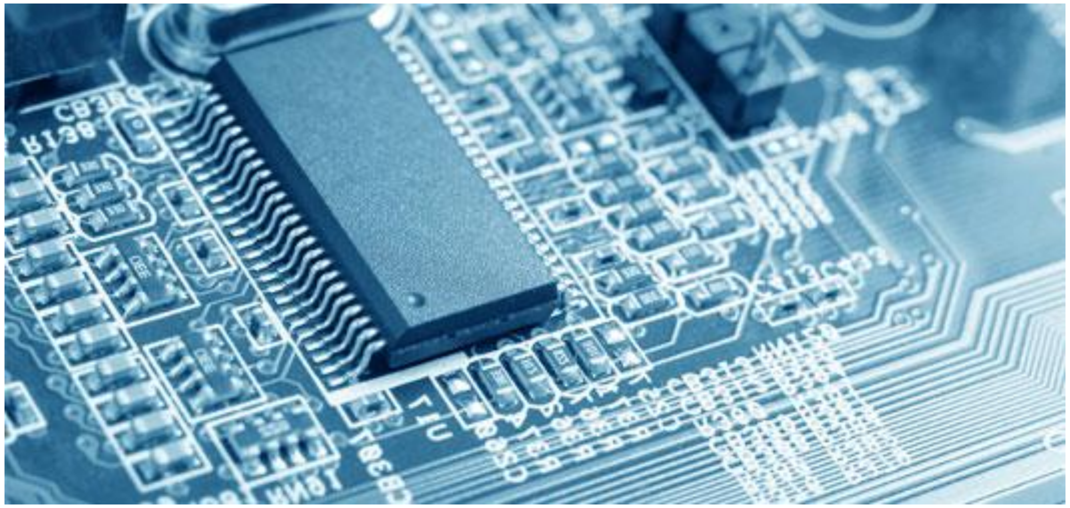
NAZARBAYEV UNIVERSITY
SCHOOL OF SCIENCE AND TECHNOLOGY



ROBT305 - Embedded Systems

Lecture 13 – Hardware for Embedded Systems

3 November, 2015

Course Logistics

Reading Assignment:

Chapter 2 of the Real-Time Systems Design and Analysis textbook

Homework Assignment #4 demonstration at class on Thursday 5 November

Project Assignment 1 is on Moodle due to 17 November (class demonstration)

Topics

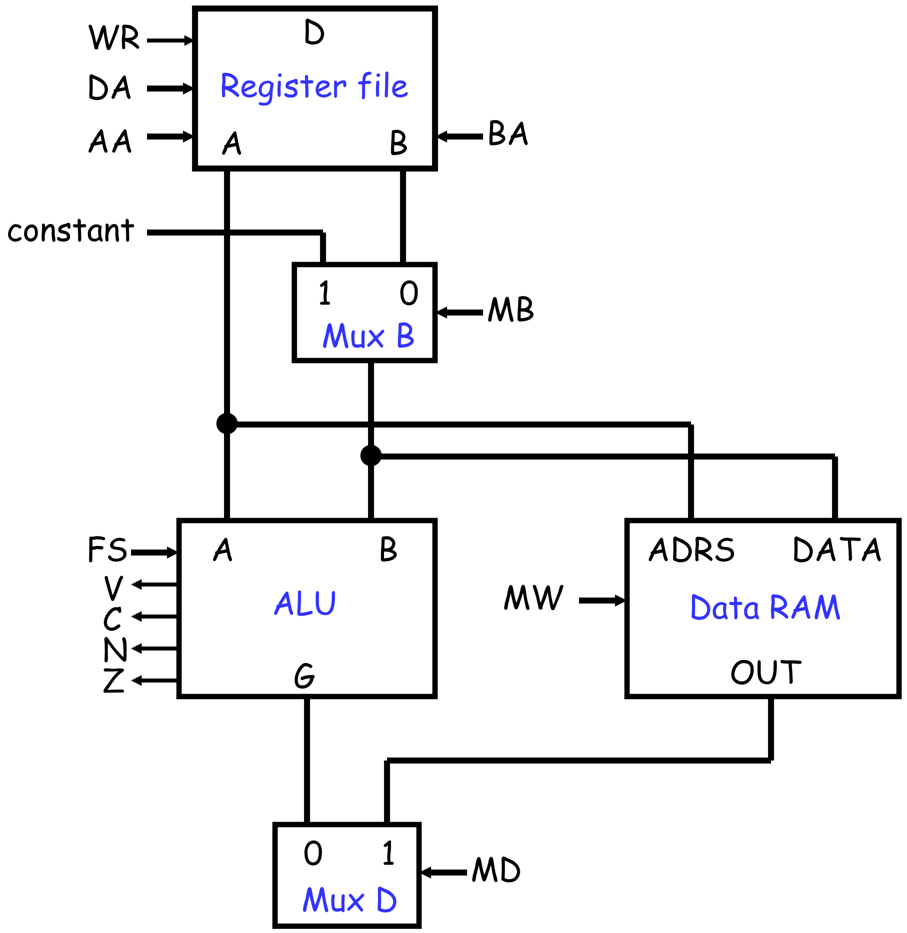
Topics to Cover Today
Datapath and Instruction Set Architecture
Interrupts
Pipelining
RISC and CISC CPU architectures
Memory technologies: Cache, ROM, RAM

Introduction

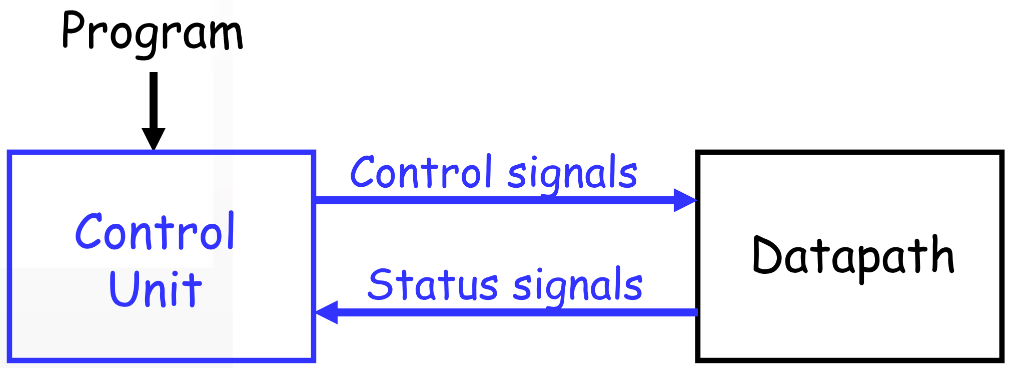
- Embedded systems interact closely with specialize input and output devices including sensors and actuators
- Hardware structure can affect timing and require special software design techniques

Datapath Review

- Set WR = 1 to write one of the registers.
- DA is the register to save to.
- AA and BA select the source registers.
- MB chooses a register or a constant operand.
- FS selects an ALU operation.
- MW = 1 to write to memory.
- MD selects between the ALU result and the RAM output.
- V, C, N and Z are status bits.



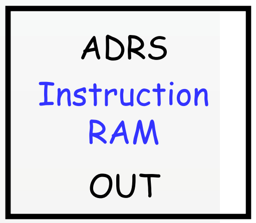
Block Diagram of a Processor

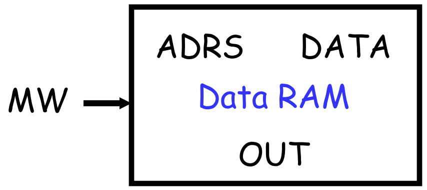


- The control unit connects programs with the datapath.
  - It converts program instructions into control words for the datapath, including signals WR, DA, AA, BA, MB, FS, MW, MD.
  - It executes program instructions in the correct sequence.
  - It generates the "constant" input for the datapath.
- The datapath also sends information back to the control unit. For instance, the ALU status bits V, C, N, Z can be inspected by branch instructions to alter a program's control flow.

Where Does the Program Go?

