Basic Processing Unit

UNIT II

Overview

- Instruction Set Processor (ISP) or Central Processing Unit (CPU) – the unit which executes instructions and coordinates the activities of the other units
- A typical computing task consists of a series of steps specified by a sequence of machine instructions that constitute a program.
- An instruction is executed by carrying out a sequence of more fundamental operations.

Some Fundamental Concepts

Fundamental Concepts

- Processor fetches one instruction at a time and perform the operation specified.
- Instructions are fetched from successive memory locations until a branch or a jump instruction is encountered.
- Processor keeps track of the address of the memory location containing the next instruction to be fetched using Program Counter (PC).
- Instruction Register (IR) – holds the instruction fetched from memory

Executing an Instruction

- Fetch the contents of the memory location pointed to by the PC. The contents of this location are loaded into the IR (fetch phase).

$$IR \leftarrow [[PC]]$$

- Assuming that an instruction is 4bytes long, increment the contents of the PC by 4 to point to the next instruction. (fetch phase)

$$PC \leftarrow [PC] + 4$$

- Carry out the actions specified by the instruction in the IR (execution phase).

Processor Organization

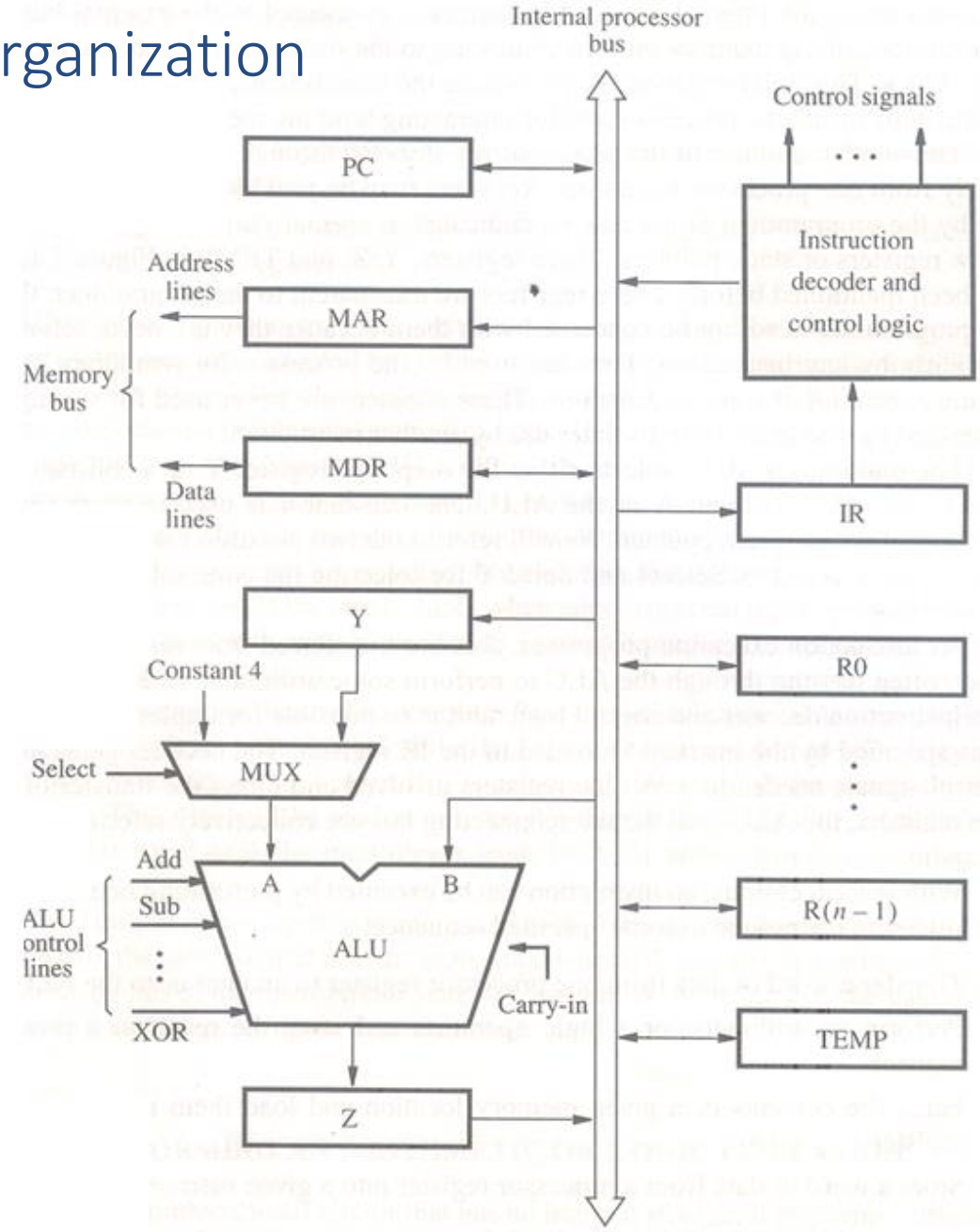


Figure 7.1 Single-bus organization of the datapath inside a processor.

Internal organization of the processor

- ALU – Used to perform arithmetic and logical operation.
- Registers for temporary storage
- Various digital circuits for executing different micro operations.(gates, MUX, decoders, counters).
- Internal path for movement of data between ALU and registers.
- Driver circuits for transmitting signals to external units.
- Receiver circuits for incoming signals from external units.

- PC:
  - Keeps track of execution of a program
  - Contains the memory address of the next instruction to be fetched and executed.

- MAR:
  - Holds the address of the location to be accessed.
  - I/p of MAR is connected to Internal bus and an O/p to external bus.

- MDR:
  - Contains data to be written into or read out of the addressed location.
  - It has 2 inputs and 2 Outputs.
  - Data can be loaded into MDR either from memory bus or from internal processor bus.

- The data and address lines are connected to the internal bus via MDR and MAR

Registers:

- ❖The processor registers R0 to Rn-1 vary considerably from one processor to another.
- ❖Registers are provided for general purpose, used by programmer.
- ❖Special purpose registers-index & stack registers.
- ❖Registers Y,Z &TEMP are temporary registers used by processor during the execution of some instruction.

Multiplexer:

- ❖Select either the output of the register Y or a constant value 4 to be provided as input A of the ALU.
- ❖Constant 4 is used by the processor to increment the contents of PC.

Data Path:

- The registers, ALU and interconnecting bus are collectively referred to as the data path.

Executing an Instruction

- Execution of an instruction involves one or more of the following operations:

  - **Register Transfers:** Transfer a word of data from one processor register to another or to the ALU.
  - **ALU operations:** Perform an arithmetic or a logic operation and store the result in a processor register.
  - **Load operation**: Fetch the contents of a given memory location and **load** them into a processor register.
  - **Store operation:** Store a word of data from a processor register into a given memory location.

Register Transfers

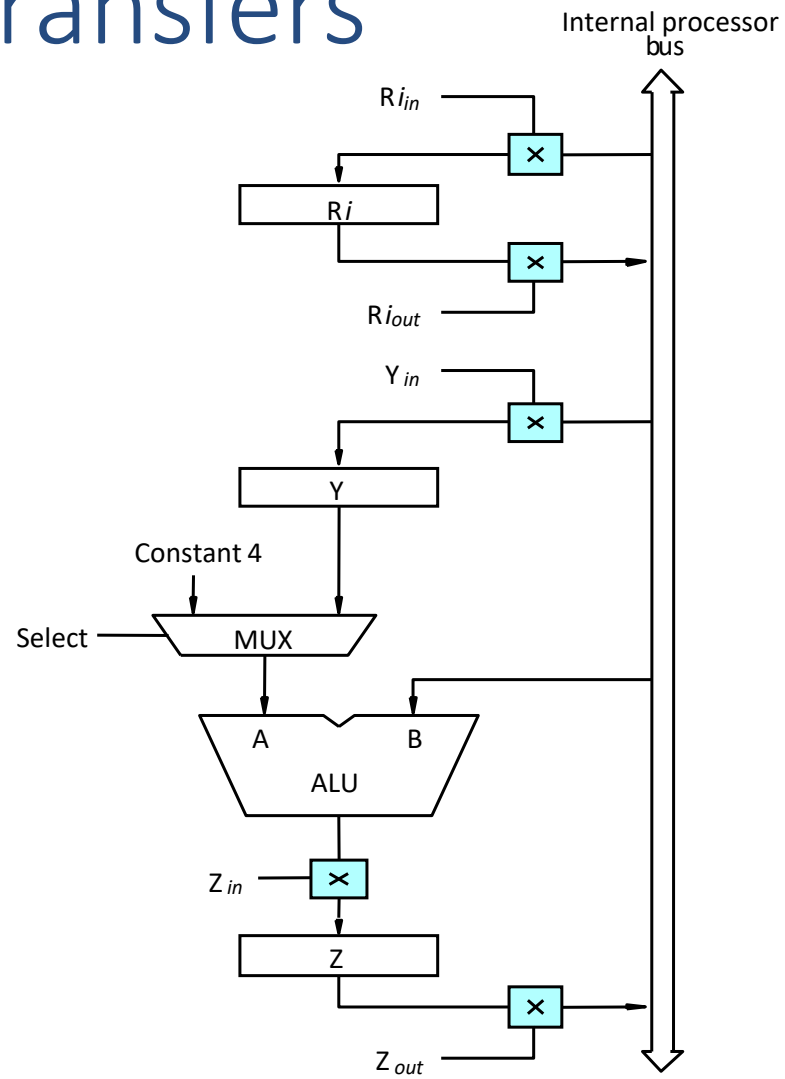


Figure 7.2. Input and output gating for the registers in Figure 7.1.

- The input and output gates for register Ri are controlled by signals Ri_{in} and Ri_{out} .

- Ri_{in} is set to1 – data available on internal bus are loaded into Ri.

- Ri_{out} is set to1 – the contents of register are placed on the bus.

Data transfer between two registers:

EX:

Transfer the contents of R1 to R4.

