer one actor is the sender and another one is the receiver (independently from who is master and who is slave). This data direction is also defined by the bus protocol.
- A protocol has to be able to handle address and regular data.
- Another protocol concept is multiplexing: share a set of wires to multiple data pieces. The most common ways are: serializing and mixing.

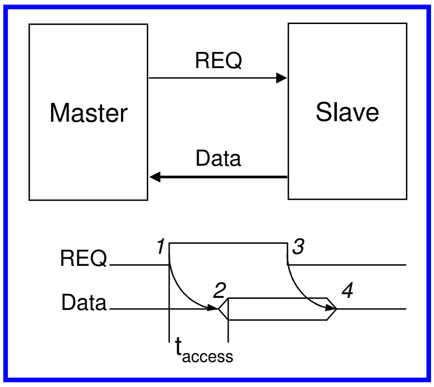


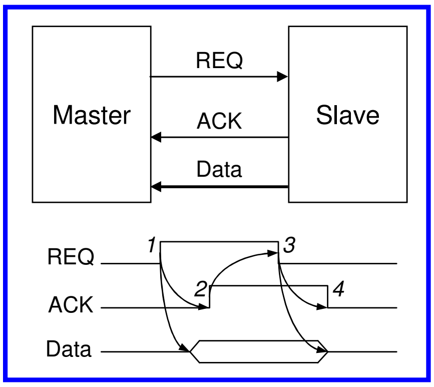
Serializing of data



Mixing of address and data

Protocol control methods





Strobe protocol

1. Master set REQ to receive data.
2. Slave puts data on bus within time t_{access}.
3. Master receives data and resets REQ.
4. Slave ready for next request.

Master*: *I want the data on the data bus after a predefined access time.

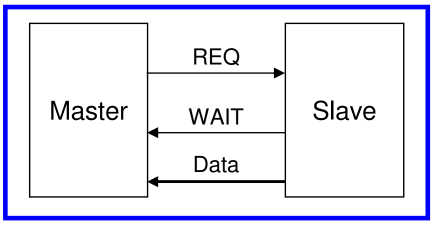
Handshake protocol

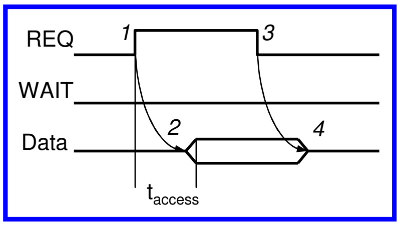
1. Master sets REQ to receive data.
2. Slave puts data on bus and sets ACK.
3. Master receives data and resets REQ.
4. Slave ready for next request.

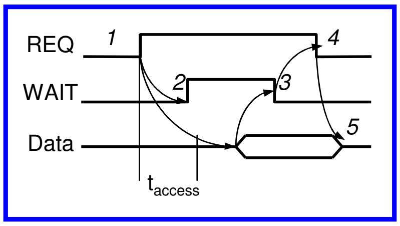
Master*: *I want the data on the data bus soon and let me know when it's ready.

Protocol control methods (cont'd)

- A handshake protocol can adjust to a slave with varying response times, unlike the strobe protocol. However, when response time is known, the handshake protocol is slower and requires an extra line for acknowledge.
- To achieve both speed and varying response time advantages, a compromise protocol is often used (two cases: fast response and slow-response cases).







***Master**: I want the data on the data bus after a predefined access time and if you cannot finish by then let me know that, and then let me know when it's ready.*

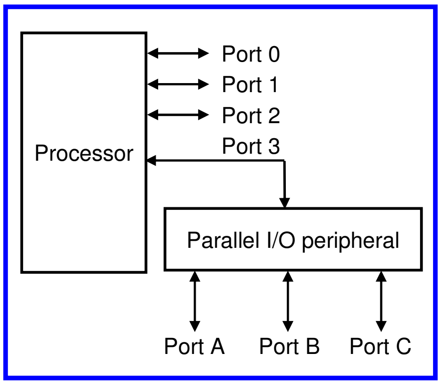
Example: ISA (Industry Standard Architecture) bus protocol

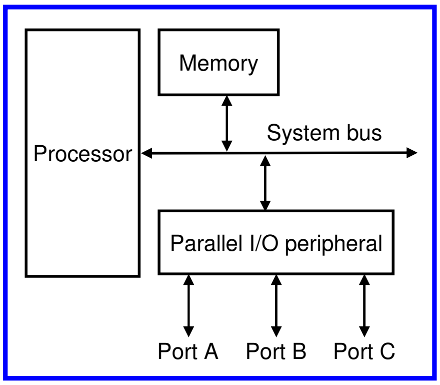
1. Master set REQ to receive data.
2. Slave puts data on bus within time t_{access} (WAIT line is unused).
3. Master receives data and resets REQ.
4. Slave ready for next request.