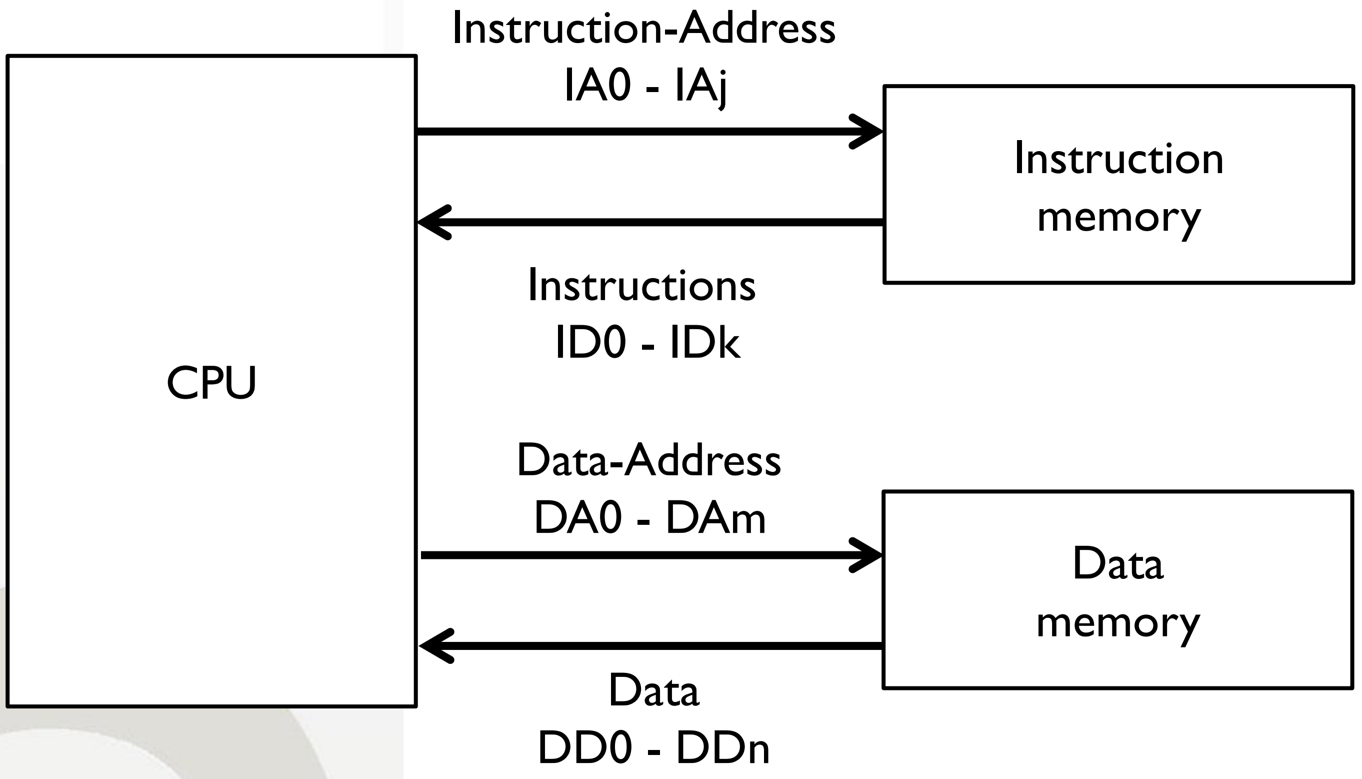
- Harvard architecture includes two memory units.
  - An instruction memory holds the program.
  - A separate data memory is used for computations.
  - **The advantage is that we can read an instruction *and* load or store data in the same clock cycle.**
- For simplicity, our diagrams do not show any WR or DATA inputs to the instruction memory.





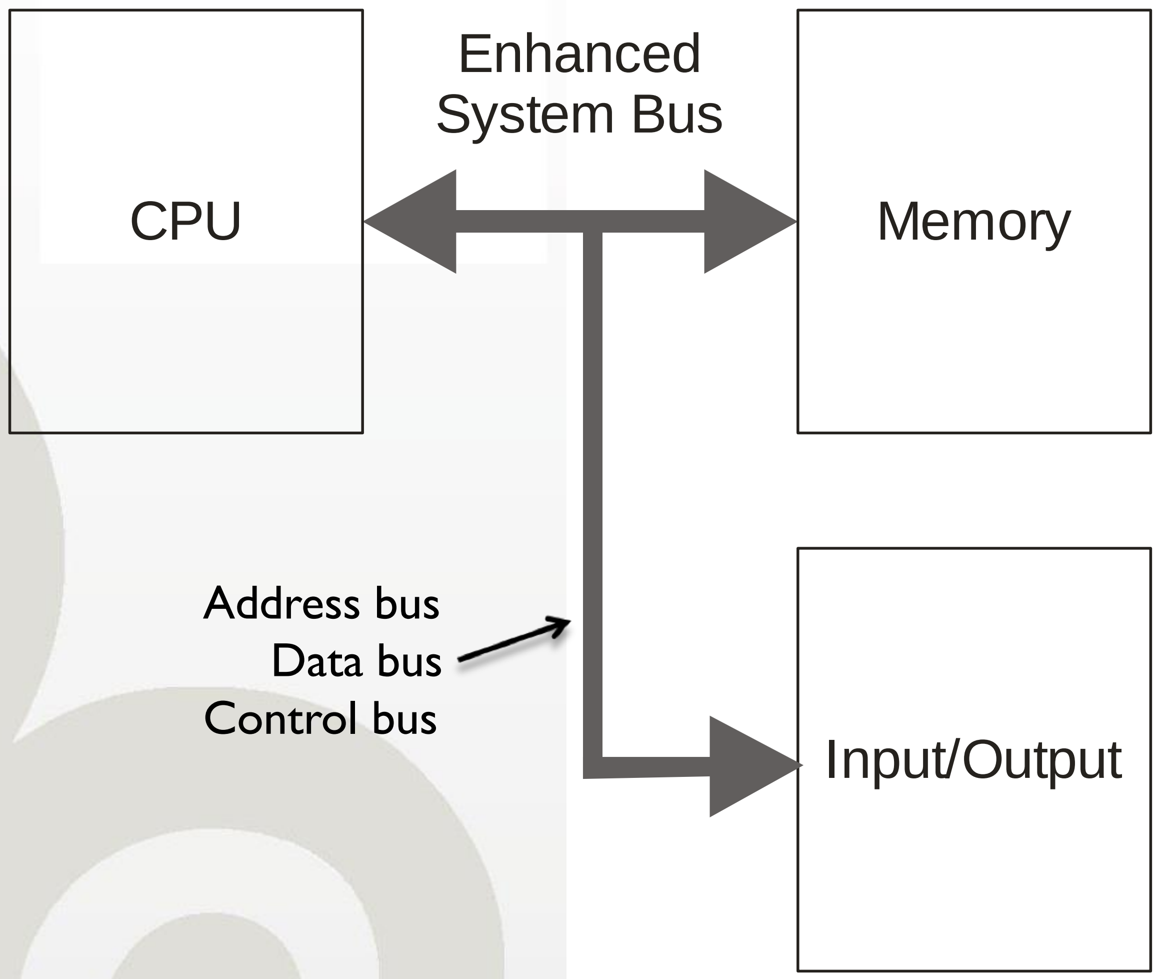
- Caches in modern CPUs often feature a Harvard architecture like this.
- However, there is usually a single main memory that holds both program instructions and data, in a Von Neumann architecture.

Harvard Architecture



Possible to have different bus widths for instruction and data transfer

Von Neumann Architecture

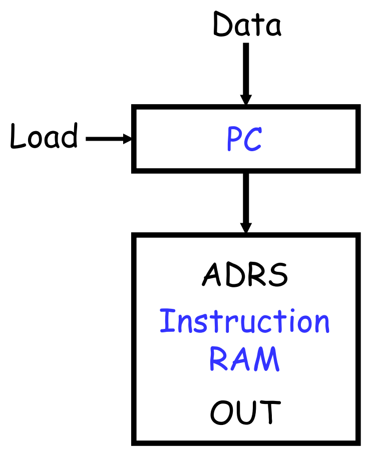


John von Neumann
1903 – 1957

Also known as Princeton Architecture

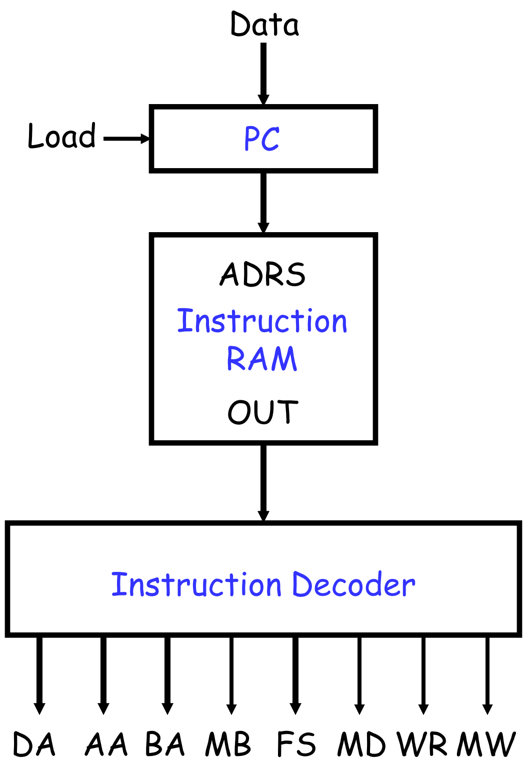
Program Counter

- A program counter or PC addresses the instruction memory, to keep track of the instruction currently being executed.
- On each clock cycle, the counter does one of two things.
  - If Load = 0, the PC increments, so the next instruction in memory will be executed.
  - If Load = 1, the PC is updated with Data, which represents some address specified in a jump or branch instruction.