1. Enable output of register R1 by setting R1out=1. This places the contents of R1 on the processor bus.

2. Enable input of register R4 by setting R4in=1. This loads the data from the processor bus into register R4.

Performing an Arithmetic or Logic Operation

- The ALU is a combinational circuit that has no internal storage.
- ALU gets the two operands from MUX and bus. The result is temporarily stored in register Z.
- What is the sequence of operations to add the contents of register R1 to those of R2 and store the result in R3? i.e. Add R1, R2, R3
    1. R1out, Yin
    2. R2out, SelectY, Add, Zin
    3. Zout, R3in

Processor Organization

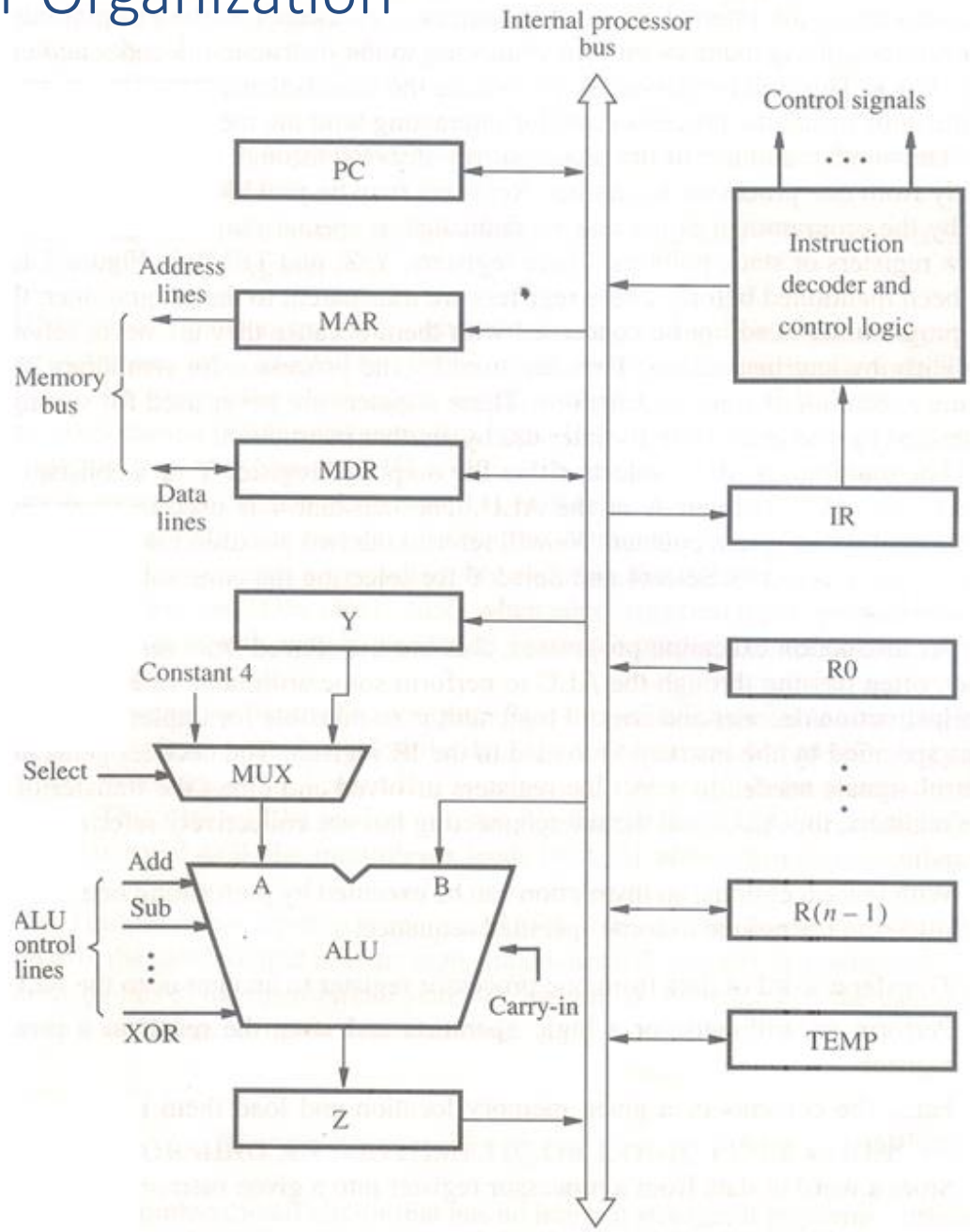


Figure 7.1 Single-bus organization of the datapath inside a processor

Add R1, R2, R3

1.R1out, Yin

2.R2out, SelectY, Add, Zin

3.Zout, R3in

- Step 1: Output of the register R1 and input of the register Y are enabled, causing the contents of R1 to be transferred to Y.

- Step 2: The multiplexer's select signal is set to select Y causing the multiplexer to gate the contents of register Y to input A of the ALU.

- Step 3: The contents of Z are transferred to the destination register R3.

Load Operation - Fetching a Word from Memory

- Address into MAR; issue Read operation; data into MDR.

Figure 7.4. Connection and control signals for register MDR.

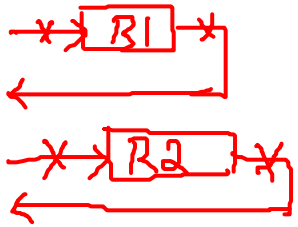
Fetching a Word from Memory

- The response time of each memory access varies (cache miss)
- To accommodate this, the processor waits until it receives an indication that the requested operation has been completed (Memory-Function-Completed, MFC).
- Move (R1), R2
  - MAR ← [R1]
  - Start a Read operation on the memory bus
  - Wait for the MFC response from the memory
  - Load MDR from the memory bus
  - R2 ← [MDR]

R1out, MARin, Read

MDRinE,WMFC

MDRout, R2in



Timing

MAR ← [R1]

Assume MAR
is always available
on the address lines
of the memory bus.

Start a Read operation on the memory bus

Wait for the MFC response from the memory

Load MDR from the memory bus

R2 ← [MDR]

Storing a word in memory

- Move R2,(R1)

1. R1out, MARin
2. R2out, MDRin, Write
3. MDRoutE, WMFC

Execution of a Complete Instruction

- Add (R3), R1
- Fetch the instruction
- Fetch the first operand (the contents of the memory location pointed to by R3)
- Perform the addition
- Store the result into R1

Execution of a Complete Instruction

Add (R3), R1

Execution of Branch Instructions

- A branch instruction replaces the contents of PC with the branch target address, which is usually obtained by adding an offset X given in the branch instruction.
- The offset X is usually the difference between the branch target address and the address immediately following the branch instruction.
- Unconditional branch – Jump instructions

Execution of unconditional Branch Instruction

Step	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Offset-field-of-IR_{out}, Select Y, Add, Z_{in}
5	Z_{out}, PC_{in}, End

Figure 7.7(a). Control sequence for an unconditional branch instruction.

Execution of conditional Branch Instruction

Step	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Offset-field-of-IR_{out}, Select Y, Add, Z_{in}, If N=0 then End
5	Z_{out}, PC_{in}, End

Figure 7.7(b). Control sequence for an unconditional branch instruction.

If N=1 then branch is taken; execute an instruction from branch target.
N=0 branch is not taken; execute a sequentially next instruction.

Multiple-Bus Organization

- General purpose registers are combined into a single block called register file.
- Two output ports allow the contents of two different registers to be accessed simultaneously and have their contents placed on buses A and B.
- Third port allows the data on bus C to be loaded into a third register during the same clock cycle.
- Bus A & B are used to transfer the source operands to A & B inputs of the ALU.
- The result of ALU operation is transferred to the destination over the bus C.

- ALU may simply pass one of its 2 input operands unmodified to bus C.
- The ALU control signals for such an operation R=A or R=B.
- Incrementer unit is used to increment the PC by 4.
- Using the incrementer eliminates the need to add the constant value 4 to the PC using the main ALU.
- The source for the constant 4 at the ALU input multiplexer can be used to increment other address such as *loadmultiple* & *storemultiple*

Multiple-Bus Organization

- Add R4, R5, R6

Control sequence for the instruction Add R4,R5,R6, on three-bus organization

Step	Action
1	PC_{out}, R=B, MAR_{in}, Read, IncPC
2	WMF C
3	MDR_{outB}, R=B, IR_{in}
4	$R4_{outA}$, $R5_{outB}$, SelectA, Add, $R6_{in}$, End

ADD (R3), R2, R1
i.e. R1← (R3) + [R2]

- PCout, R=B, MARin, Read, IncPC
- WMFC
- MDRoutB, R=B, IRin
- R3outB, R=B, MARin, Read
- WMFC
- MDRoutB, R2outA,Add, R1in, End

Quiz

- What is the control sequence for execution of the instruction

  Add R1, R2

  including the instruction fetch phase? (Assume single bus architecture)

On single bus (R2← [R1] + [R2])

1. **PCout, MARin, Read, Select4, Add, Zin**
2. **Zout, PCin, Yin, WMFC**
3. **MDRout, Irin**
4. **R1out, Yin**
5. **R2out, Add, Zin**
6. **Zout, R2in, END**

Hardwired Control

Overview

- To execute instructions, the processor must have some means of generating the control signals needed in the proper sequence.
- Computer designers use a wide variety of technology to solve this problem
- This approach fall in two categories: hardwired control and microprogrammed control

- Hardwired system can operate at high speed; but with little flexibility.
- The control unit uses a fixed logic circuit to interpret instructions and generate sequence of control signals from them.
- Each steps in this sequence is completed in one clock cycle.
- A counter may be used to keep the track of the control steps (refer fig 7.10)
- Each count of this counter corresponds to one control step

The required control signals are determined by the following information.

1. contents of the control step counter
2. contents of the instruction register
3. contents of the condition code flags
4. External input signals such as MFC and interrupt request.

