Section IV:
Digital System Organization

CEG 360/560; EE 451/651

Digital System Design

Dr. Travis Doom, Assistant Professor

Department of Computer Science and Engineering

Wright State University

Thanks to M. Mano, J. Wakerly, R. Haggard

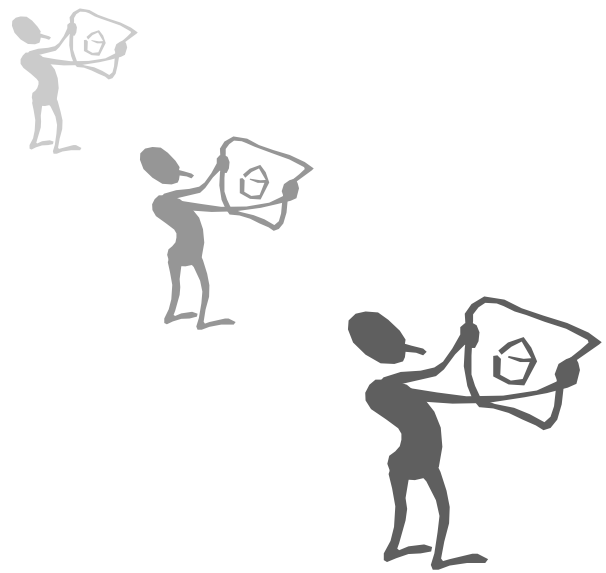
Acknowledgements

- These slides were developed with the aid of examples found in:
  - "Logic and Computer Design Fundamentals" - M. Morris Mano
  - "Digital Design: Principles and Practices" - John Wakerly

- The original version of many of these slides were kindly provided by:
  - Dr. Roger L. Haggard et al.
  - Prentice Hall, Inc.

Outline

- **Complex System Design**
  - **Register-Transfer Level Design**
  - **Data paths**
    - **The Control Word**
    - **Pipelining**
  - **Control unit**
    - **Hardwired Control**
    - **Microprogrammed Control**
    - **Programmable Control Units**
- Simple Computer Architecture
  - Computer Instructions
  - Instruction Set Architecture
- Issues in Computer Design
  - CISC vs. RISC
  - Memory Hierarchy



Introduction

- How are complex systems designed?
- What is a microoperation?
- How are register-transfer-level operations controlled?
- How is control organized implemented in a complex digital system?
- How does pipelining work?
- What is the relationship between a C-program and the instructions that executed by the microprocessor?
- What are the defining characteristics of modern RISC architectures?
- Why is memory organized hierarchically?

Complex System Design (1)

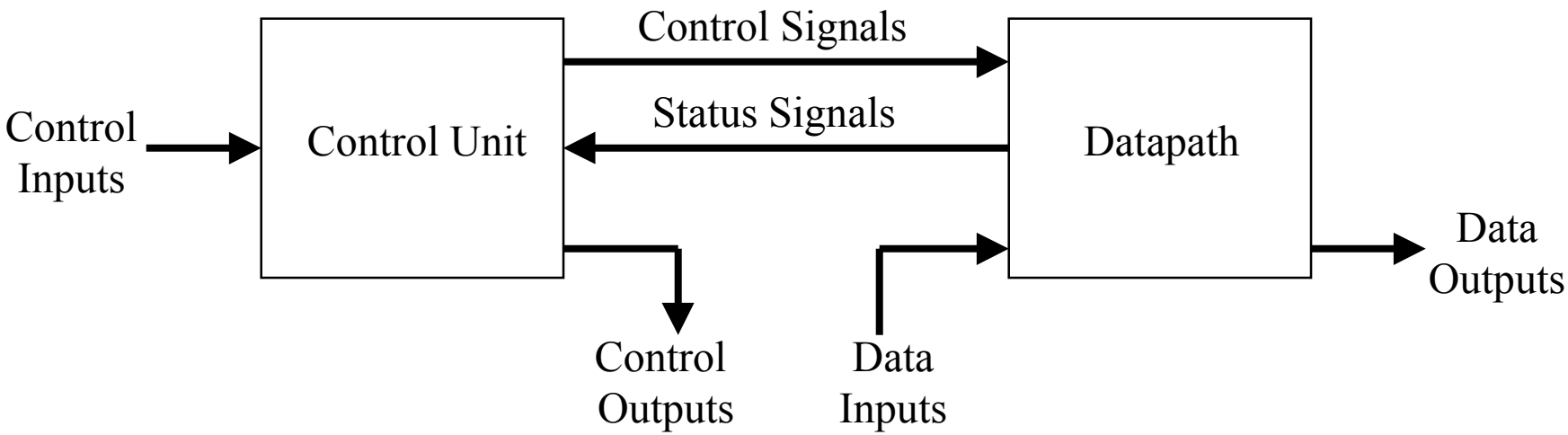
- A digital system is a sequential circuit with specified behavior.
  - A microprocessor is a digital system.