Instruction Decoder

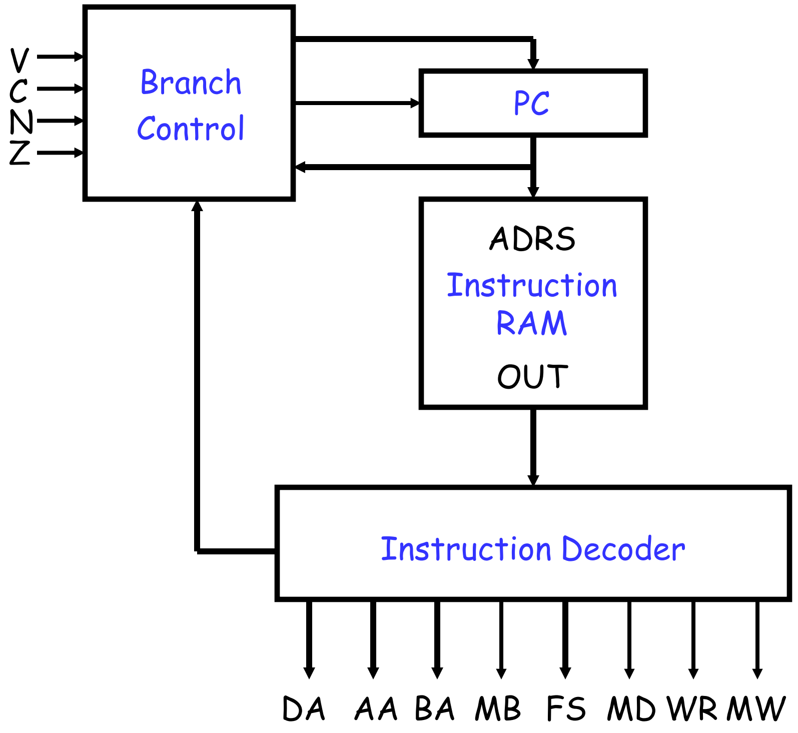
- The instruction decoder is a combinational circuit that takes a machine language instruction and produces the matching control signals for the datapath.
- These signals tell the datapath which registers or memory locations to access, and what ALU operations to perform.



(to the datapath)

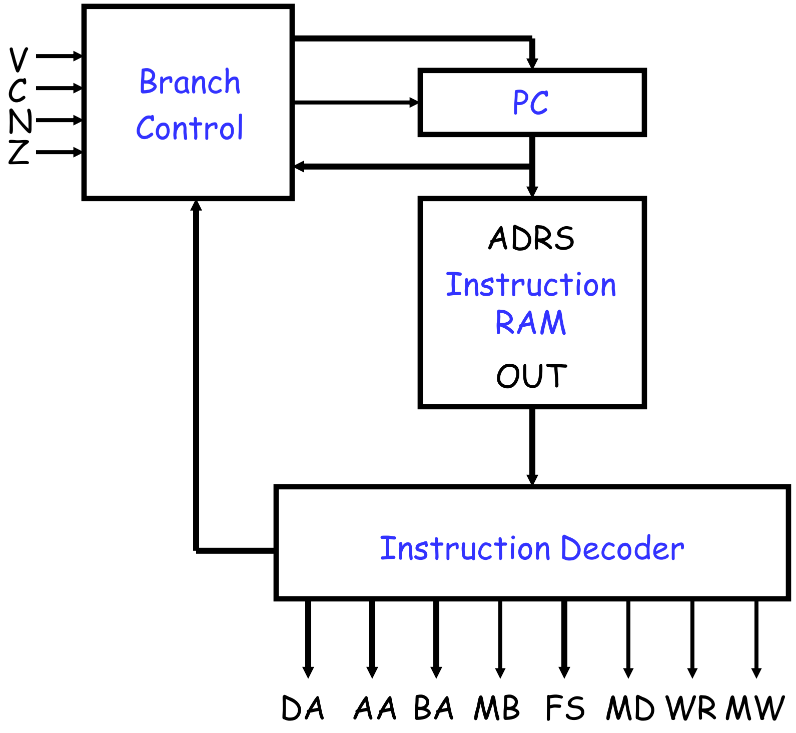
Jumps and Branches

- Finally, the branch control unit decides what the PC's next value should be.
  - For jumps, the PC should be loaded with the target address specified in the instruction.
  - For branch instructions, the PC should be loaded with the target address only if the corresponding status bit is true.
  - For all other instructions, the PC should just increment.

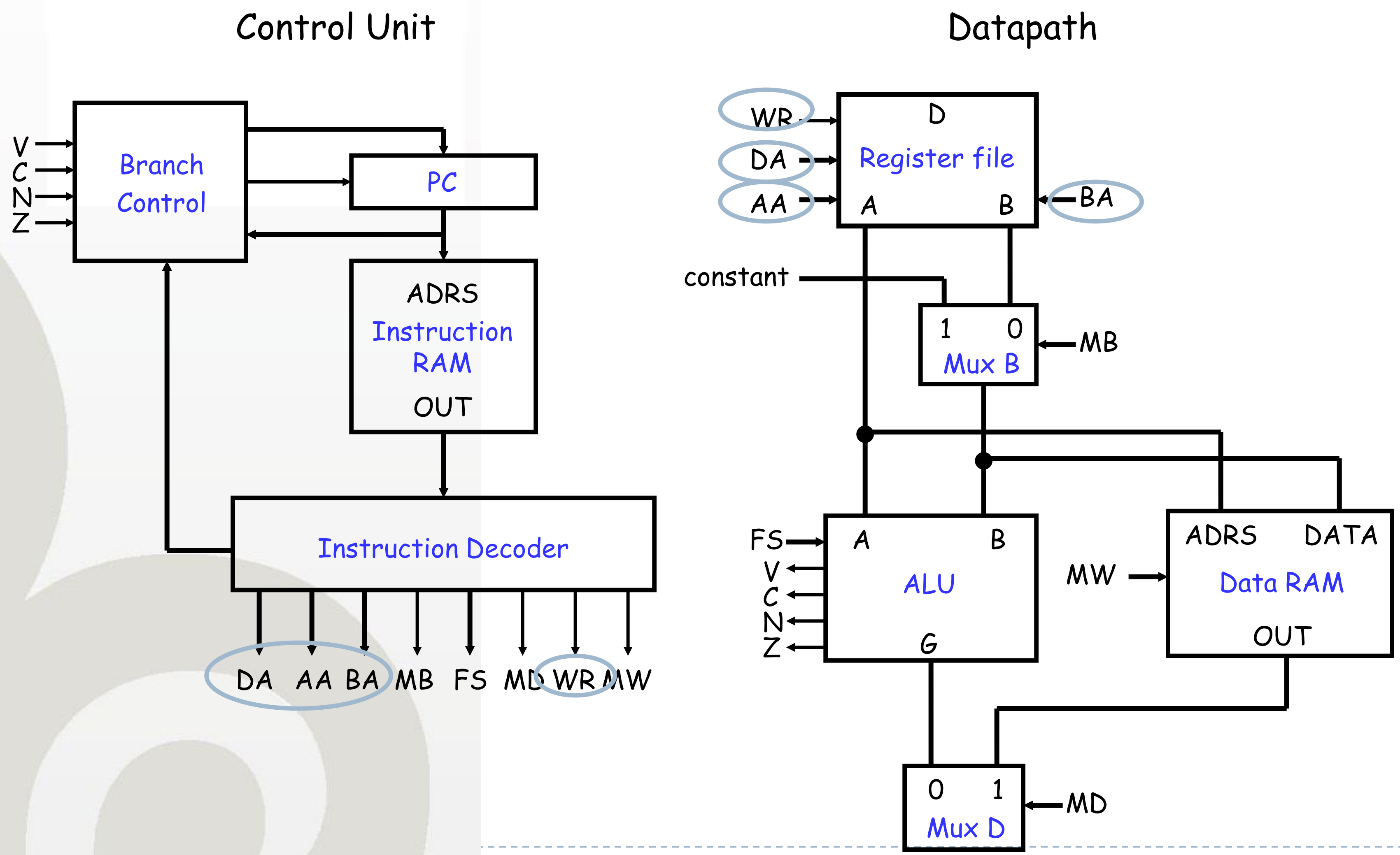


That's It!

- This is the basic control unit. On each clock cycle:
  1. An instruction is read from the instruction memory.
  2. The instruction decoder generates the matching datapath control word.
  3. Datapath registers are read and sent to the ALU or the data memory.
  4. ALU or RAM outputs are written back to the register file.
  5. The PC is incremented, or reloaded for branches and jumps.