1. Master sets REQ to receive data.
2. Slave cannot put data on bus within t_{access} and sets WAIT.
3. Slave puts data on bus and resets WAIT.
4. Master receives data and resets REQ.
5. Slave ready for next request.

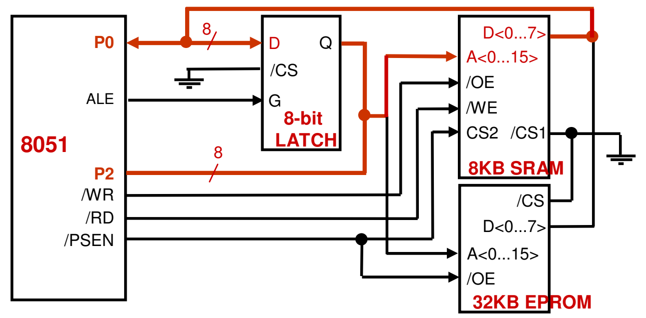
Port-based processor-peripherals communication

- The most common communication in embedded systems is the input & output of data to and from a processor during the communication with peripherals.
- Some processors communicate with the peripherals through ports and not through a system bus. A port is connected to a dedicated register that can be read and written just like any register in the processor (parallel or port-based communication).
- Often, the implementation of an embedded application requires more ports than those available on a particular processor, and then an extended parallel I/O peripheral can be used.
- In other cases, the system may require parallel communication with some peripherals, but the processor may only support a system bus.
- In this case a parallel I/O peripheral can be used (connected to the system bus on one side and has several ports to the other side which are connected to registers inside the peripheral accessible by the processor).

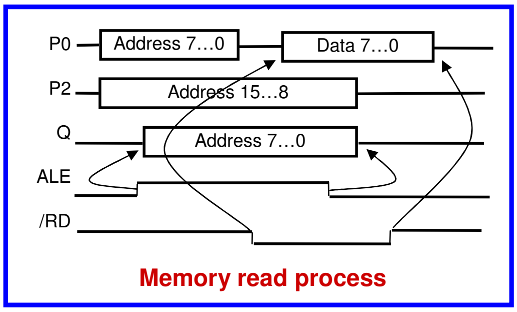




Example of port-based communication



Communication of 8051 (8-bit) controller with memories



Memory read process

- In 8051, the address space is 64KB, thus it is addressable with 16 bits through ports P0 (LSBs) and P2 (MSBs). P0 is also used for the eight LSBs of data.
- The microcontroller places the memory address to be read, on ports P0 and P2.
- P2 holds the 8 address MSBs and retains its value throughout the read operation. P0 holds the 8 address LSBs that is stored inside the 8-bit latch.
- The ALE (address latch enable) signal is used to trigger the latching of P0.
- 8051 sets high impedance on P0 to allow the memory to drive it with the requested data.
- The memory outputs valid data as long as RD is active, and 8051 reads them.

Processor-peripherals communication with bus

- Three main issues regarding the communication of a processor with the peripherals through a system bus are:
  - ✓ The addressing procedure: how the system address map is used in order the processor to communicate with the memory and the peripherals.
  - ✓ The interrupt-driven communication: the processor accepts interrupt signals in order to read and process data from a peripheral.
  - ✓ The direct memory access (DMA) for transferring data between memories and peripherals, without going through the processor.
  - ✓ Arbitration: how to handle simultaneous servicing requests of peripherals.

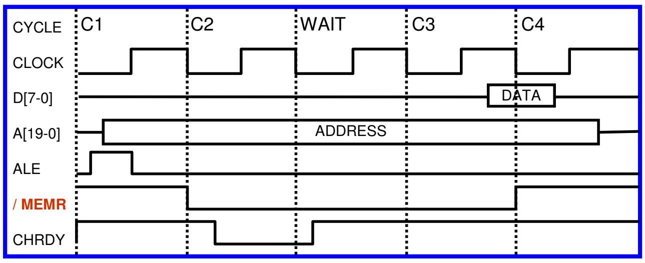
Addressing procedure

- The system bus is a set of wires consisting of address, data and control wires, and the processor uses the bus to access the memory as well as the peripherals.
- The processor may use two methods for communication over a system bus:
  - ✓ The memory-mapped method: the peripherals occupy specific addresses in the existing system address space. For example, in a bus with 16-bit address the lower 32K addresses may correspond to memory addresses, while the upper 32K may correspond to peripheral addresses.
  - ✓ The standard (or I/O-mapped) method: the bus includes an additional line (pin) to indicate whether the processor's access is to memory or to a peripheral. For example, when this specific signal is 0, the address bus corresponds to a memory address, and when the signal is 1, the address corresponds to a peripheral.
- In the memory-mapped method, the processor does not need special instructions for communicating with peripherals, and the assembly instructions involving memory will also work for peripherals (addresses can correspond to memory locations or peripherals').
- In the standard method there is no loss of memory addresses to be used for the peripherals. Also, simpler address decoding logic can be used within the peripherals, since the peripherals' number is smaller than the address space and the high-order address bits can be ignored.