Hardwired Control Unit Organization

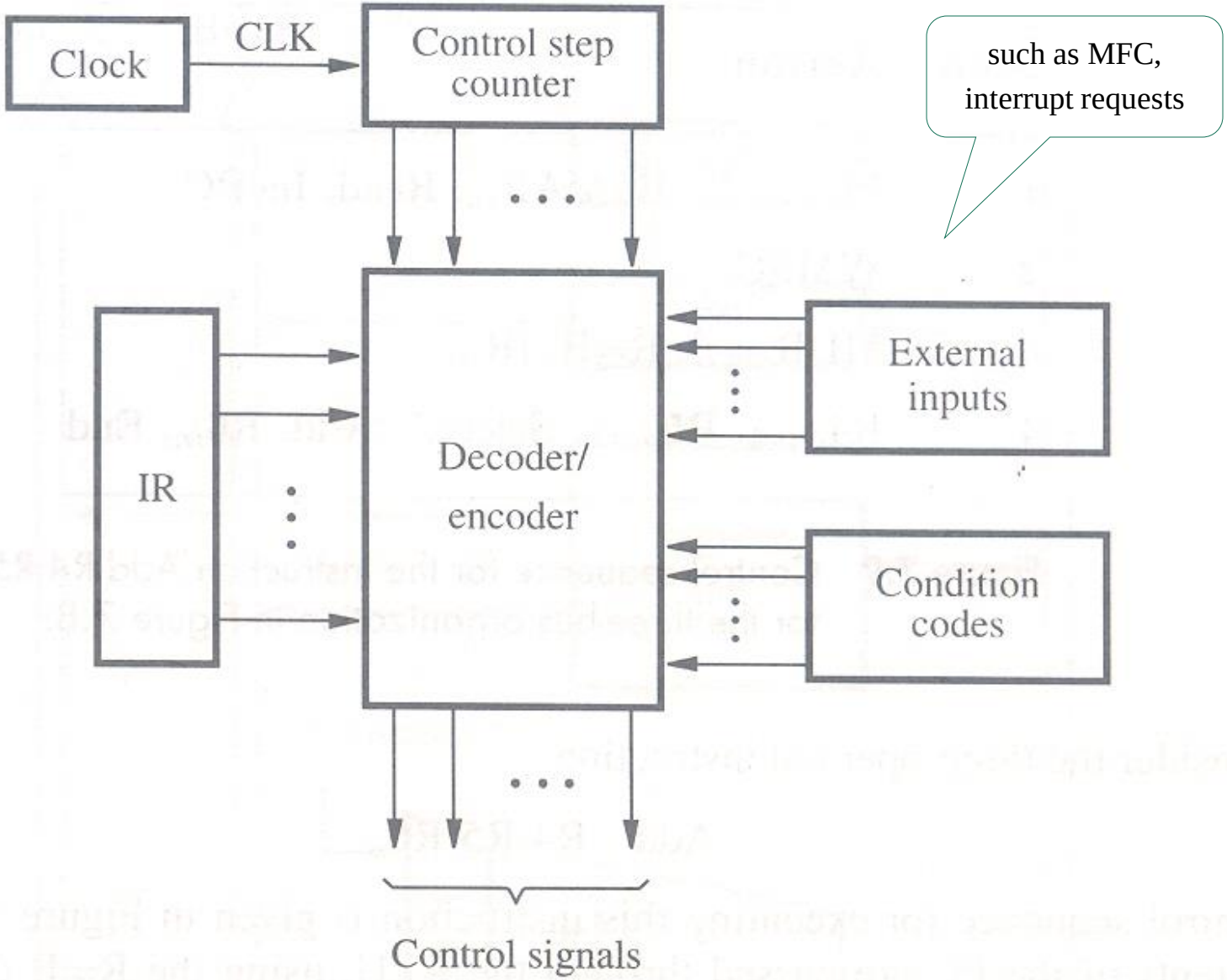


Figure 7.10 Control unit organization.

- The decoder/encoder block diagram is a combinational circuit that generate the required control outputs, depending on the state of all its inputs.
- In the figure 7.11 the decoding and encoding functions are separated.

- The step decoder provides a separate signal line for each step, in the control sequence.

- The output of the instruction decoder consists of a separate line for each machine instruction.
- For any instruction loaded in the IR, one of the output lines INS_1 through INS_m is set to 1, and all other lines are set to 0.
- The encoder generate appropriate control signals y_{in}, pc_{out} , Add, End and so on by combining the input signals.

Detailed Block Diagram

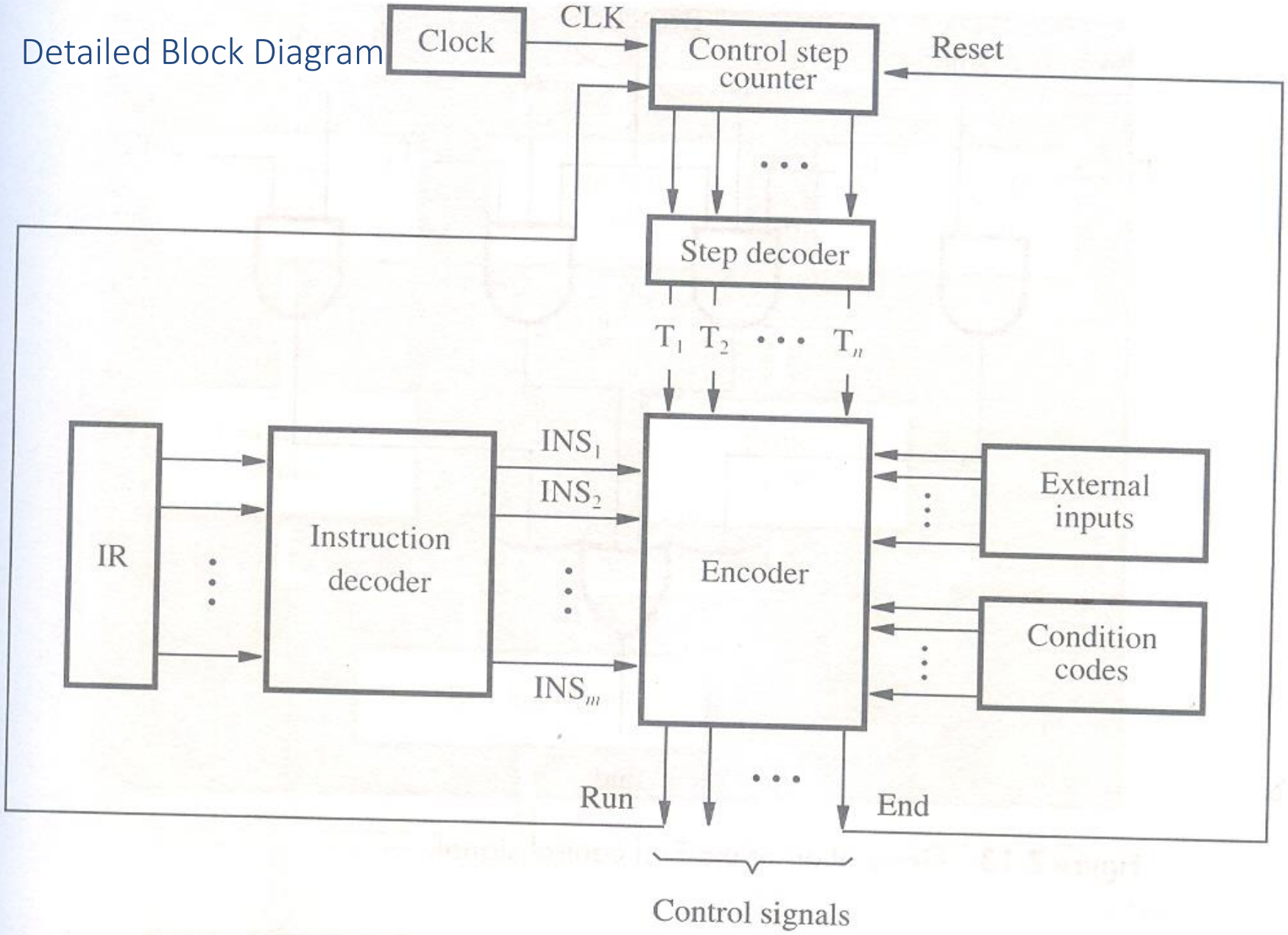
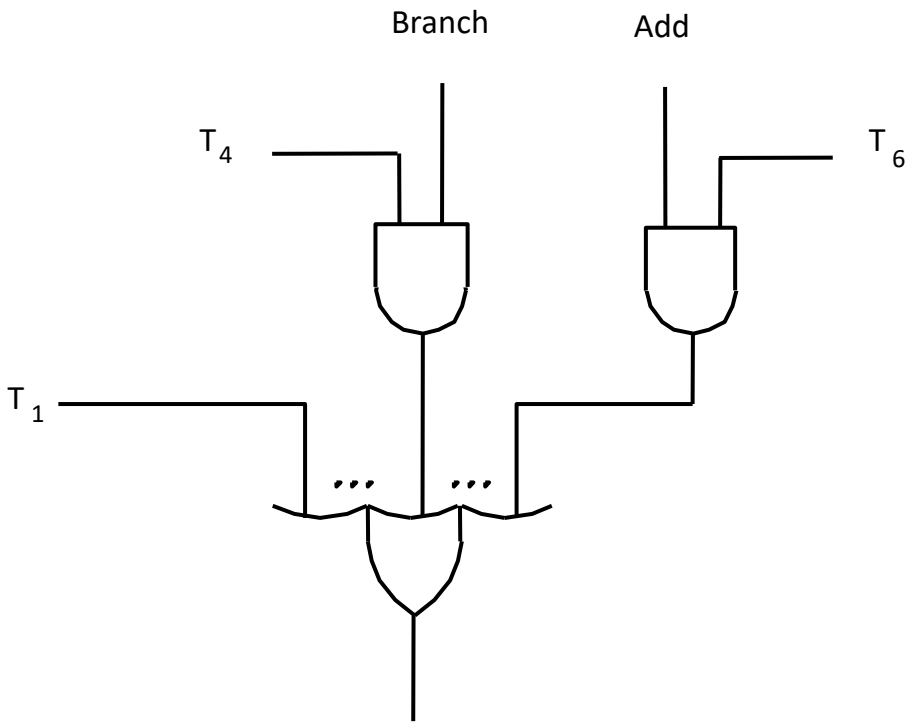


Figure 7.11 Separation of the decoding and encoding functions.

Example: Generation of Z_{in} by the encoder in a signal for the single-bus processor

- $Z_{in} = T_1 + T_6 \bullet ADD + T_4 \bullet BR + \ldots$



Zin is asserted during time slot T1 for all instructions, during T6 for an Add instruction, during T4 for an unconditional branch instruction, and son