The Whole Processor



Instruction Set Architecture (ISA) for Simple Computer

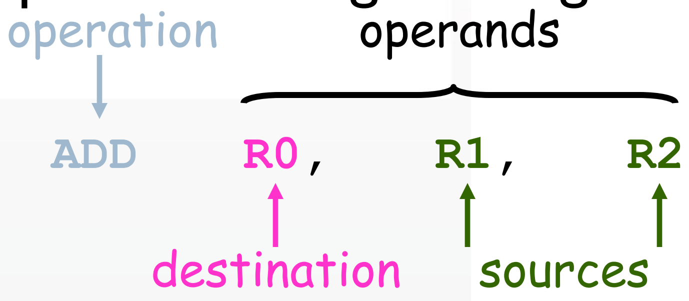
- A programmable system uses a sequence of ***instructions*** to control its operation
- A typical instruction specifies:
  - Operation to be performed
  - Operands to use, and
  - Where to place the result, or
  - Which instruction to execute next
- Instructions are stored in RAM or ROM as a *program*
- The addresses for instructions in a computer are provided by a ***program counter*** *(PC)* that can
  - Count up
  - Load a new address based on an instruction and, optionally, status information
- Executing an instruction - activating the necessary sequence of operations specified by the instruction

Fetch and Execute Cycle

- Programs are a sequence of macroinstructions or macrocode stored in the main memory in binary form.
- Macroinstructions are sequentially fetched from the main memory location pointed to by the program counter, and placed in the instruction register.
- Each instruction consists of an operation code or opcode field and zero or more operand fields.
- The control unit decodes the instruction.
- After executing the instruction, the next macroinstruction is retrieved from main memory and executed.
- Certain macroinstructions or external conditions may cause a nonconsecutive macroinstruction to be executed.
- This process is called the fetch-execute cycle (or fetch-decode-execute).
- Even when "idling," the computer is fetching and executing an instruction that causes no effective change to the state of the CPU and is called a no-op (for no-operation).

Data Manipulation Instructions

- Data manipulation instructions correspond to ALU operations.
- For example, here is a possible addition instruction, and its equivalent using our register transfer notation:



Register transfer instruction:

$$R0 \leftarrow R1 + R2$$

- This is similar to a high-level programming statement like

```
R0 = R1 + R2
```

- Here, all of the operands are registers.

A Small Example

- Here's an example register-transfer operation.

```
M[1000] ← M[1000] + 1
```

- This is the assembly-language equivalent:

```
LD  R0, #1000        // R0 ← 1000
LD  R3, (R0)         // R3 ← M[1000]
ADD R3, R3, #1       // R3 ← R3 + 1
ST  (R0), R3         // M[1000] ← R3
```

- An awful lot of assembly instructions are needed!
  - For instance, we have to load the memory address 1000 into a register first, and then use that register to access the RAM.
  - This is due to our relatively simple datapath design, which only allows register and constant operands to the ALU.

Control Flow Instructions

- Programs consist of a lot of sequential instructions, which are meant to be executed one after another.
- Thus, programs are stored in memory so that:
  - Each program instruction occupies one address.
  -

```
768:    LD  R0, #1000      // R0 ← 1000
769:    LD  R3, (R0)       // R3 ← M[1000]
770:    ADD R3, R3, #1     // R3 ← R3 + 1
771:    ST  (R0), R3       // M[1000] ← R3
```

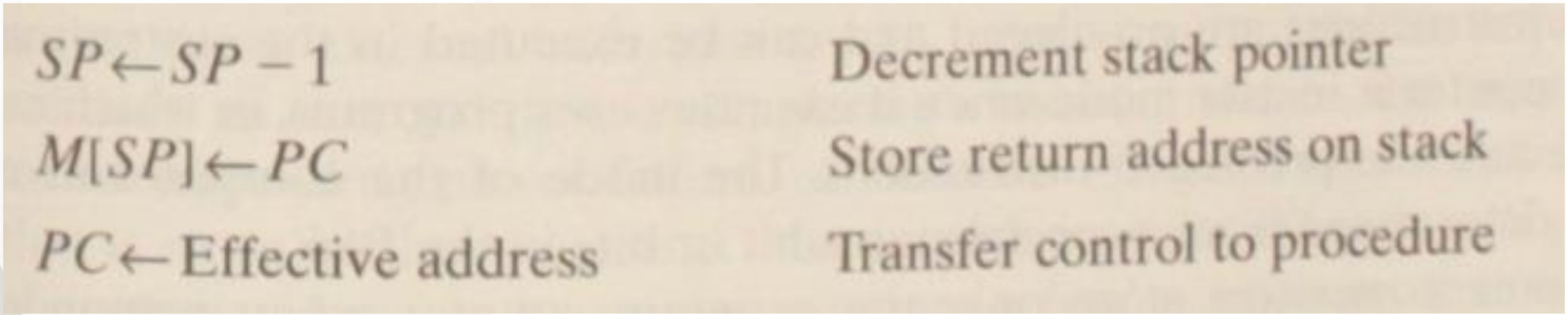
- A program counter (PC) keeps track of the current instruction address.
  - Ordinarily, the PC just increments after executing each instruction.
  - But sometimes we need to change this normal sequential behavior, with special control flow instructions.

Procedure Call and Return Instructions

- A **procedure** (is also called a **subroutine**) is a self contained sequence of instructions that performs a given computational task
- During the execution of a program, a procedure may be called to perform its function many times at various points in the program using a **call procedure** and **a memory stack**

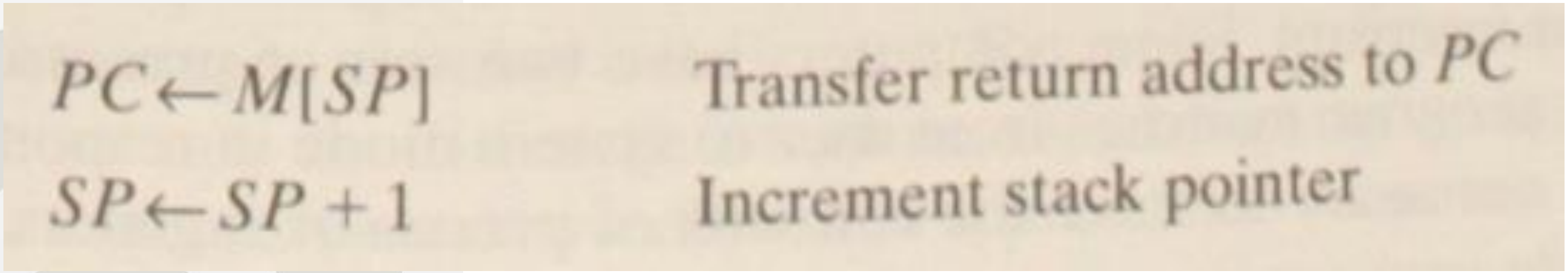
Procedure Call and Return Instructions

- A procedure call instruction using a stack



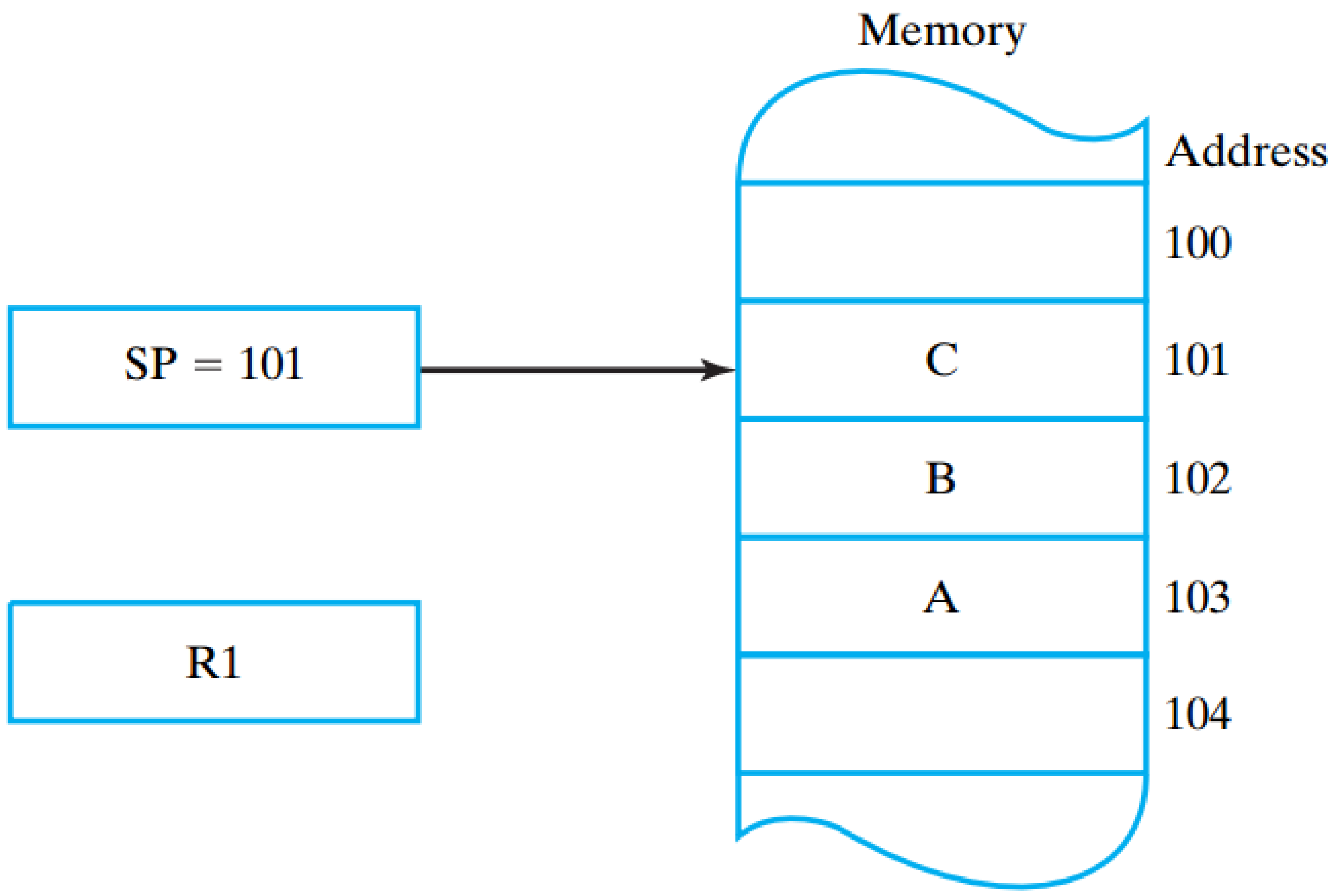
$SP \leftarrow SP - 1$	Decrement stack pointer
$M[SP] \leftarrow PC$	Store return address on stack
$PC \leftarrow$ Effective address	Transfer control to procedure