- Specifying large digital systems with state tables may be exceptionally difficult, due to the number of states involved.
  - As in computer programming, most digital systems are designed using a modular, hierarchical approach.
  - The system is partitioned into modular subsystems.
    - Each subsystem performs a well defined function with specified interface.
  - Interconnection the various subsystems though data and control signals results in a digital system.

Complex System Design (2)

- Most digital systems are partitioned into two top-level modules:
  - Data Unit (or Datapath): performs data-processing operations.
  - Control Unit: determines the sequence of these operations.
- Datapaths are sequential systems.
  - the system state is defined by the contents of the registers.
  - the functionality is the set of defined operations that can be performed on the contents of the registers.
  - Elementary operations are usually, but not always, performed in parallel on a string of bits in one clock tick.
- A microoperation is an elementary operation performed on data stored in the datapath. They fall into four general categories:
  - Transfer microoperations: transfer binary data from one register (or data input/memory) to another.
  - Arithmetic microoperations: perform arithmetic on data in registers.
  - Logic microoperations: perform bit manipulations on data in registers.
  - Shift microoperations: shift data in registers.

Interaction between Data and Control Units



- Control Signals - signals that activate data-processing functions.
  - To activate a sequence of such operations, the control unit sends the proper sequence of control signals to the datapath.
- Status Signals - signals that describe aspects of the state of the datapath.
  - The control unit uses these signals in determining the specific sequence of operations to be performed.
- Other Signals - allow the control unit and datapath to interact with other parts of the system, such as memory and input-output logic.

Register-Transfer Level Design

- An approach to specify, analyze, and design systems too complex to use the state-table based approaches commonly utilized in “simple” designs.

- The Register-Transfer Level (RTL) approach is characterized by:
  - A digital system is viewed as divided into a data subsystem and a control subsystem.
  - The state of the data subsystem consists of the contents of the registers.
  - The function of the system is performed as a sequence of register transfers.
  - A *register transfer* is a transformation performed on the datum while the datum is transferred from one register to another.
  - The sequence of register transfers is controlled by the control subsystem.
- The operation of the device can be designed as a sequence of register transfers can be designed using state diagrams, ASM charts, etc.
  - Each transfer must correspond to a sequence of microoperations.
  - The control unit implements the RTL design through microoperations.

RTL Languages (1)

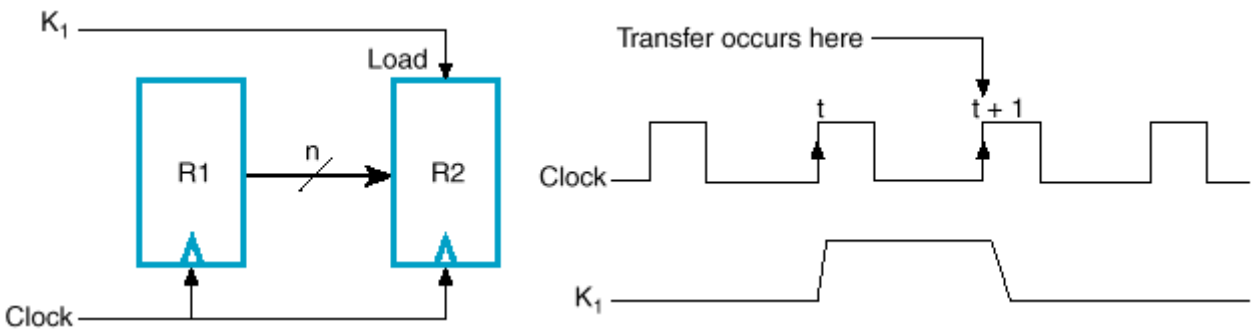
- The notation for register transfers are sufficiently complete to describe any digital system at the register-transfer level.
  - known as register-transfer languages.