Control signals for Add (R3), R1

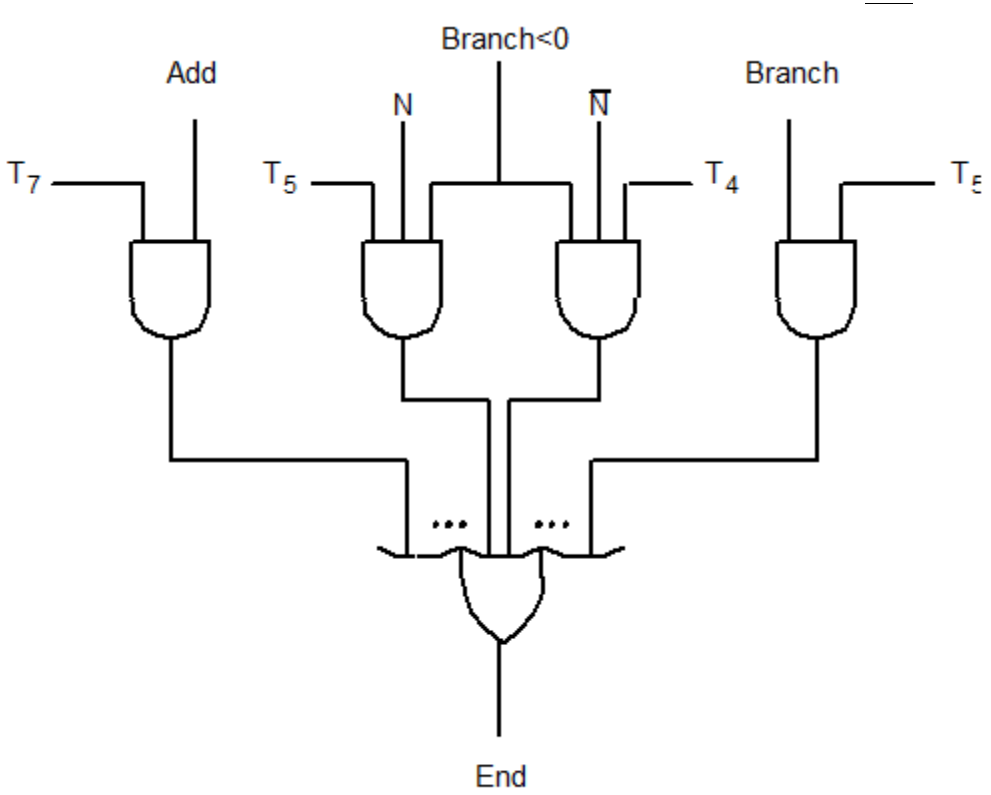
Step	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMF C
3	MDR_{out}, IR_{in}
4	$R3_{out}$, MAR_{in}, Read
5	$R1_{out}$, Y_{in}, WMF C
6	MDR_{out}, SelectY, Add, Z_{in}
7	Z_{out}, $R1_{in}$, End

Control signals for Conditional branch

Step	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMF C
3	MDR_{out}, IR_{in}
4	Offset-field-of-IR $_{out}$, Select Y, Add, Z_{in}, If N=0 then End
5	Z_{out}, PC_{in}, End

Generating End control signal

- End = $T_7 \bullet ADD + T_5 \bullet BR + (T_5 \bullet N + T_4 \bullet N') \bullet BRN + \ldots$



The END signals starts a new instruction fetch cycle by resetting the control step counter to its starting value.

- RUN, set to 1, causes the counter to be incremented by 1 at the end of every clock cycle, when it set to 0 the counter stop counting. This is needed whenever the WMFC signal is issued.

A Complete Processor

Microprogrammed Control

Overview

- Control signals are generated by a program similar to machine language programs.
- Control Word (CW); microroutine; microinstruction

Overview

- **Control Word** (CW) is a word whose individual bits represent various control signals.
- Every instruction will need a sequence of CWs for its execution.
- At every step, some control signals are asserted (=1) and all others are 0
- Sequence of CWs for an instruction forms the **microroutine** for that instruction.
- Each CW in this microroutine is referred to as a **microinstruction**.

- Every instruction will have its own microroutine which is made up of microinstructions.
- Microroutines for all instructions in the instruction set of a computer are stored in a special memory called **Control Store.**
- Control signals are generated by sequentially reading the CWs of the corresponding microroutine from the control store.

Basic organization of a microprogrammed control unit

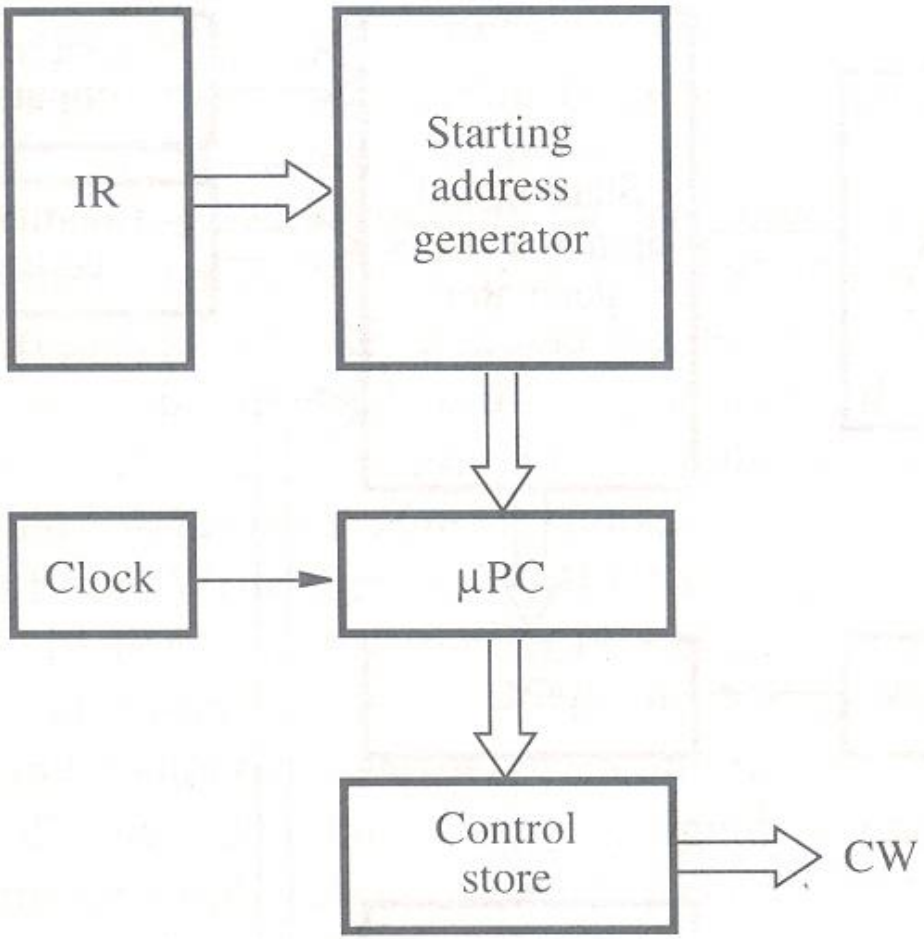


Figure 7.16 Basic organization of a microprogrammed control unit.

- Microprogram counter (mPC) is used to read CWs from control store sequentially.
- When a new instruction is loaded into IR, starting address generator generates the starting address of the microroutine.
- This address is loaded into the mPC. mPC is automatically incremented by the clock, so successive microinstructions are read from the control store.

Execution of conditional Branch Instruction

Step	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMF C
3	MDR_{out}, IR_{in}
4	Offset-field-of-IR_{out}, Add, Select Y, Z_{in}, If N=0 then End
5	Z_{out}, PC_{in}, End

Figure 7.7(b). Control sequence for an conditional branch instruction.

If N=1 then branch is taken; execute an instruction from branch target.
N=0 branch is not taken; execute a sequentially next instruction.

Overview

- The previous organization cannot handle the situation when the control unit is required to check the status of the condition codes or external inputs to choose between alternative courses of action.
- Use conditional branch microinstruction.

Address	Microinstruction
0	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
1	Z_{out}, PC_{in}, Y_{in}, WMF C
2	MDR_{out}, IR_{in}
3	Branch to starting address of appropriate microroutine
. . . . .	
25	If N=0, then branch to microinstruction 0
26	Offset-field-of-IR_{out}, SelectY, Add, Z_{in}
27	Z_{out}, PC_{in}, End

Figure 7.17. Microroutine for the instruction Branch<0.

- Control words at location 25, 26, and 27 implent the microroutine for branch<0
- Control word at location 25 tests the N bit of the condition code register
- If this bit is 0 branch takes place to location 0 to fetch new instruction
- Otherwise control word at location 26 is executed, to fetch the instruction from branch target address.

Overview

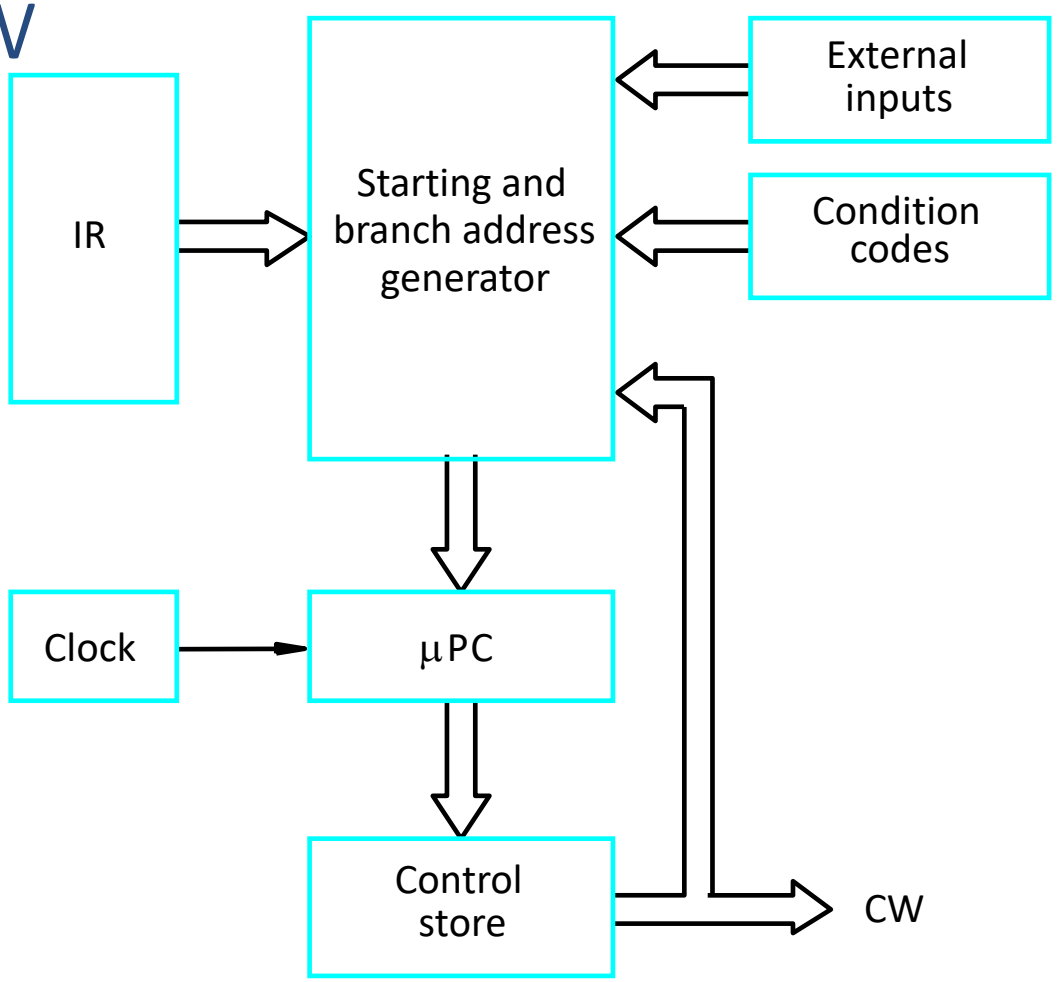


Figure 7.18. Organization of the control unit to allow conditional branching in the microprogram.