The return instruction is implementing by popping the stack and transferring the return address to the PC



$PC \leftarrow M[SP]$	Transfer return address to PC
$SP \leftarrow SP + 1$	Increment stack pointer

Stack Instructions



Interrupts

- A **program interrupt** transfers control from a program that is currently running to another service program as a result of an externally or internally generated request.
- Control returns to the original program after the service program is executed
- External interrupts – hardware mechanism for providing asynchronous response to external events (requests).
- Used to signal that I/O transfer was completed or needs to be initiated

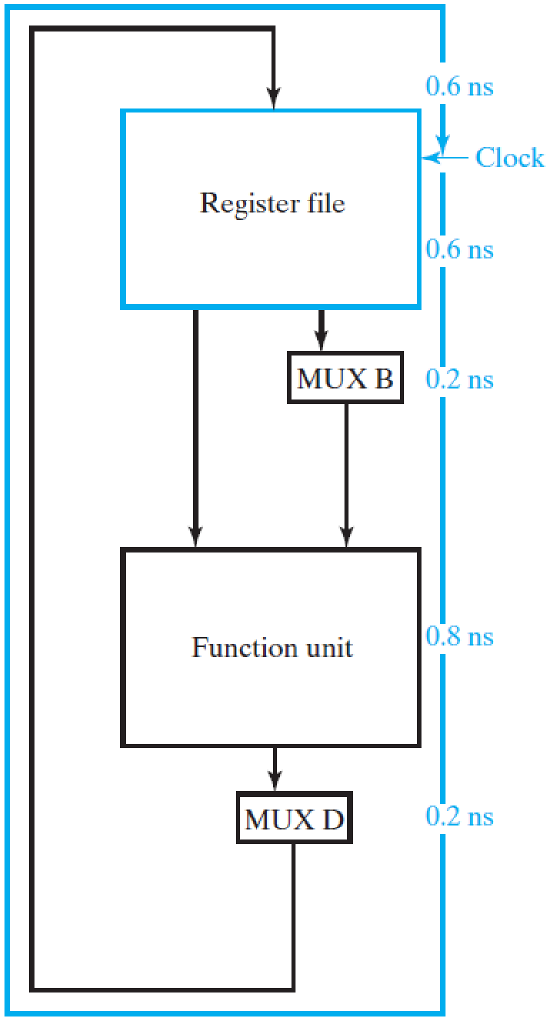
Interrupt Service Process

Typical interrupt process:

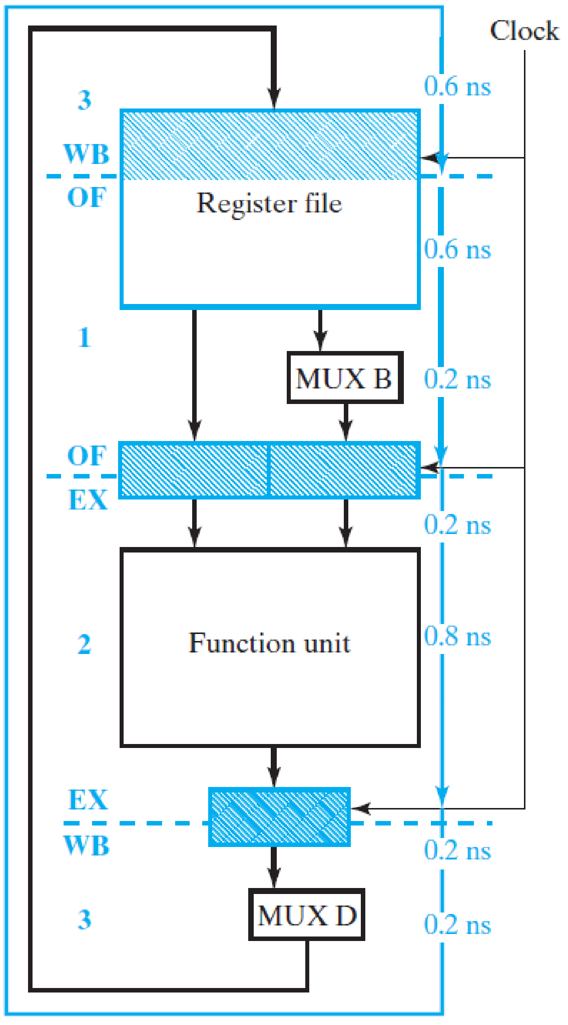
- The interrupt request line is activated
- The interrupt request is latched by the CPU hardware (~)
- The processing of the ongoing instruction is completed(~)
- The content of program counter PC is pushed to stack
- The content of status register SR is pushed to stack
- The PC is loaded with the interrupt handler's address
- The interrupt handler is executed (~)
- The original content of SR is popped from stack
- The original content of PC is popped from stack

(~) sources of variable length latency

Datapath Timing



(a) Conventional

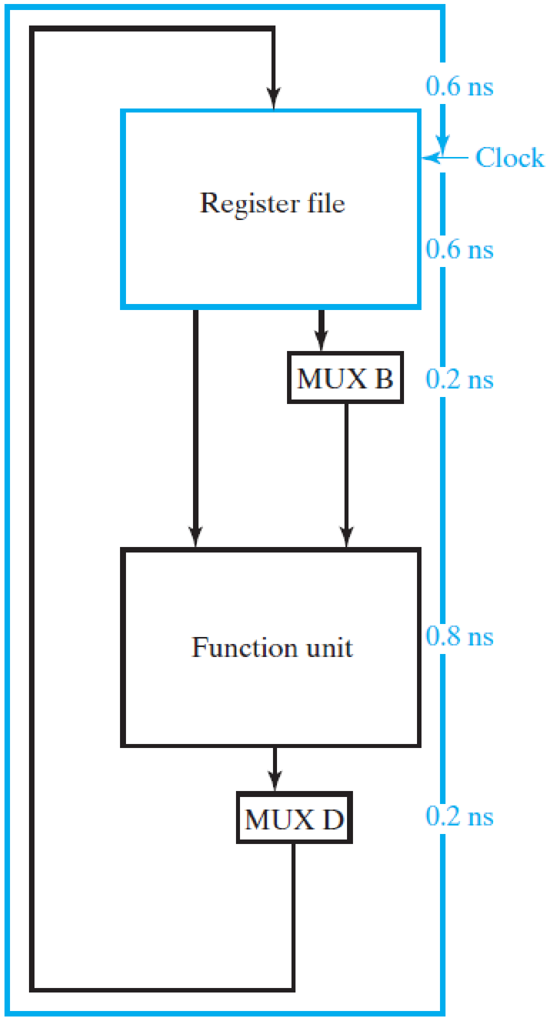


(b) Pipelined

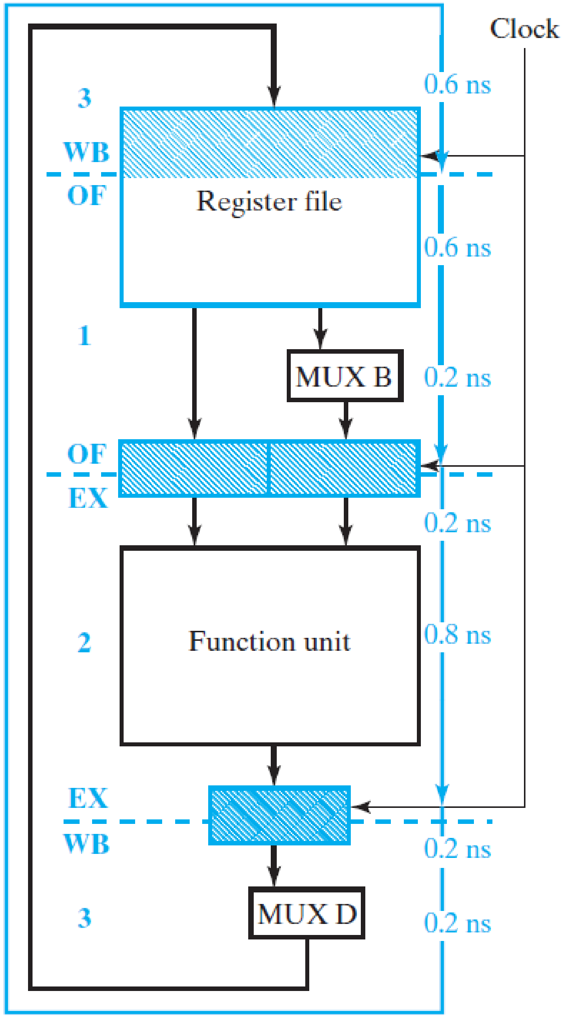
- ❑ Fig. A (left) illustrates maximum delay values for each of the components of a typical datapath
- ❑ About 2.4 ns is needed to perform a single microoperation
- ❑ The max rate of microoperation execution is the inverse of 2.4 ns = 416.7MHz - max. frequency at which the clock can be operated with clock period 2.8 ns
- ❑ To reduce clock period can be done by breaking up the 2.8 ns delay with registers – **pipelined datapath**