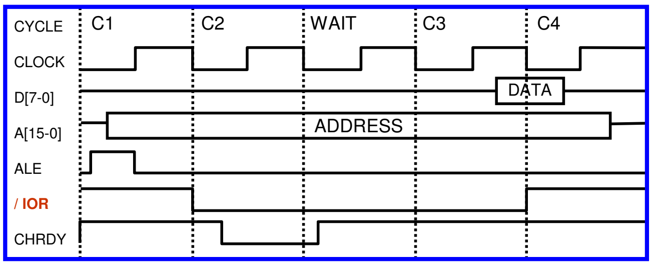
Example of standard (I/O-mapped) method

- ISA bus supports standard or I/O-mapped method:
  - ✓ The signal /IOR is used instead of /MEMR for peripheral read (also there is a signal /IOW for the write operation).
  - ✓ 16-bit address space is used for I/O, while 20-bit address space is used for the memory addressing.
  - ✓ The two protocols (for the communication of the memory and peripherals with the processor) are very similar.

Memory read bus operation



Peripheral read bus operation



Interrupt-driven communication

- Suppose that a peripheral gets new data at unpredictable intervals, which must be serviced by the processor. There are two ways to carry out this procedure:
  - ✓ The processor can check the peripheral regularly to see if data has arrived.
    - This is achieved by interleaving the other tasks of the processor with a routine that checks for new data in a peripheral (perhaps by checking for a true value '1' in a register of the peripheral).
    - Costly in terms of clock cycles, especially in case of many peripherals.
    - The processor could check at less frequent intervals, but then it may not process the data fast enough.
  - ✓ The peripheral can interrupt the processor when it has new data (interrupt-driven I/O communication).
    - The processor requires an extra pin (Int).
    - At the end of each instruction, the processor checks Int and if is set, the processor jumps to a particular address at which a subroutine (ISR) exists that services the interrupt.
    - The checking of the Int pin is performed by the control unit of the processor in parallel with the instruction execution, so no extra cycles are needed.

Interrupt-driven communication (cont'd)

- There are two methods by which a processor using interrupts determines the address at which the interrupt service routine (ISR) is located.

First method: Fixed ISR location

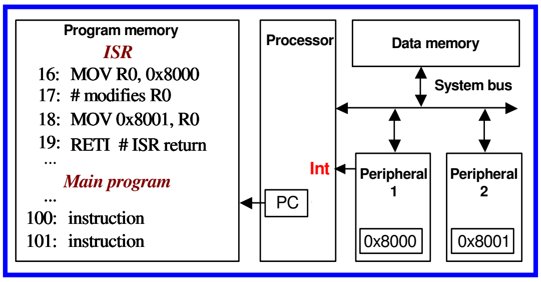
- In some processors, the address to which the processor jumps on an interrupt is fixed. The programmer either puts the ISR there or if not enough bytes are available in that region of memory, puts there a link to the real ISR.

- In processors with fixed ISR addresses, several interrupt pins are needed to support interrupts from multiple peripherals.

Interrupt-driven communication (cont'd)