Comparison of hardwired and microprogrammed control

Attribute	Hardwired Control	Microprogrammed Control
Speed	Fast	Slow
Control functions	Implemented in hardware	Implemented in software
Flexibility	Not flexible to accommodate new system specifications or new instructions	More flexible, to accommodate new system specification or new instructions redesign is required
Ability to handle large/complex instruction sets	Difficult	Easier
Ability to support operating systems and diagnostic features	Very difficult	Easy
Design process	Complicated	Orderly and systematic
Applications	Mostly RISC microprocessors	Mainframes, some microprocessors
Instructionset size	Usually under 100 instructions	Usually over 100 instructions
ROM size	-	2K to 10K by 20-400 bit microinstructions
Chip area efficiency	Uses least area	Uses more area

Thank you

Pipelining

Overview

- Pipelining is widely used in modern processors.
- Pipelining improves system performance in terms of throughput.
- Pipelined organization requires sophisticated compilation techniques.

Basic Concepts

Making the Execution of Programs Faster

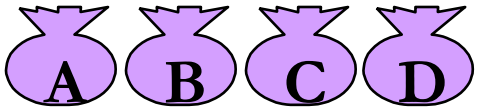
- To increase the speed of execution either
  - Use faster circuit technology to build the processor and the main memory

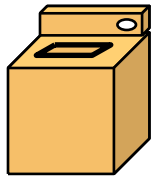
    or
  - Arrange the hardware so that more than one operation can be performed at the same time.

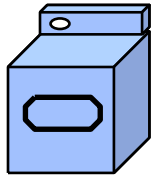
- In the latter way, the number of operations performed per second is increased even though the elapsed time needed to perform any one operation is not changed.

Traditional Pipeline Concept

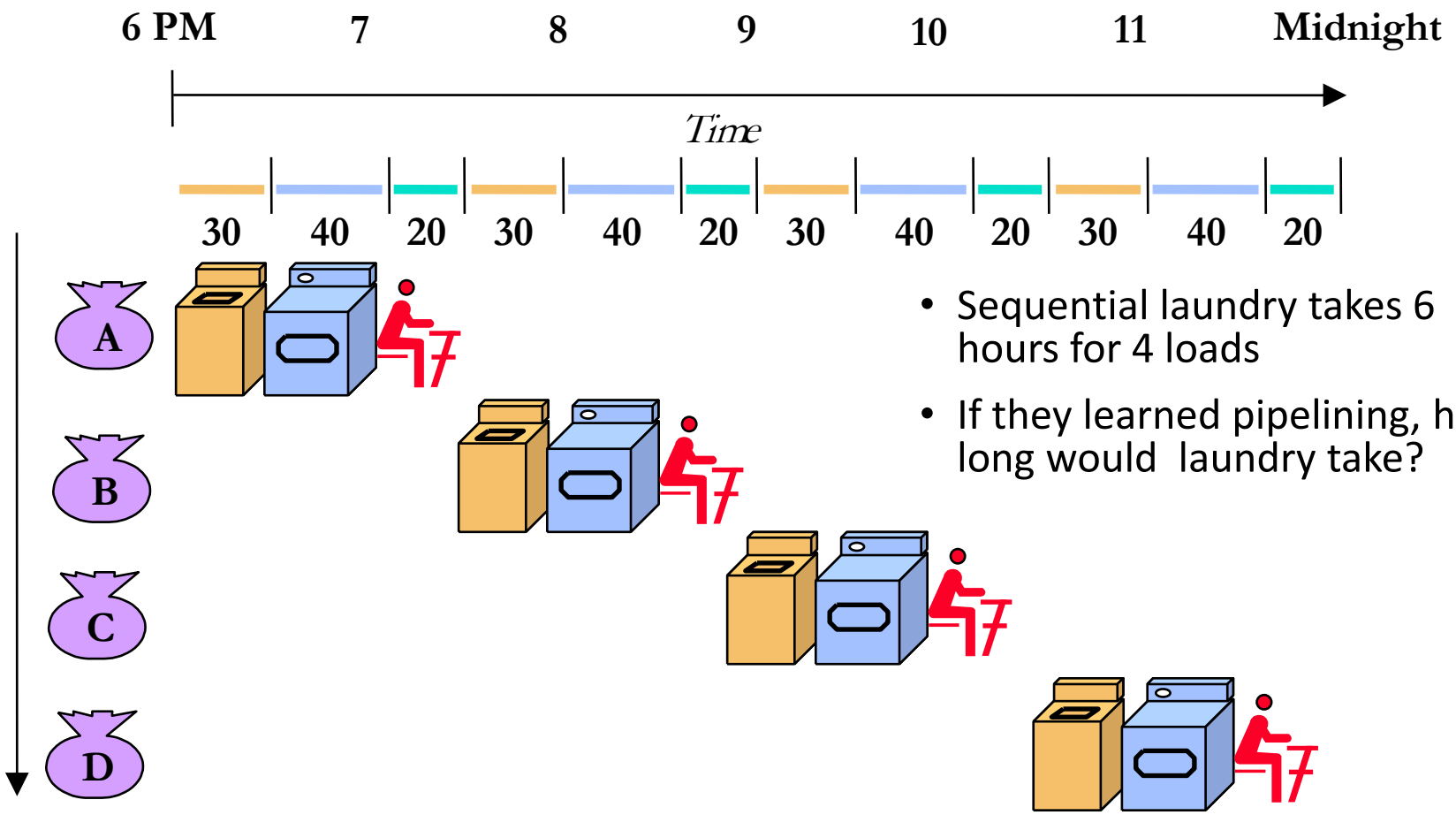
- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 40 minutes
- “Folder” takes 20 minutes





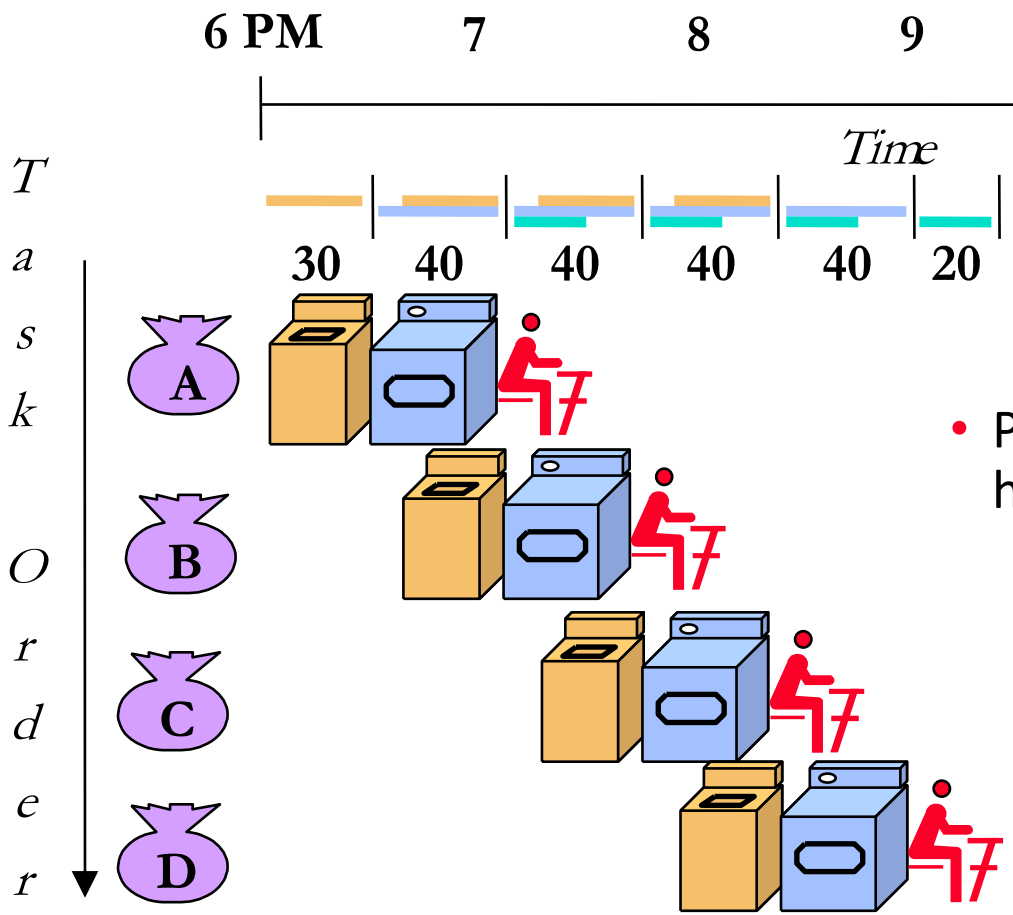


Traditional Pipeline Concept



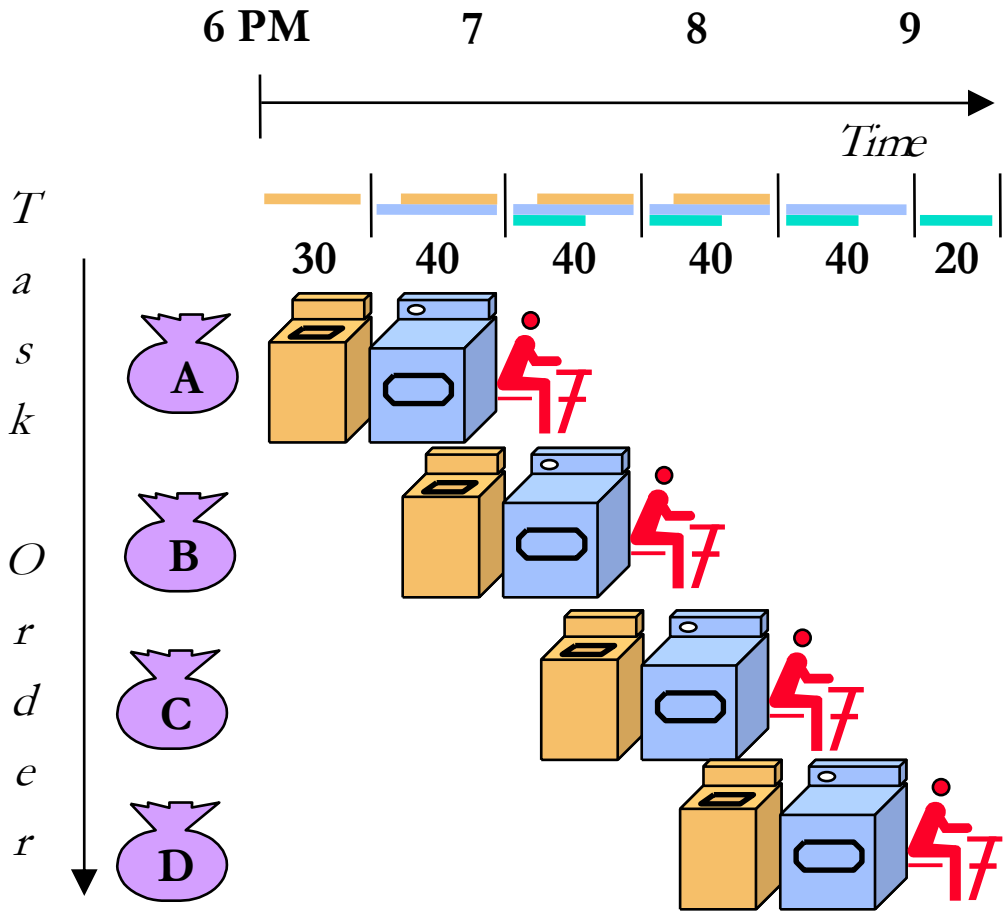
- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Traditional Pipeline Concept