c.
Kime

Datapath Timing



(a) Conventional

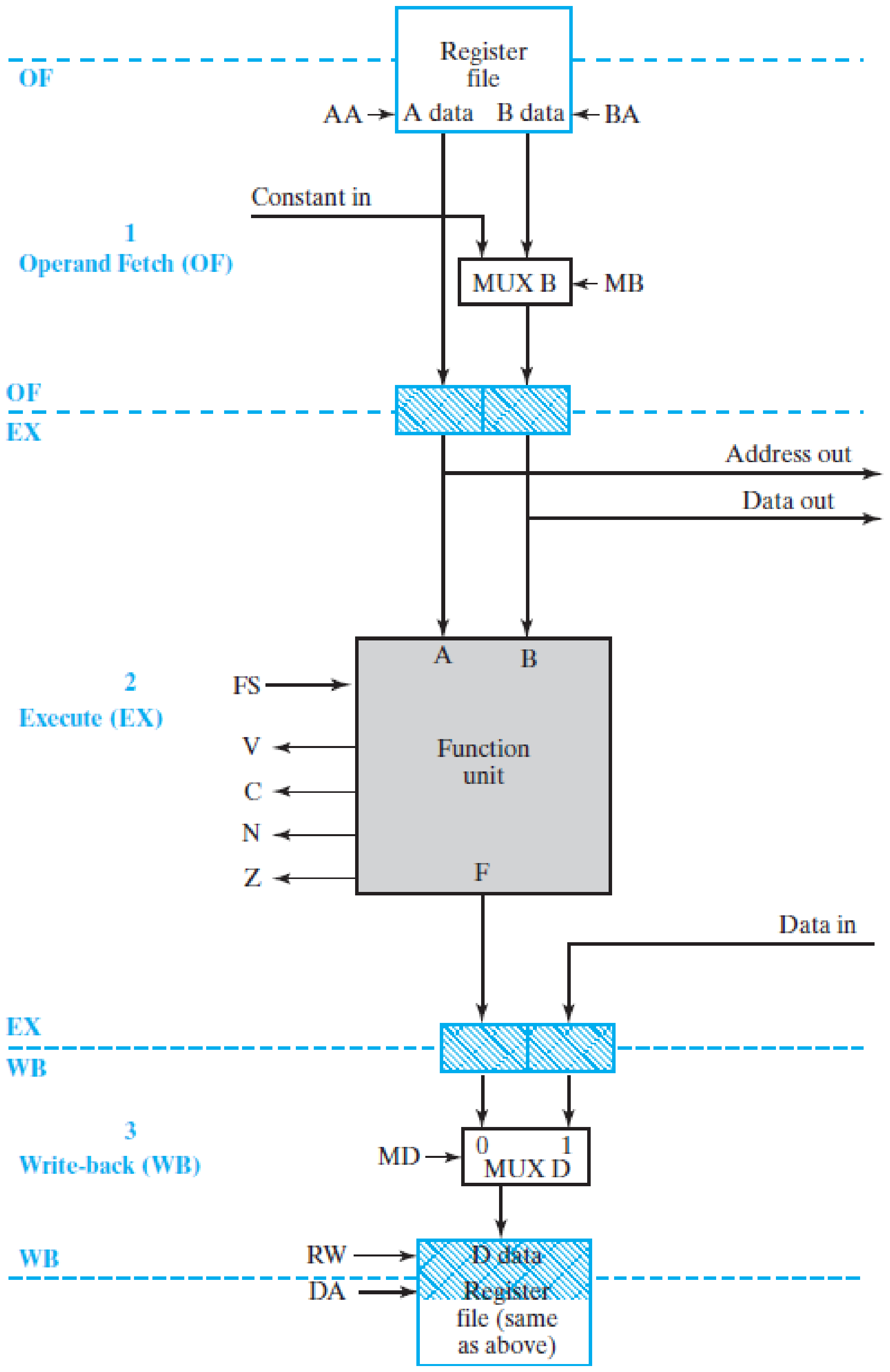


(b) Pipelined

- ❑ Fig. B shows desired structure with division to thee stages:
  1. Operation fetch (OF)
  2. Execution (EX)
  3. Write-back (WB)
- ❑ OF has delay for reading the register file and selecting MUX B = 0.8 ns
- ❑ EX has 1.0 ns delay mainly due Fuction unit
- ❑ WB has 1.0 ns in total for selecting MUX D and writing back to the register file
- ❑ Thus, all delays are at most 1.0 ns allowing clock period of 1.0 ns and max. clock frequency 1 GHz

Pipelined Datapath

- ❑ More detailed diagram of the pipelined datapath
- ❑ The register file is shown twice
- ❑ Pipelining imparts an implicit execution parallelism in the different cycles of processing an instruction.
- ❑ With pipelining, more instructions can be processed in different cycles simultaneously, improving processor performance.

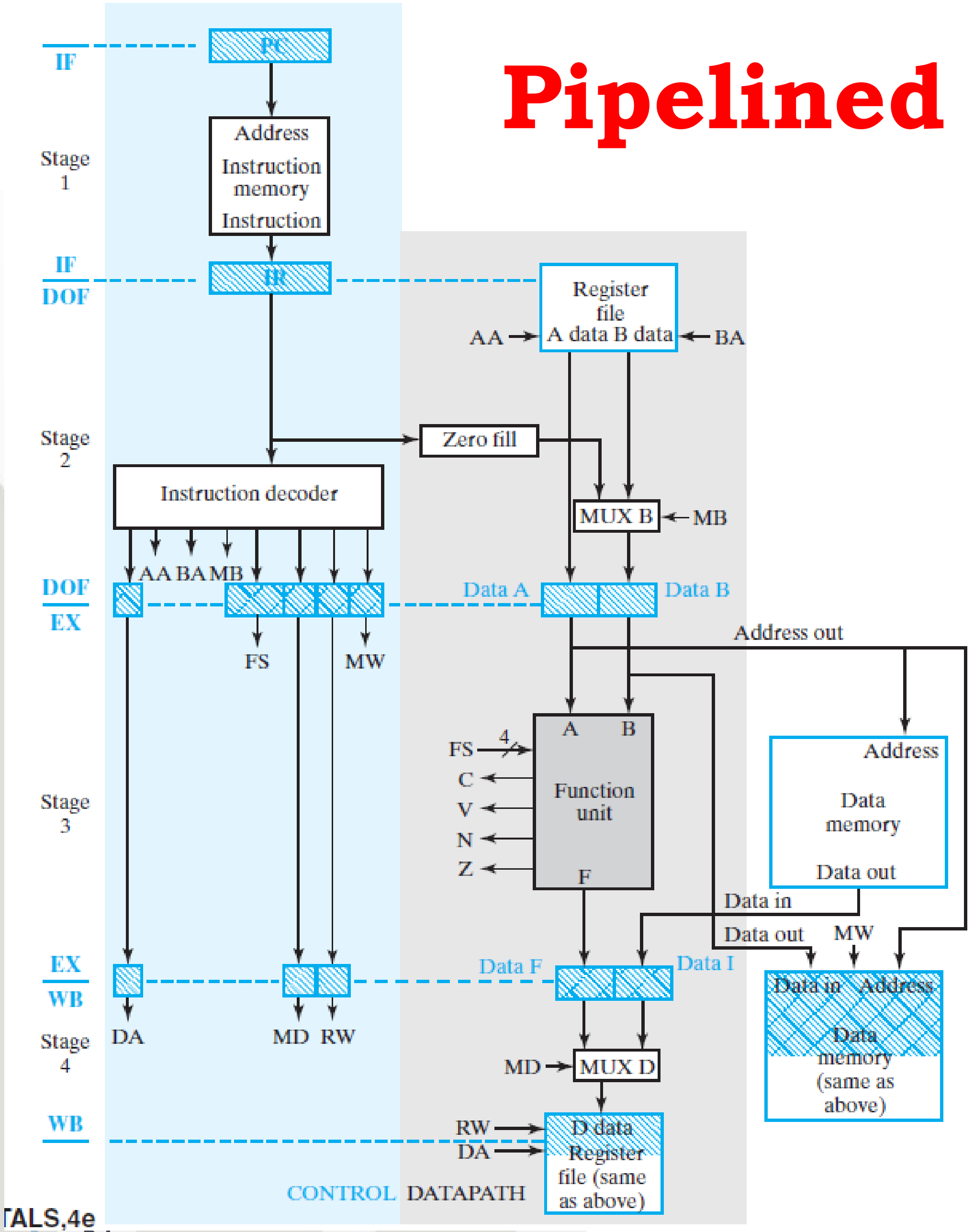


Pipeline Execution

Clock cycle

Microoperation		1	2	3	4	5	6	7	8	9
R1←R2 − R3	1	OF	EX	WB						
R4 ← sl R6	2		OF	EX	WB					
R7←R7 + 1	3			OF	EX	WB				
R1←R0 + 2	4				OF	EX	WB			
Data out ←R3	5					OF	EX	WB		
R4 ← Data in	6						OF	EX	WB	
R5 ← 0	7							OF	EX	WB