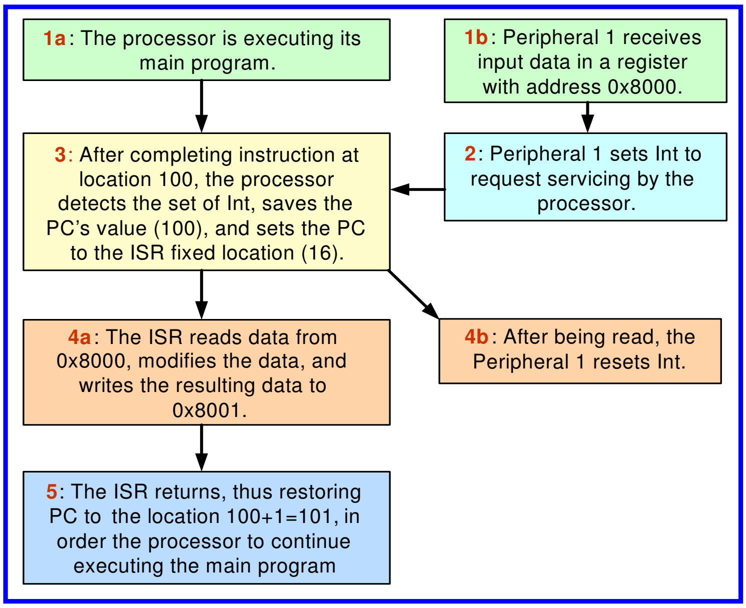
Example

Data received by Peripheral 1 must be read, transformed and then written to Peripheral 2. Peripheral 1 might represent a sensor & Peripheral 2 might represent a display.



Program memory

```
ISR
16:  MOV R0, 0x8000
17:  # modifies R0
18:  MOV 0x8001, R0
19:  RETI  # ISR return
...
Main program
...
100:  instruction
101:  instruction
```



1a: The processor is executing its main program.

1b: Peripheral 1 receives input data in a register with address 0x8000.

2: Peripheral 1 sets Int to request servicing by the processor.

3: After completing instruction at location 100, the processor detects the set of Int, saves the PC's value (100), and sets the PC to the ISR fixed location (16).

4a: The ISR reads data from 0x8000, modifies the data, and writes the resulting data to 0x8001.

4b: After being read, the Peripheral 1 resets Int.

5: The ISR returns, thus restoring PC to the location 100+1=101, in order the processor to continue executing the main program

Interrupt-driven communication (cont'd)

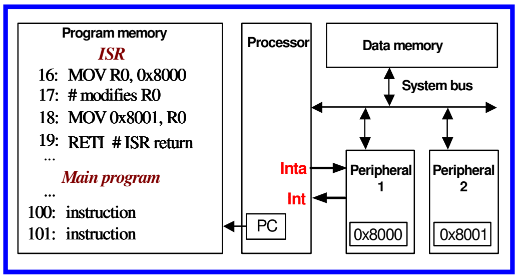
Second method: Use of vectored interrupt to determine the ISR location

- This approach is especially common in systems with a system bus, since there may be numerous peripherals that can request service.
- In this method the processor has one interrupt pin, which any peripheral can set.
- After detecting the interrupt, the processor set another pin to inform the peripheral that it has detected the interrupt and to request from the peripheral to provide the location (address) of the Interrupt Service Routine (ISR).
- The peripheral provides the address of ISR on the data bus, and the microprocessor reads the address and jumps to the ISR.

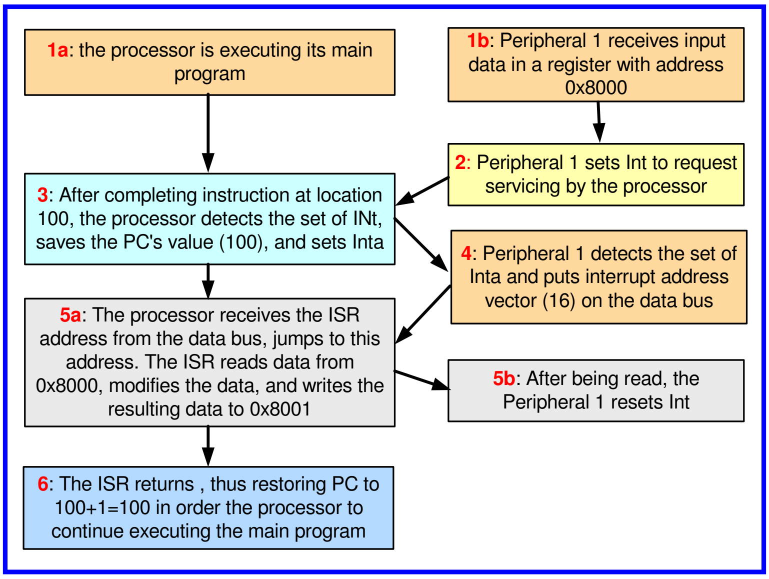
Interrupt-driven communication (cont'd)

Example

Data received by Peripheral 1 must be read, transformed and then written to Peripheral 2. Peripheral 1 might represent a sensor & Peripheral 2 might represent a display.



```
Program memory
ISR
16:  MOV R0, 0x8000
17:  # modifies R0
18:  MOV 0x8001, R0
19:  RETI  # ISR return
...
Main program
...
100:  instruction
101:  instruction
```



1a: the processor is executing its main program

1b: Peripheral 1 receives input data in a register with address 0x8000

2: Peripheral 1 sets Int to request servicing by the processor

3: After completing instruction at location 100, the processor detects the set of INt, saves the PC's value (100), and sets Inta