- Registers are denoted by uppercase letters (sometimes followed by numbers) that indicate the function of the register
  - e.g. R0, R1, AR, PC, MAR, et al.
  - The individual bits can be denoted using parenthesis and bit numbers or labels
    - e.g. R0(0), R0(7:0), PC(L), PC(H)

- Data transfer is denoted in symbolic form by the means of the replacement operator ←.
  - e.g. R2 ← R1

RTL Languages (2)

- Normally we want a given transfer to occur not for every clock pulse, but only for specific values of the control signals.
  - RTL conditional statements:
    - e.g. If (K1 = 1) Then (R2 ← R1)
  - Control function notation (Colon, :)
    - e.g. K1: R2 ← R1



- All RTL statements occur in response to a clock tick. A comma is used to separate two or more register transfers that are executed at the same time.
  - e.g. Go: R2 ← R1, R4 ← R3

RTL Languages (3)

- Register to Memory Transfers are denoted using square brackets surrounding the memory address.
  - e.g. DR ← M[AR] (Read operation)
  - e.g. M[AR] ← SR (Write operation)

Symbol	Description	Examples
Letters (and numerals)	Denotes a register	*AR, R2, DR, IR*
Parentheses	Denotes a part of a register	*R2*(1), *R2*(7:0), *AR*(*L*)
Arrow	Denotes transfer of data	*R1* ← *R2*
Comma	Separates simultaneous transfers	*R1* ← *R2*, *R2* ← *R1*
Square brackets	Specifies an address for memory	*DR* ← *M*[*AR*]

RTL Languages (4)

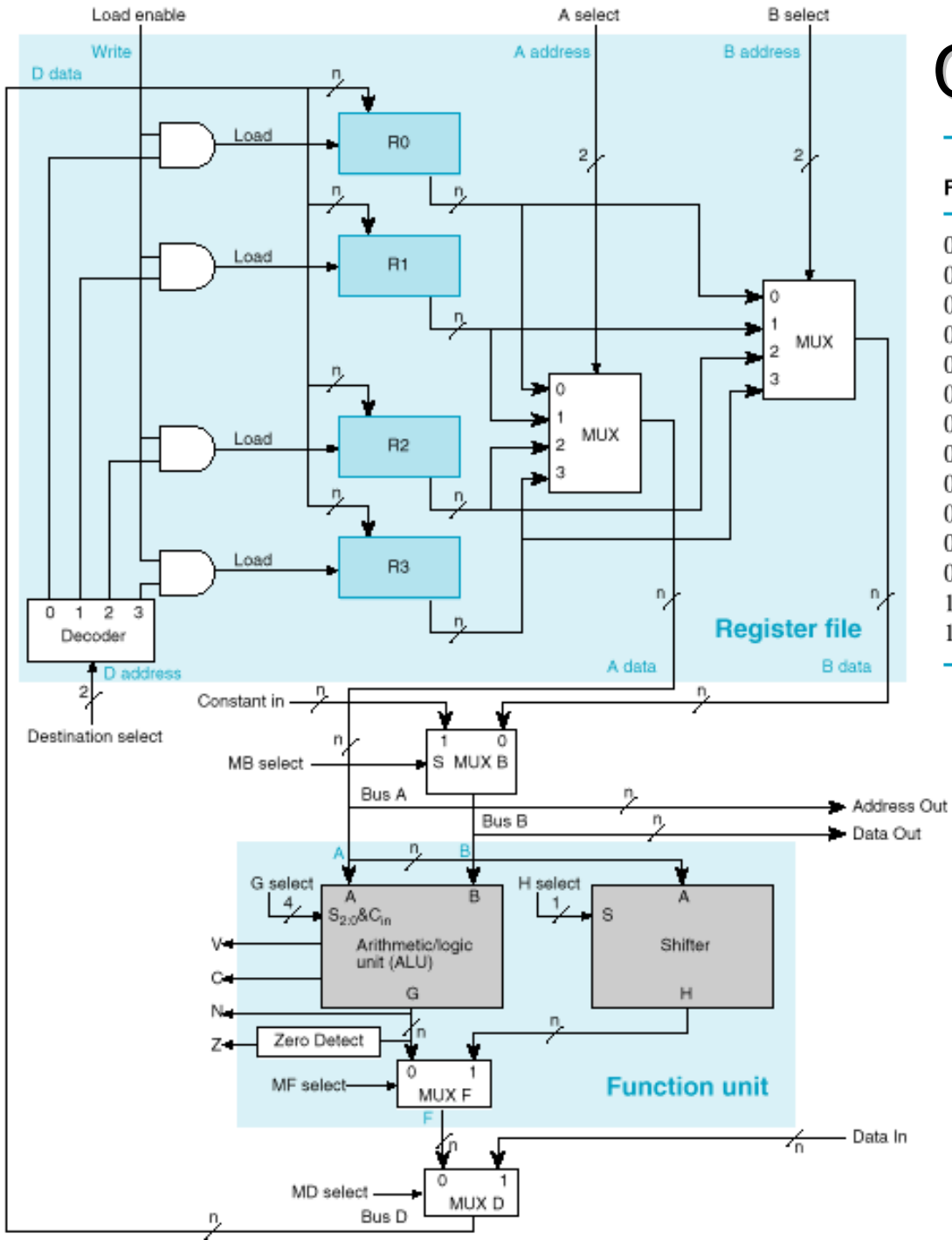
Symbolic designation	Description
$R0 \leftarrow R1 + R2$	Contents of $R1$ plus $R2$ transferred to $R0$
$R2 \leftarrow \overline{R2}$	Complement of the contents of $R2$ (1's complement)
$R2 \leftarrow \overline{R2} + 1$	2's complement of the contents of $R2$
$R0 \leftarrow R1 + \overline{R2} + 1$	$R1$ plus 2's complement of $R2$ transferred to $R0$ (subtraction)
$R1 \leftarrow R1 + 1$	Increment the contents of $R1$ (count up)
$R1 \leftarrow R1 - 1$	Decrement the contents of $R1$ (count down)