- 7 microoperations requires 9 clock cycles to execute completely.
- Time required = 9 x 1ns = 9 ns compared to 7 x 2.4 ns = 16.8 ns for the traditional datapath (1.9 times faster using pipeline)



Pipelined Control

- The control has added
  Stage 1 - Instruction Fetch (IF)
- Stage 2 – Decode and Operand Fetch (DOF)
- Location of the pipeline platfroms balance partitioning of the delays with max. no more than1 ns.
- Max clock frequency is 1 GHz
- An instruction takes
  4 x 1 ns = 4 ns to execute

Pipelining

- A new instruction is completed at the rate of one clock cycle
- Pipelining can actually degrade performance – if any of the instructions in the pipeline are a branch instruction, the prefetched instructions further in the pipeline are no longer valid and must be flushed.
- Superpipelined architectures can be achieved if the fetch execute cycle can be decomposed further.
- A superscalar architecture can use redundant hardware to replicate one or more stages in the pipeline.

RISC Architecture

- Reduced instruction set computers (RISC):

1. Simple instruction taking one clock cycle
2. Memory access by load/store instructions only
3. Highly pipelining instruction processing
4. Hard-wired control unit
5. Small number of instructions
6. Instructions are fixed format and length
7. Few addressing modes
8. Multiple sets of work registers
9. Complexity handled by compilers and software

CISC Architercture

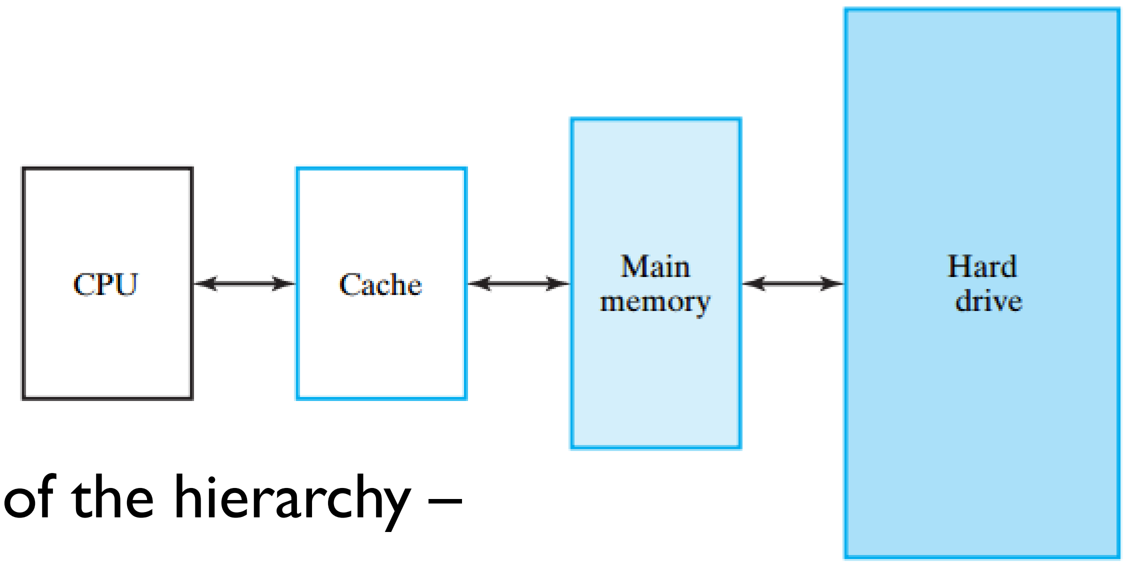
- Complex instruction set computers (CISC):

1. Complex instructions take multiple clock cycles
2. Practically any instruction can reference memory
3. No instruction pipelining
4. Microprogrammed control unit
5. Large number of instructions
6. Instructions are of variable format and length
7. Great variety of addressing modes
8. Single set of work registers
9. Complexity handled by the microprogram and hardware

CISC vs RISC

- RISC has fewer instructions, complex operations are sequence of simple instructions
- RISC's major advantage in real-time systems:
  - is the shorter average instruction execution time than for CISCs
  - shorter interrupt latency, hence shorter response times
- RISC's downsides:
  - Processors associated with cashes and multistage pipelines
  - Response time increases with low cache hit ratios
- Often tolerable in firm and soft RTOS

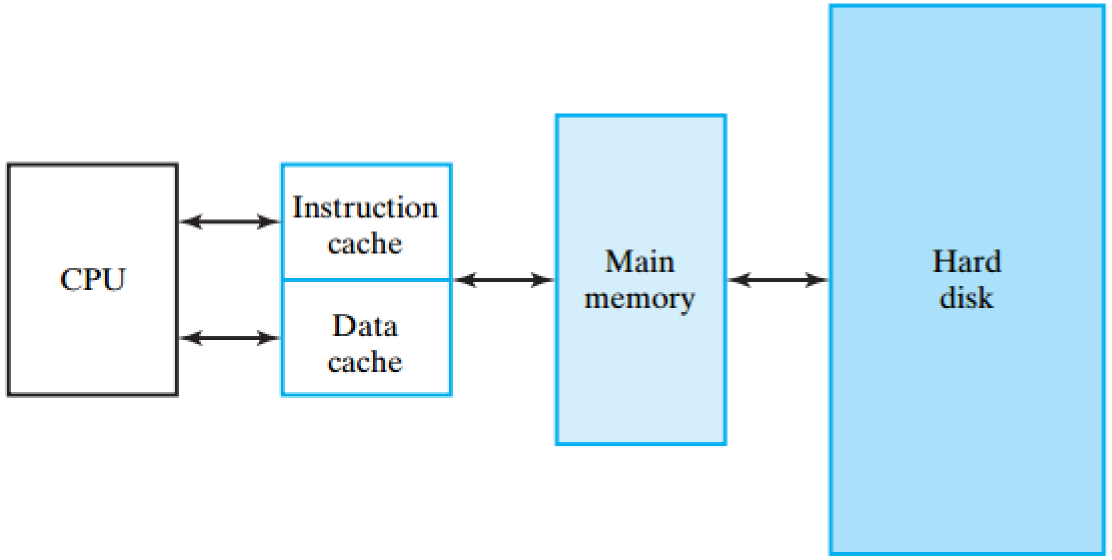
Hierarchical Memory Organization



- A cache – lowest level of the hierarchy – a small, fast memory.
- Main memory RISC serves directly most of CPU instructions, operand fetches not satisfied by the cache
- Top level – hard drive – which is accessed only in the very infrequent cases in which CPU instruction or operand fetch is not found in main memory.

Cache

- A small block of fast memory where frequently used instructions and data are kept. The cache is much smaller than the main memory.
- Suppose that cache memory is divided to two cashes: one for instructions and one for data.
- Thus, one instruction can be fetched and one operand can be fetched/stored, in a single clock cycle if the cashes are fast enough