4: Peripheral 1 detects the set of Inta and puts interrupt address vector (16) on the data bus

5a: The processor receives the ISR address from the data bus, jumps to this address. The ISR reads data from 0x8000, modifies the data, and writes the resulting data to 0x8001

5b: After being read, the Peripheral 1 resets Int

6: The ISR returns , thus restoring PC to 100+1=100 in order the processor to continue executing the main program

Interrupt-driven communication (cont'd)

- As a compromise between fixed and vectored interrupt methods, we can use an interrupt address table.
- In this method , we still have only one interrupt pin on the processor, but we also create in the processor's memory a table that holds ISR addresses (a typical table might have 256 entries).
- A peripheral rather than providing the ISR address, instead provides a number corresponding to an entry in the interrupt address table.
- The processor reads this entry number from the bus, and then reads the corresponding table entry to obtain the ISR address.
- Compared to the memory, the table is very small, so the number of bits required to encode an entry is small, thus reducing the communication complexity.

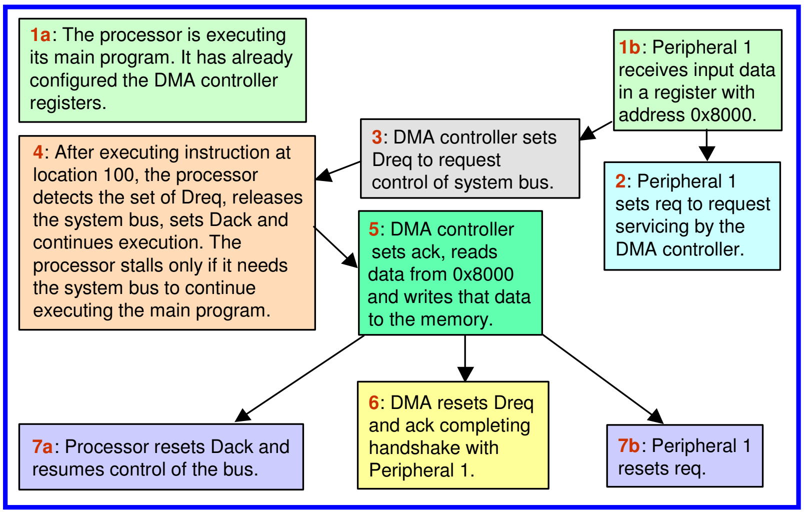
Direct memory access (DMA)

- Commonly, the data being accumulated in a peripheral should be first stored in memory before being processed by a program running on the microprocessor.
- Such temporary storage before processing is called buffering.
- To implement buffering, we could write a simple ISR on the processor such that the peripheral would interrupt the processor whenever it had data to be stored in the memory.
- The ISR will simplify transfer data from the peripheral to the memory, and after that the processor will continue with its regular program.
- However, the jump of the processor to an ISR requires the storing of the processor's state (registers content), and then the restoring of its state when returning from the ISR.
- The storing and restoring procedures and the fact that the processor cannot execute its regular program while moving the registers content, lead to important inefficiency (in terms of consumed clock cycles).
- In order to eliminate such inefficiency, the Direct Memory Access (DMA) method is used.

Direct memory access (cont'd)

- In the DMA method, we use a controller (DMA controller), whose purpose is to transfer data directly between memories and peripherals.
- A peripheral requests servicing from the DMA controller, which then requests to get the control of the system bus from the processor.
- The processor has just to give the control of the bus to the DMA controller and do not need to jump to an ISR and to store and restore its state.
- The processor can execute its regular program while the DMA controller has the control of the bus, as long as the regular program does not require use of the bus (if this occur the processor have to wait for the DMA controller to complete).
- A system with a separate bus between the processor and cache may be able to execute parts of its regular program from the cache while the DMA controller has the control of the system bus.

Direct memory access (cont'd)



1a: The processor is executing its main program. It has already configured the DMA controller registers.

1b: Peripheral 1 receives input data in a register with address 0x8000.

2: Peripheral 1 sets req to request servicing by the DMA controller.

3: DMA controller sets Dreq to request control of system bus.

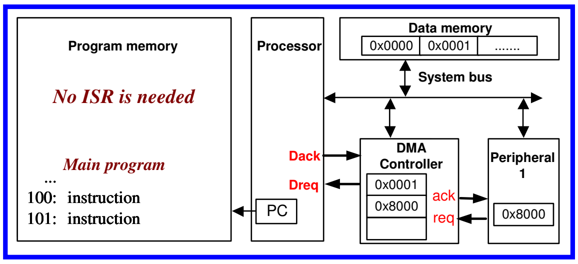
4: After executing instruction at location 100, the processor detects the set of Dreq, releases the system bus, sets Dack and continues execution. The processor stalls only if it needs the system bus to continue executing the main program.