- Pipelined laundry takes 3.5 hours for 4 loads

Traditional Pipeline Concept



- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Stall for Dependences

Use the Idea of Pipelining in a Computer

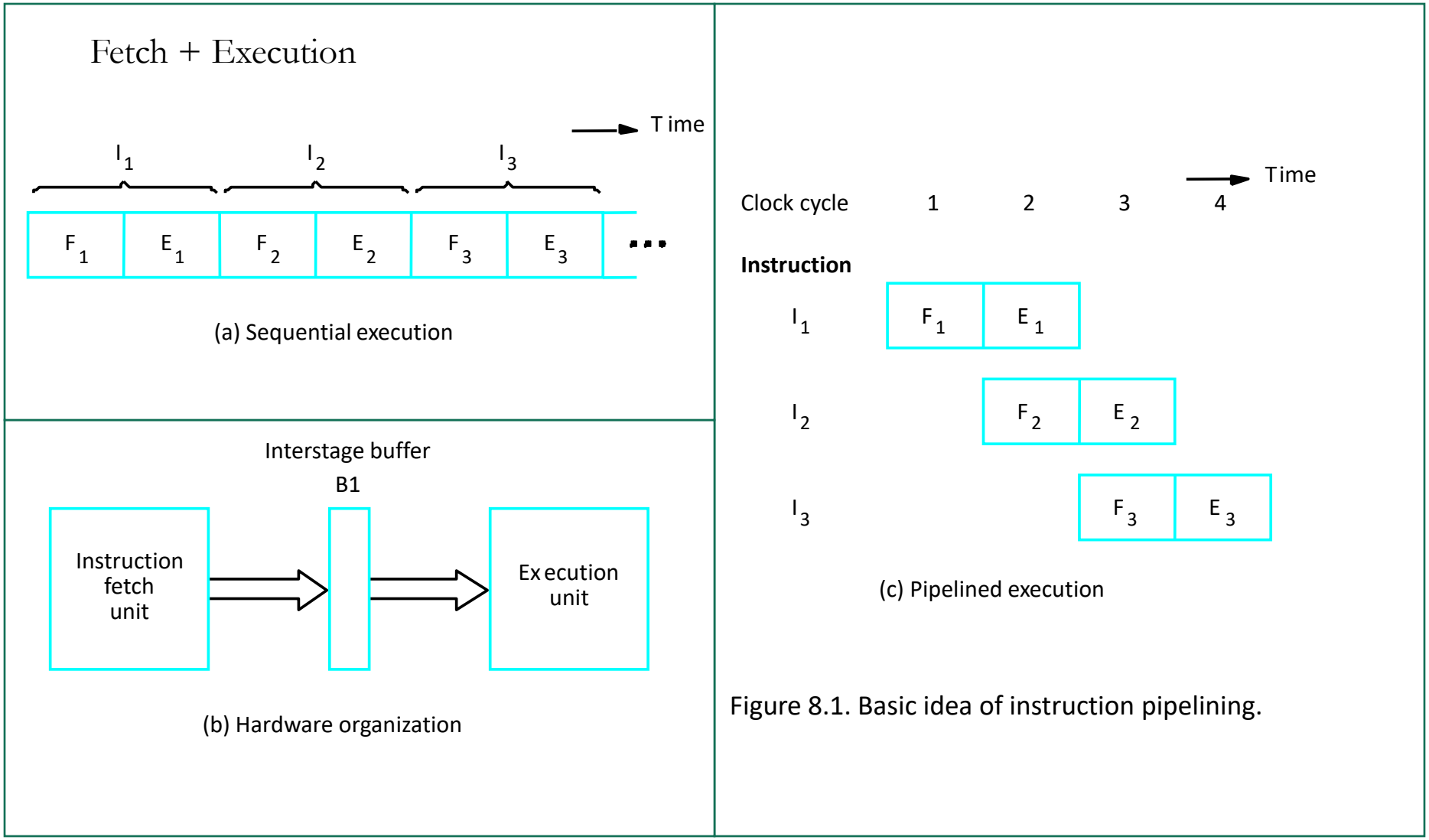


Figure 8.1. Basic idea of instruction pipelining.

Use the Idea of Pipelining in a Computer

Fetch + Decode
+ Execution + Write

Textbook page: 457

Role of Cache Memory

- Each pipeline stage is expected to complete in one clock cycle.
- The clock period should be long enough to let the slowest pipeline stage to complete.
- Faster stages can only wait for the slowest one to complete.
- Since main memory is very slow compared to the execution, if each instruction needs to be fetched from main memory, pipeline is almost useless.
- Fortunately, we have cache.

Pipeline Performance

- The potential increase in performance resulting from pipelining is proportional to the number of pipeline stages.
- However, this increase would be achieved only if all pipeline stages require the same time to complete, and there is no interruption throughout program execution.
- Unfortunately, this is not true.

Pipeline Performance

Pipeline Performance

- The previous pipeline is said to have been stalled for two clock cycles.
- Any condition that causes a pipeline to stall is called a hazard.
- Data hazard – any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. So some operation has to be delayed, and the pipeline stalls.
- Instruction (control) hazard – a delay in the availability of an instruction causes the pipeline to stall.
- Structural hazard – the situation when two instructions require the use of a given hardware resource at the same time.

Pipeline Performance

Instruction hazard

Idle periods –
stalls (bubbles)

Pipeline Performance

Structural hazard

Load X(R1), R2

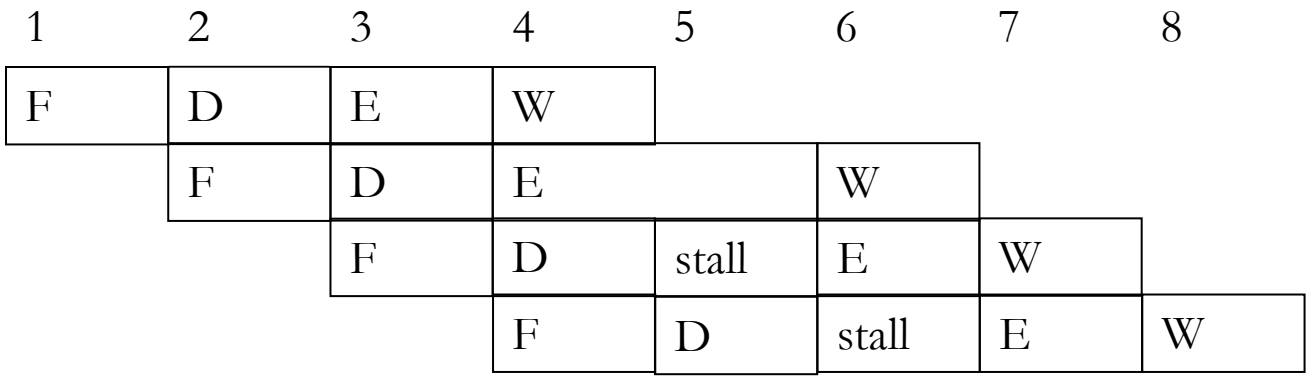
Pipeline Performance

- Again, pipelining does not result in individual instructions being executed faster; rather, it is the throughput that increases.
- Throughput is measured by the rate at which instruction execution is completed.
- Pipeline stall causes degradation in pipeline performance.
- We need to identify all hazards that may cause the pipeline to stall and to find ways to minimize their impact.

Quiz

- Four instructions, the I2 takes two clock cycles for execution. Draw the figure for 4-stage pipeline, and figure out the total cycles needed for the four instructions to complete.

Solution



Data Hazards

- We must ensure that the results obtained when instructions are executed in a pipelined processor are identical to those obtained when the same instructions are executed sequentially.
- Hazard occurs

  $A \leftarrow 3 + A$

  $B \leftarrow 4 \times A$
- No hazard

  $A \leftarrow 5 \times C$

  $B \leftarrow 20 + C$
- When two operations depend on each other, they must be executed sequentially in the correct order.
- Another example:

  Mul R2, R3, R4

  Add R5, R4, R6

Data Hazards

Figure 8.6. Pipeline stalled by data dependency between D_2 and W_1.

Operand Forwarding

- Instead of from the register file, the second instruction can get data directly from the output of ALU after the previous instruction is completed.
- A special arrangement needs to be made to “forward” the output of ALU to the input of ALU.