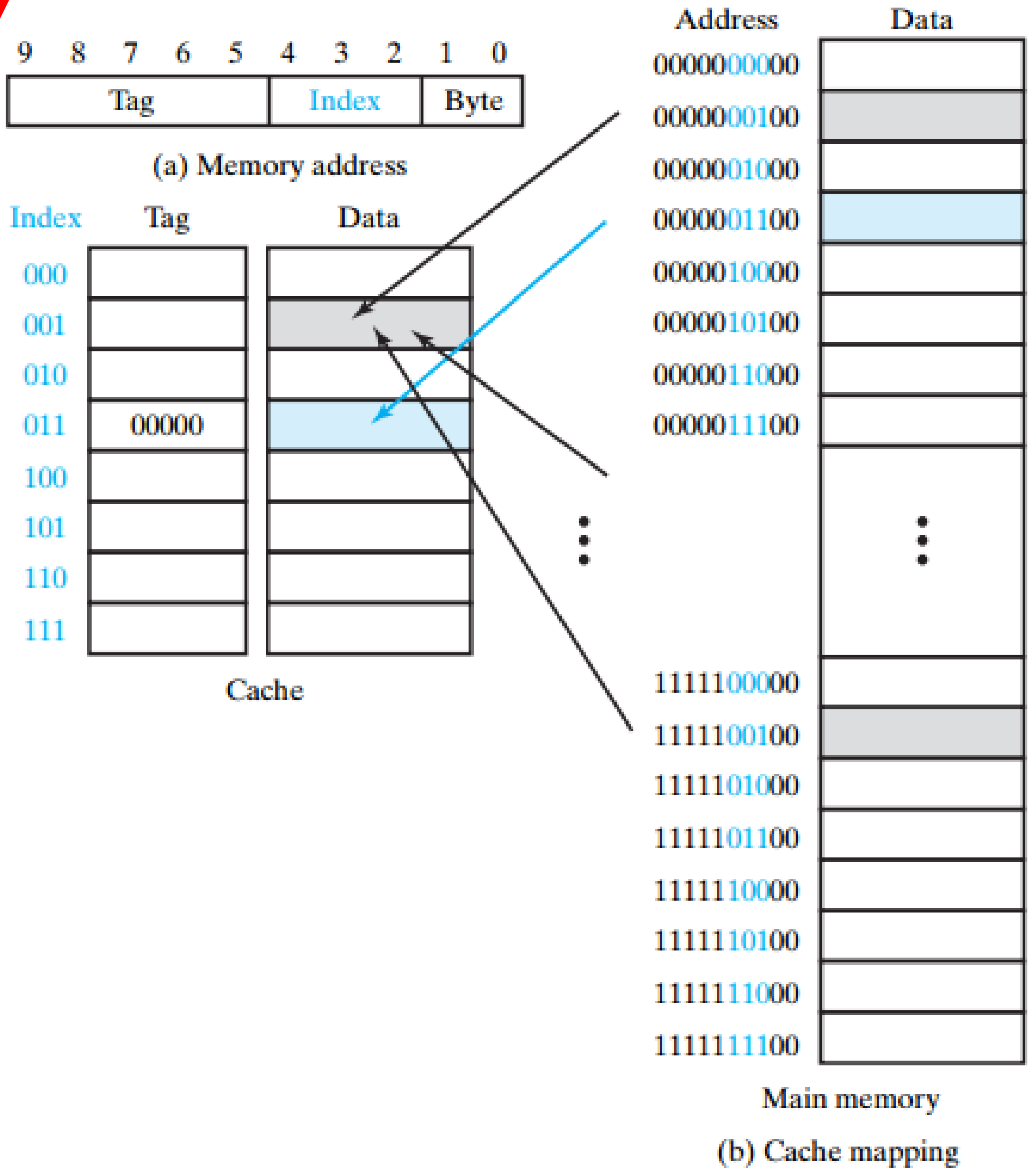


Cache

- Assume that typical access time to DRAM is about 10 ns
- If CPU accesses only RAM memory than CPU with 1 ns clock cycle would operate too slow. At 1/10 of its full speed
- If 95% of all memory accesses are made with cashes taking 2 ns and
- Remaining 5% of memory accesses take 10 ns.
- Then the average time is **0.95 x 2 + 0.05 x10 = 2.4 ns**

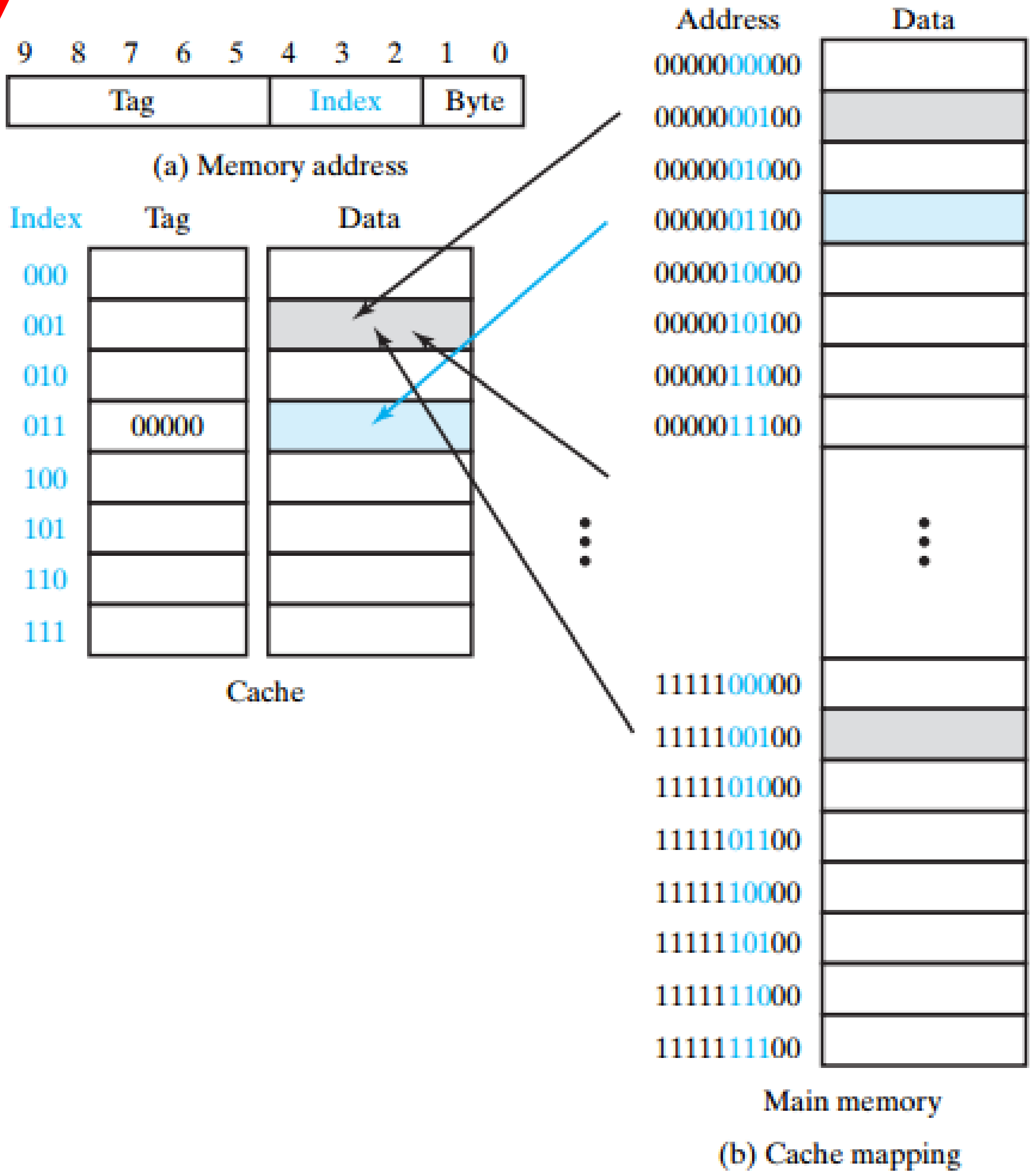
Cache Memory

- Assume a very small cache of 8 x 32 bit words and a small main memory with 1 KB (256 words)
- Bits 2 to 4 of the main memory address are the cache address (index)
- Upper 5 bits of the main memory address (tag) are stored in the cache with the data
- Main memory address 0000001100 has cache address 011 and tag 00000
- The tag combined with index and 00 identify the address in the main memory



Cache Memory

- Let CPU fetch instruction from location 0000001100 from main memory.
- The cache separates the tag 00000 from the cache address 011, internally fetches the tag and stores word from location 011 in the cache memory
- Then, the cache compares the tag fetched with the tag portion of the address from the CPU.
- If the tag fetched is 00000, then the tags match. And the stored word fetched from cache memory is the desired instruction.
- If the tag fetched from cache memory is not 00000, then there is a tag mismatch and the instruction is retrieved from the main memory.
- When such cache miss occurs, the word from main memory not only go to CPU but is stored to the cache in anticipation of future accesses to the same memory address.



Cache

- Usage:
  - Upon memory access check the address tags to see if data is in the cache.
  - If present, retrieve data from cache.
  - If data or instruction is not already in the cache, cache contents are written back and new block read from main memory to cache.
  - The needed information is then delivered from cache to CPU and the address tags adjusted.

Cache

- Performance benefits are a function of cache hit radio.
- If needed data or instructions are not found in the cache, then the cache contents need to be written back (if any were altered) and overwritten by a memory block containing the needed information.
- Overhead can become significant when the hit ratio is low.
  - Therefore a low hit ratio can degrade performance.
- If the locality of reference is low, a low number of cache hits would be expected, degrading performance.
- Using a cache is also non-deterministic – it is impossible to know *a priori* what the cache contents and hence the overall access time will be.
- In multitasking real-time systems – high probability of cache misses due to frequent switching between different tasks and interrupts.

Memory Technologies: ROM

Two classes of memory :
RAM – random access memory
ROM – read-only memory

- Electrically erasable programmable ROM (EEPROM) and Flash (ROM) can be rewritten similar to RAM devices
- Erasing and writing process is much slower comparing to RAM and limited to 100000 -1000000 times
- EEPROM is mainly used for nonvolatile program and parameter storing
- Flash – application program and data records

Wait States

- When a microprocessor must interface with a slower peripheral or memory device, a wait state may be needed to be added to the bus cycles.
- Wait states extends the microprocessor read or write cycle by a certain number of processor clock cycles to allow the device or memory to "catch up."
- For example, EEPROM, RAM, and ROM may have different memory access times. Since RAM memory is typically faster than ROM, wait states would need to be inserted when accessing RAM.
- Wait states degrade overall systems performance, but preserve determinism.

Any Questions?