5: DMA controller sets ack, reads data from 0x8000 and writes that data to the memory.

6: DMA resets Dreq and ack completing handshake with Peripheral 1.

7a: Processor resets Dack and resumes control of the bus.

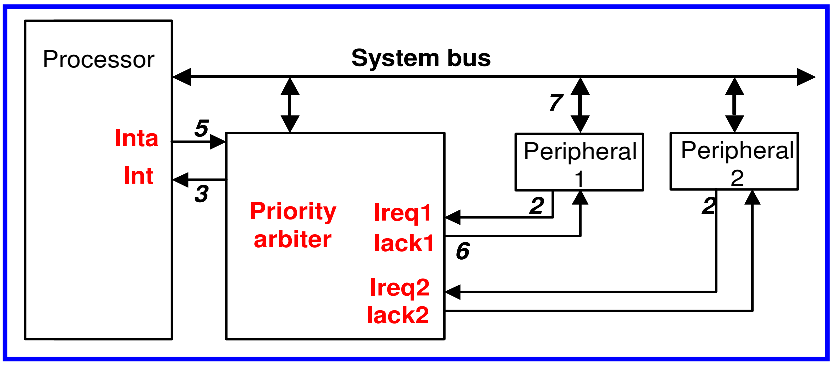
7b: Peripheral 1 resets req.



Arbitration in a system bus

- In embedded systems communication, several situations exist in which multiple peripherals might request service from a single resource (interrupt requests to a processor, DMA requests to a DMA controller etc.).
- It is necessary to have a method to arbitrate among the simultaneous servicing requests of the peripherals, i.e. to decide which one of the peripherals will get service and thus which peripherals need to wait.
- One arbitration method uses a module called priority arbiter: each of the peripherals makes its request to the arbiter and the arbiter send an interrupt to the processor and waits for the interrupt acknowledgment.
- The arbiter then provides an acknowledgment to exactly one peripheral, which permits to that peripheral to put its interrupt vector address on the data bus.
- This causes the processor to jump to a ISR that services that specific peripheral.

Arbitration in a system bus (cont'd)



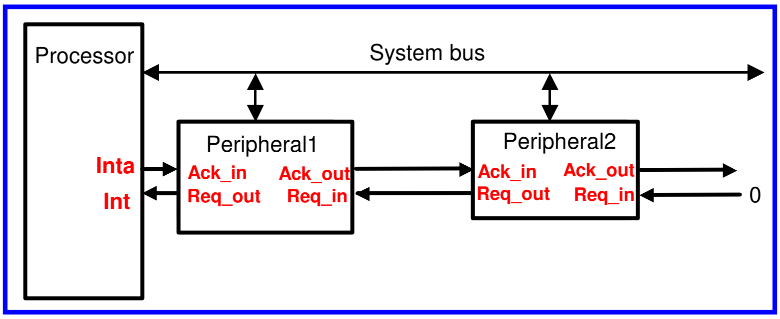
1. Processor is executing its program.
2. Peripheral1 needs service so sets Ireq1. Peripheral2 also needs service & sets Ireq2.
3. Priority arbiter detects that at least one Ireq input is set, so sets Int.
4. Processor stops executing its program and stores its state.
5. Processor sets Inta.
6. Priority arbiter sets Iack1 to acknowledge Peripheral 1.
7. Peripheral 1 puts its interrupt address vector on the system bus.
8. Processor jumps to the address of ISR read from data bus, ISR is executed & returns.
9. Processor resumes executing its program.

Arbitration in a system bus (cont'd)

- Priority arbiters use one of two schemes to determine priority among the peripherals:
  - ✓ In fixed priority arbitration, each peripheral has an unique rank among all the peripherals. The rank can be presented as a number (1, 2, 3 …), so if two peripherals ask for service simultaneously, the arbiter chooses the one with the higher rank.
  - ✓ In rotating priority arbitration (also called round-robin), the arbiter changes priority of peripherals based on the history of servicing of those peripherals. For example, a rotation priority scheme offer service to the least-recently serviced of the requesting peripherals (more complex arbiter functionality).
- We prefer fixed priority when there is a clear difference in priority among peripherals (different nature of peripherals), however, in many cases the peripherals are somewhat of same nature, so by ranking them could case high-ranked peripherals to get much more servicing than low-ranked ones.
- Rotating priority ensures a more fair distribution of servicing in these cases.