Handling Data Hazards in Software

- Let the compiler detect and handle the hazard:

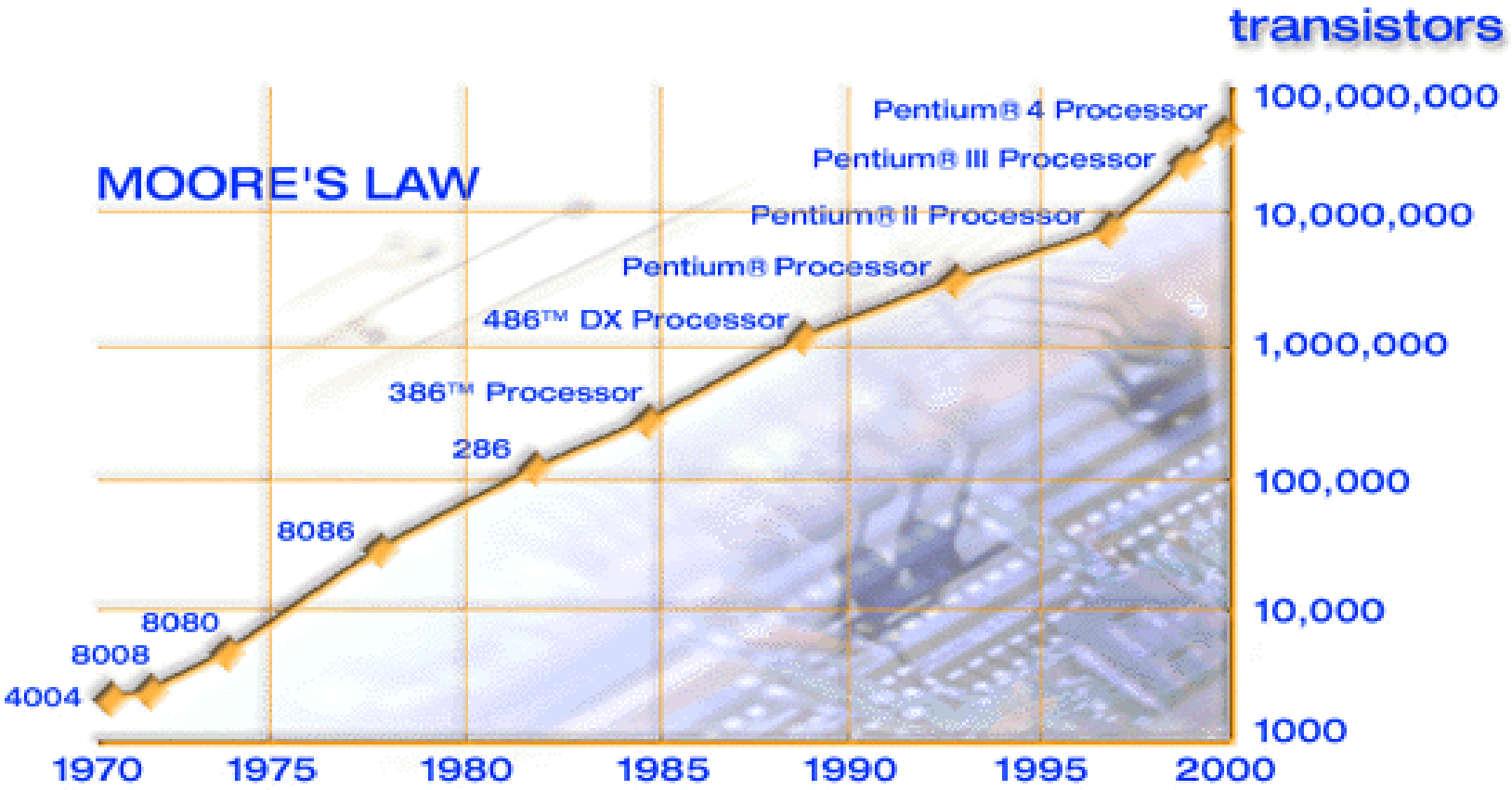
```
I1: Mul  R2, R3, R4
    NOP
    NOP
I2: Add  R5, R4, R6
```

- The compiler can reorder the instructions to perform some useful work during the NOP slots.

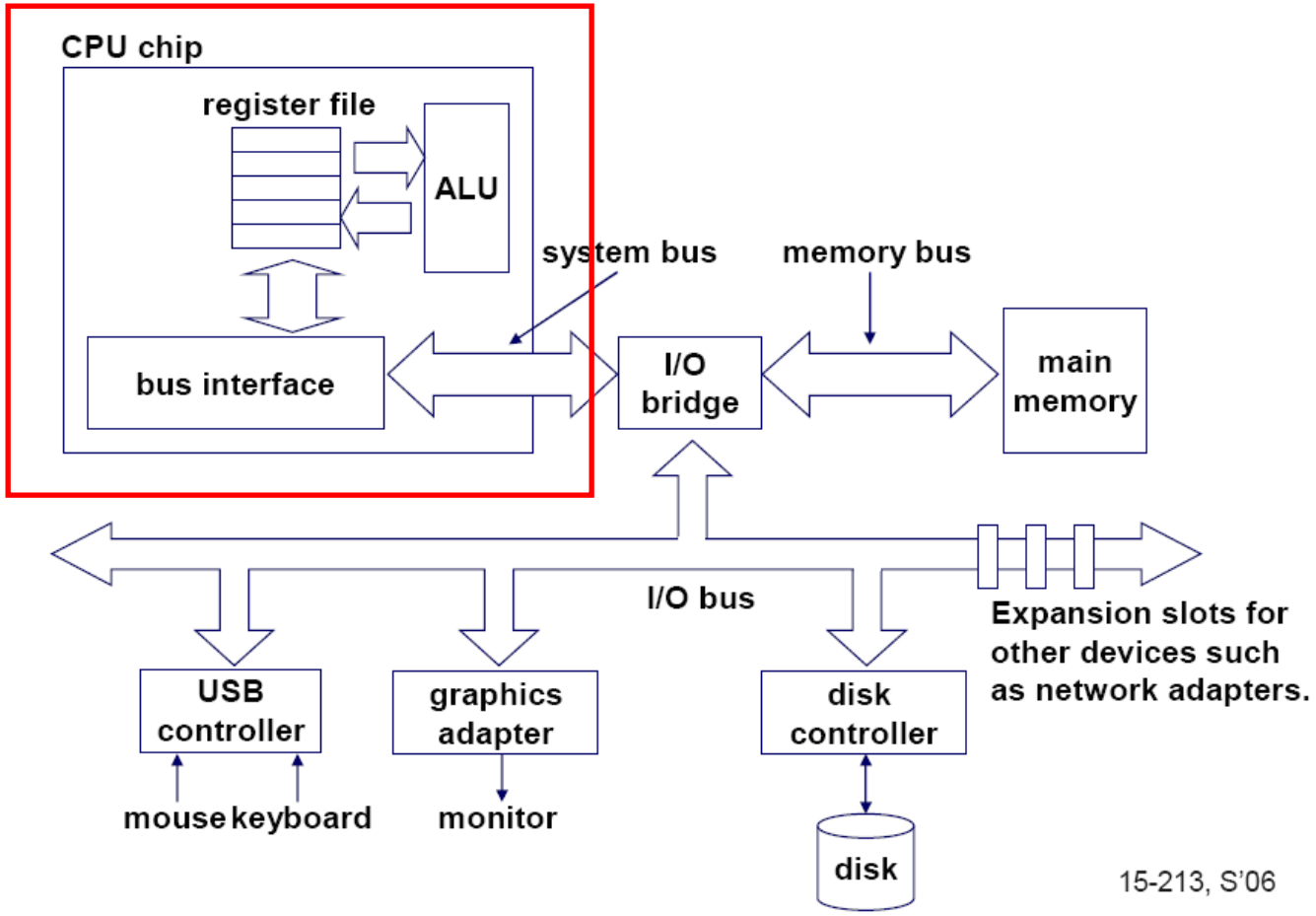
Moore's Law

- Gordon Moore, cofounder of Intel noticed a trend in IC manufacture
- **Every 2 years the number of components on an area of silicon doubled**
- He published this work in 1965 – known as Moore's Law
- His predictions were for 10 years into the future
- His work predicted personal computers and fast telecommunication networks

Graph of Moore's Law



Single-core computer



Limits

- Both physical and practical reasons pose significant constraints to simply building ever faster serial computers:
  - Transmission speeds - the speed of a serial computer is directly dependent upon how fast data can move through hardware. Absolute limits are the speed of light (30 cm/nanosecond) and the transmission limit of copper wire (9 cm/nanosecond). Increasing speeds necessitate increasing proximity of processing elements.
  - Limits to miniaturization - processor technology is allowing an increasing number of transistors to be placed on a chip. However, even with molecular or atomic-level components, a limit will be reached on how small components can be.
  - Economic limitations - it is increasingly expensive to make a single processor faster. Using a larger number of moderately fast commodity processors to achieve the same (or better) performance is less expensive.

Limits

- Relation btwn frequency of operation and power consumption is given by :

$$\text{Power} = \tfrac{1}{2} \times \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency}$$

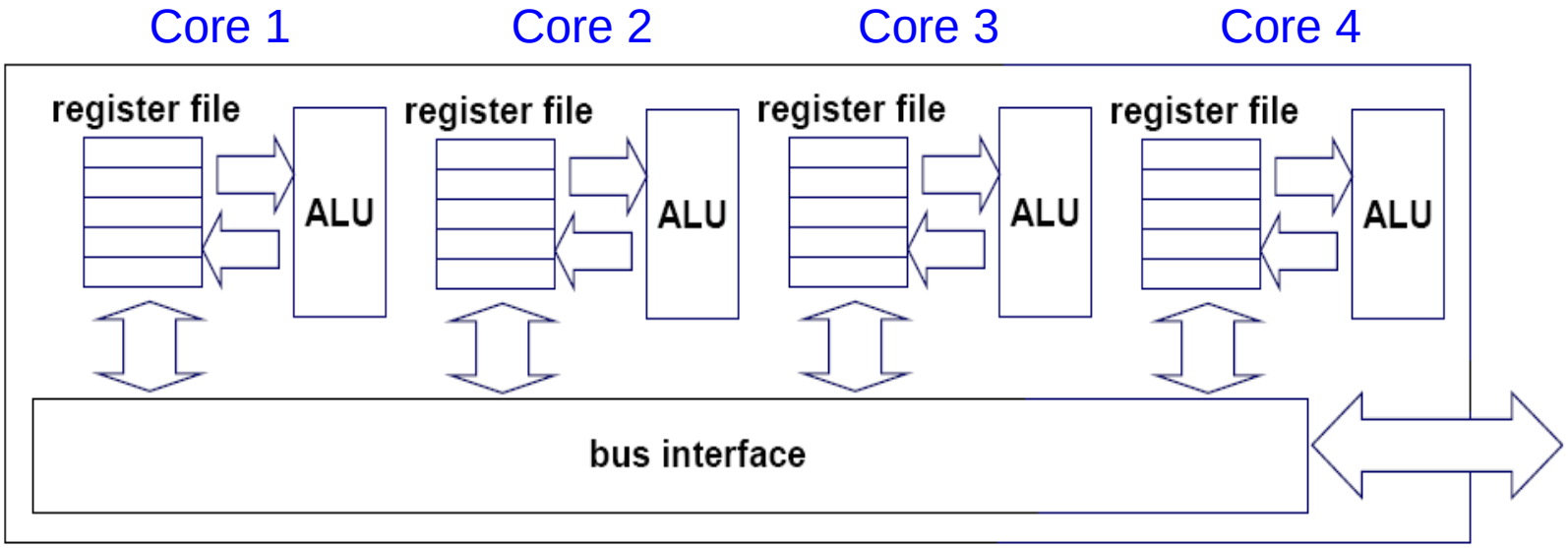
- Power consumption is a huge problem when frequency is increased
- More is the power consumed, more will be the heat generated
- Power consumption poses significant constraint for simply building ever faster serial computers

Why multi-core ?