Arbitration in a system bus (cont'd)

Daisy-chain arbitration



- In daisy-chain arbitration, arbitration is done by the peripherals.
- Apart from request output and acknowledge input, each peripheral has also a request input and an acknowledge output.
- A peripheral sets its request output if it requires servicing or if its request input is set. The second event means that one of the previous (at the right) peripherals is requesting servicing.
- Thus, if any peripheral needs servicing, its request will flow through the next (at the left) peripherals and eventually will reach the processor.
- The peripheral at the front of the chain (nearest to the processor) has the highest priority.

Arbitration in a system bus (cont'd)

- We prefer the daisy-chain arbitration method over a priority arbiter, when we want to be able to add or remove peripherals from an embedded system without redesigning the system.
- Although, we can add many peripherals to a daisy chain, in reality the servicing response time of the peripherals at the end of the chain will become very slow.
- In contrast of a daisy-chain scheme, a priority arbiter has a fixed number of channels (the system needs redesign in order to accommodate more peripherals).
- However, the daisy-chain method has the drawback of not supporting more advanced priority schemes, like rotating priority.
- In addition, if a peripheral in the chain stops working, other peripherals may lose their access to the processor.

Arbitration in a system bus (cont'd)

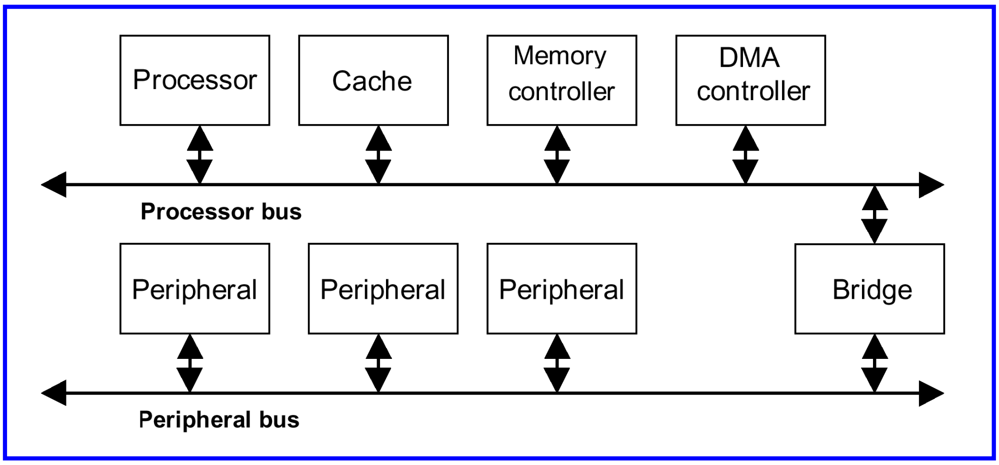
Network arbitration methods

- Many embedded systems contain multiple processors communicating via a shared bus (sometimes is called network).
- Arbitration in such cases is built into the bus protocol.
- A processor that wants to write to the bus has no way to know whether another processor tries to simultaneously write to the bus. These situations lead to collisions and cause data on the bus to be corrupted, and thus the data must be resent to the bus.
- The processors detect this collision, stop transmitting data, wait for some time and then try to transmit again.
- The bus protocol must ensure that the requesting processors do not start sending data again at the same time or must use statistical methods that make small the chances of them sending again at the same time.
- Another solution is the use an address encoding scheme such that if two addresses are written simultaneously by different processors using a bus, the highest-priority address will overwrite the lower-priority one.

Multi-level bus architectures

- A processor-based system can have numerous types of communications that must take place, varying in their speed requirements.
- The most frequent and high-speed communications are between the processor and its memories, while less frequent communications requiring less speed are between the processor and its peripherals (e.g. UART).
- We can implement a single high-speed bus for all communications, but this requires each peripheral to have a high-speed bus interface.
- Since a peripheral may not need such high-speed communication, having such an interface will lead in extra gates, energy and cost.
- In addition, having too many peripherals on a single bus may result in a slower bus.
- Systems are often designed with two levels of buses: a high-speed processor bus, and a lower-speed peripheral bus.
- The processor bus typically connects the processor, cache memory, memory controllers and high-speed co-processors (memory-word size).
- The peripheral bus connects lower-speed peripherals and emphasize to the portability and the low energy consumption of the peripherals.

Multi-level bus architectures (cont'd)



- A bridge connects the two buses: custom processing block that converts communication on the processor bus to communication on the peripherals bus and vice versa.
- For example, the processor may generate a read on the processor bus with an address corresponding to a peripheral. The bridge detects that the address corresponds to a peripheral and generates a read on the peripheral bus. After receiving the data, the bridge sends them to the processor.

Serial vs. parallel communication

- Serial communication: the bus transports one bit of data at a time, through a single data wire.
  - ✓ Offers high data throughput for long distances.
  - ✓ It needs less wiring, exhibits less capacitance resulting in low power consumption and it is cheaper.
  - ✓ It requires more complex interfacing logic and protocol: the sender needs to decompose the data words into bits, and the receiver needs to recompose the data bits into words. In addition, the control signals often are sent on the same wire with data, increasing protocol complexity.
- Parallel communication: the bus is capable to transport multiple bits of data at a time (multiple wires, one bit per wire).
  - ✓ Offers high data throughput for short distances.
  - ✓ Typically, it is used when connecting devices on the same IC or on the same board.
  - ✓ It needs more wiring and it is more expensive.
  - ✓ In case of long wires the result will be high capacitances (charge/discharge delays and high power consumption).

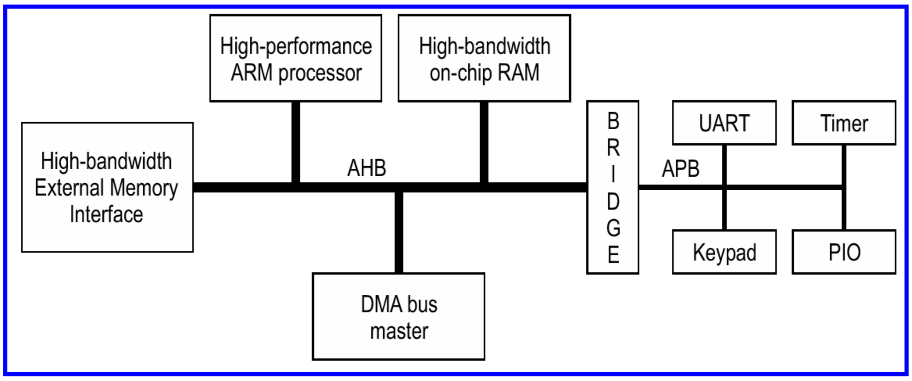
Example of serial communication: I2C bus

- I2C (Inter-IC) bus by Philips Semiconductors:
  - ✓ Two-wire serial bus protocol.
  - ✓ Enables peripheral ICs to communicate using simple communication hardware.
  - ✓ Data transfer rates up to 100 Kbits/s and 7-bit addressing possible in normal mode.
  - ✓ 3.4 Mbits/s and 10-bit addressing in fast-mode.
  - ✓ Common devices capable of interfacing to I2C bus: EPROMS, Flash, RAM memory modules, timers and microcontrollers.

Example of parallel communication: PCI bus

- PCI Bus (Peripheral Component Interconnect) by Intel:
  - ✓ High performance parallel bus.
  - ✓ Standard widely adopted by industry.
  - ✓ Interconnects chips, expansion boards, processor memory subsystems.
  - ✓ Data transfer rates of 127.2 to 508.6 Mbits/s and 32-bit addressing.
  - ✓ It has already been extended to 64-bit while maintaining compatibility with 32-bit schemes.
  - ✓ Synchronous bus architecture.
  - ✓ Multiplexed data/address lines.

Example of parallel communication: AMBA bus


High-performance ARM processor
High-bandwidth on-chip RAM
High-bandwidth External Memory Interface
AHB
DMA bus master
BRIDGE
APB
UART
Timer
Keypad
PIO

- AMBA (by ARM) is a multi-level (parallel) bus, mainly used for system-on-chip (SoC) designs.
- Provides two buses connected through a bridge:
  - ✓ A high-speed system bus (AHB) to connect processors, high-performance peripherals, DMA controller and on-chip memories.
  - ✓ A low-speed peripheral bus (APB) that follows a simpler protocol to connect timers, general-purpose (non-critical) peripherals, and serial interfaces.

Conclusions

- Communication of processors with memories and other peripherals in an embedded system is an important design task.
- A system bus may support the following types of data transfers: memory to/from processor, I/O to/from processor, and I/O to/from memory (DMA).
- A system bus includes data, address and control lines.
- There are three basic communication control methods: strobe, handshake and compromise.
- Apart from communication of a processor with its peripherals through a system bus, there can be also communication through ports (port-based or parallel communication).
- Main issues regarding the communication through a system bus are: the addressing procedure, the interrupt-driven communication, the direct memory access (DMA) and the arbitration method.
- Advanced communication schemes include: parallel and serial communication architectures as well as multi-level bus architectures.