- Difficult to make single-core clock frequencies even higher
- Deeply pipelined circuits:
  - heat problems
  - speed of light problems
  - difficult design and verification
  - large design teams necessary
  - server farms need expensive air-conditioning
- Many new applications are multithreaded
- General trend in computer architecture (shift towards more parallelism)

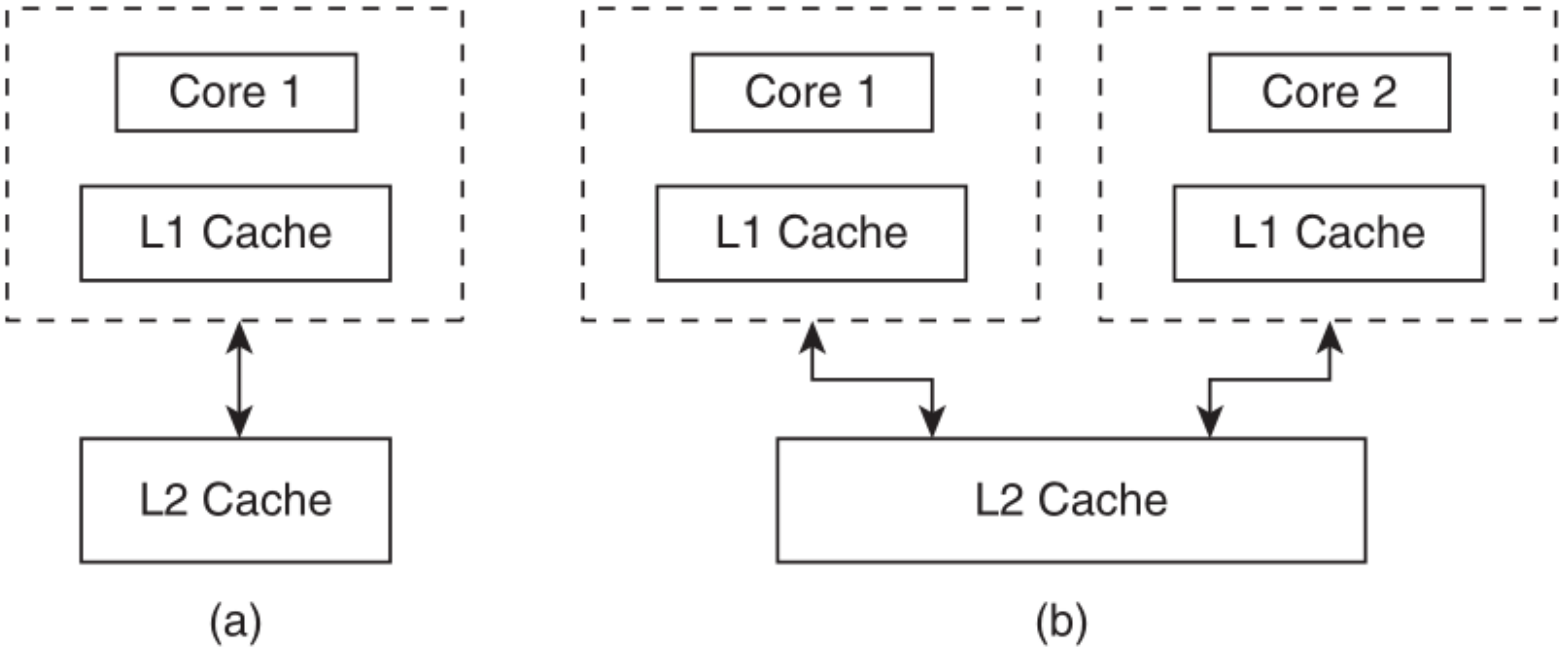
Multi-core architectures

- Replicate multiple processor cores of moderate speed on a single die.



Multi-core CPU chip

Unicore vs. Multicore



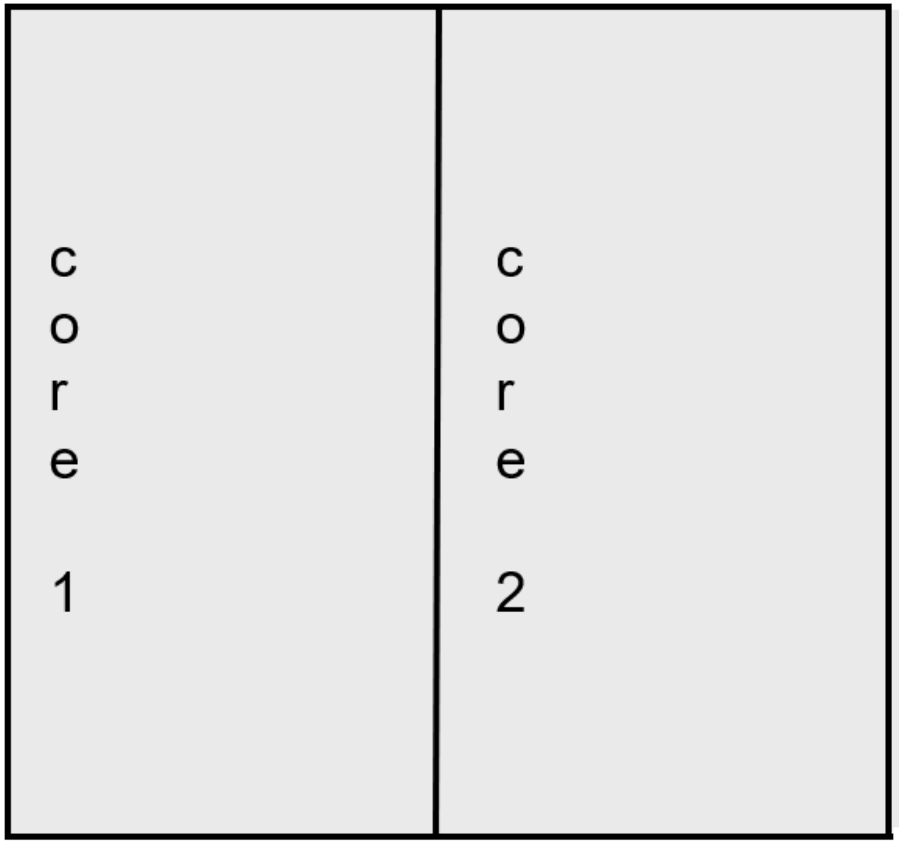
(a) Uni-core processor; (b) multicore processor.

Conventional processor
- Single core
- Dedicated caches
- One thread at a time

Multicore processors
- At least two cores
- Shared caches
- Many threads simultaneously

Multi-core CPU chip

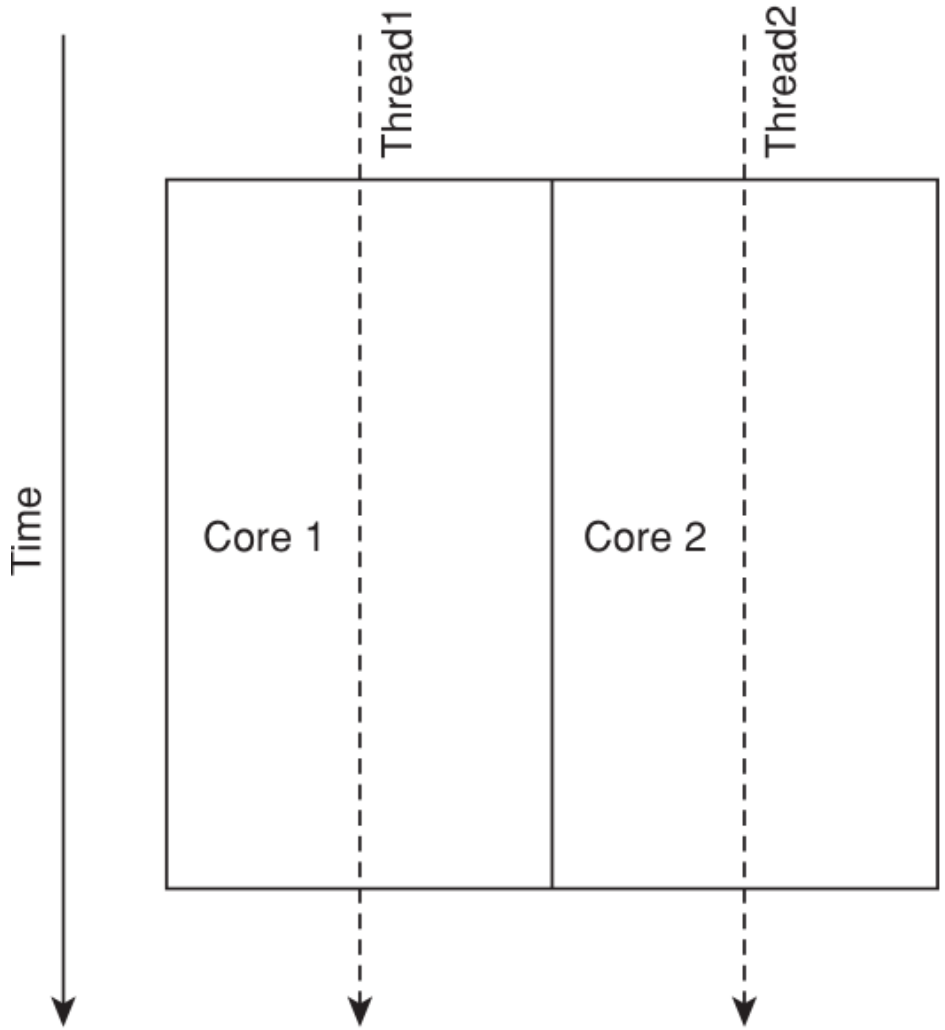
- The cores fit on a single processor socket



What is a thread?

- A process is a program in execution.
- A process is also known as heavy weight process
- A process is made to contain many lightweight processes or threads.
- Each thread performs a subtask of the problem to be solved.
- Each thread contains private local data and all threads share a common address space.
- Hence, the overhead of thread creation is lesser compared to the creation of heavyweight processes.
- Also, since threads share resources of the process to which they belong, the resources are utilized optimally.
- Above all, threads communicate with each other through shared memory locations
- Hence the communication is faster and the associated overhead is lesser compared to that of processes.

Thread level parallelism – multiple threads run on separate cores in parallel



Within each core, threads can be time-sliced