Examples of Arithmetic Microoperations

Symbolic designation	Description
$R0 \leftarrow \overline{R1}$	Logical bitwise NOT (1's complement)
$R0 \leftarrow R1 \wedge R2$	Logical bitwise AND (clears bits)
$R0 \leftarrow R1 \vee R2$	Logical bitwise OR (sets bits)
$R0 \leftarrow R1 \oplus R2$	Logical bitwise XOR (complements bits)

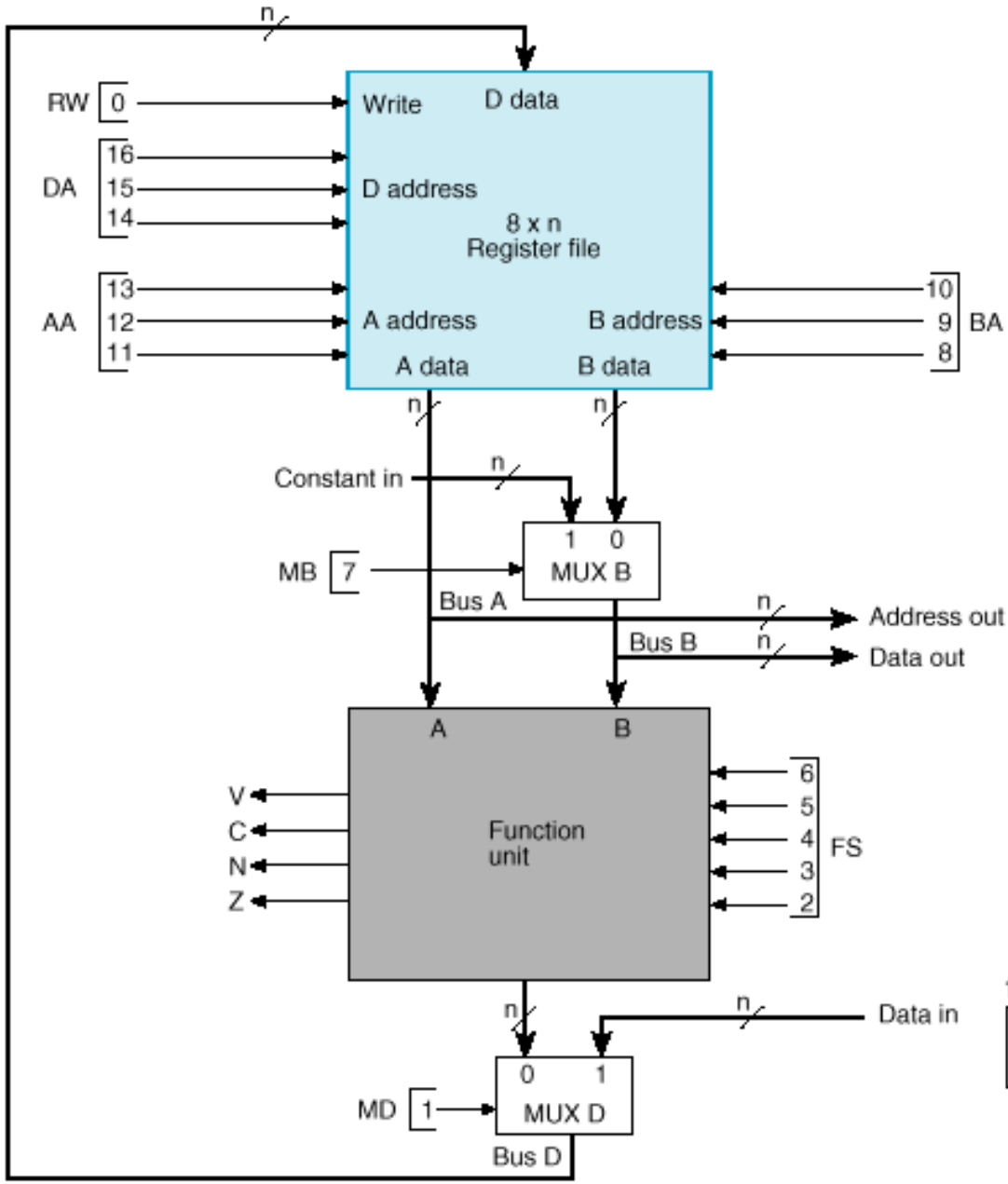
Examples of Logic Microoperations

Computer Datapath



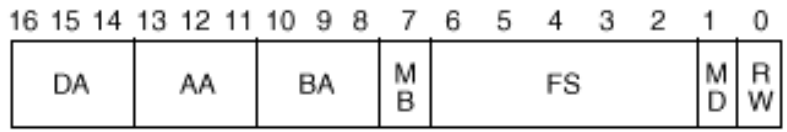
FS	MF Select	G Select	H Select	Microoperation
00000	0	0000	0	$F = A$
00001	0	0001	1	$F = A + 1$
00010	0	0010	0	$F = A + B$
00011	0	0011	1	$F = A + B + 1$
00100	0	0100	0	$F = A + \overline{B}$
00101	0	0101	1	$F = A + \overline{B} + 1$
00110	0	0110	0	$F = A - 1$
00111	0	0111	1	$F = A$
01000	0	1000	0	$F = A \wedge B$
01010	0	1010	0	$F = A \vee B$
01100	0	1100	0	$F = A \oplus B$
01110	0	1110	0	$F = \overline{A}$
10000	1	0000	0	$F = \text{sr } A$
10001	1	0001	1	$F = \text{sl } A$

Computer Datapath



(a) Block Diagram

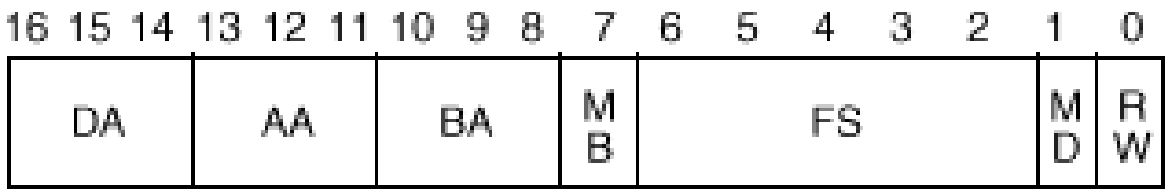
FS	MF Select	G Select	H Select	Microoperation
00000	0	0000	0	$F = A$
00001	0	0001	1	$F = A + 1$
00010	0	0010	0	$F = A + B$
00011	0	0011	1	$F = A + B + 1$
00100	0	0100	0	$F = A + \overline{B}$
00101	0	0101	1	$F = A + \overline{B} + 1$
00110	0	0110	0	$F = A - 1$
00111	0	0111	1	$F = A$
01000	0	1000	0	$F = A \wedge B$
01010	0	1010	0	$F = A \vee B$
01100	0	1100	0	$F = A \oplus B$
01110	0	1110	0	$F = \overline{A}$
10000	1	0000	0	$F = \text{sr } A$
10001	1	0001	1	$F = \text{sl } A$



(b) Control word

Control Word

- The selection variables for the datapath select the microoperation to be executed within the datapath for any given clock pulse.
- Control Word: the combined values of the datapath control inputs in a specified order.
  - Control words consist of multiple *fields*, each of which represents part of the microoperation functionality.
    - e.g. Destination Address, Operand A Address, Operand B Address, Read Immediate/Constant, Function Select, Write Immediate/Constant, Read/Write



(b) Control word

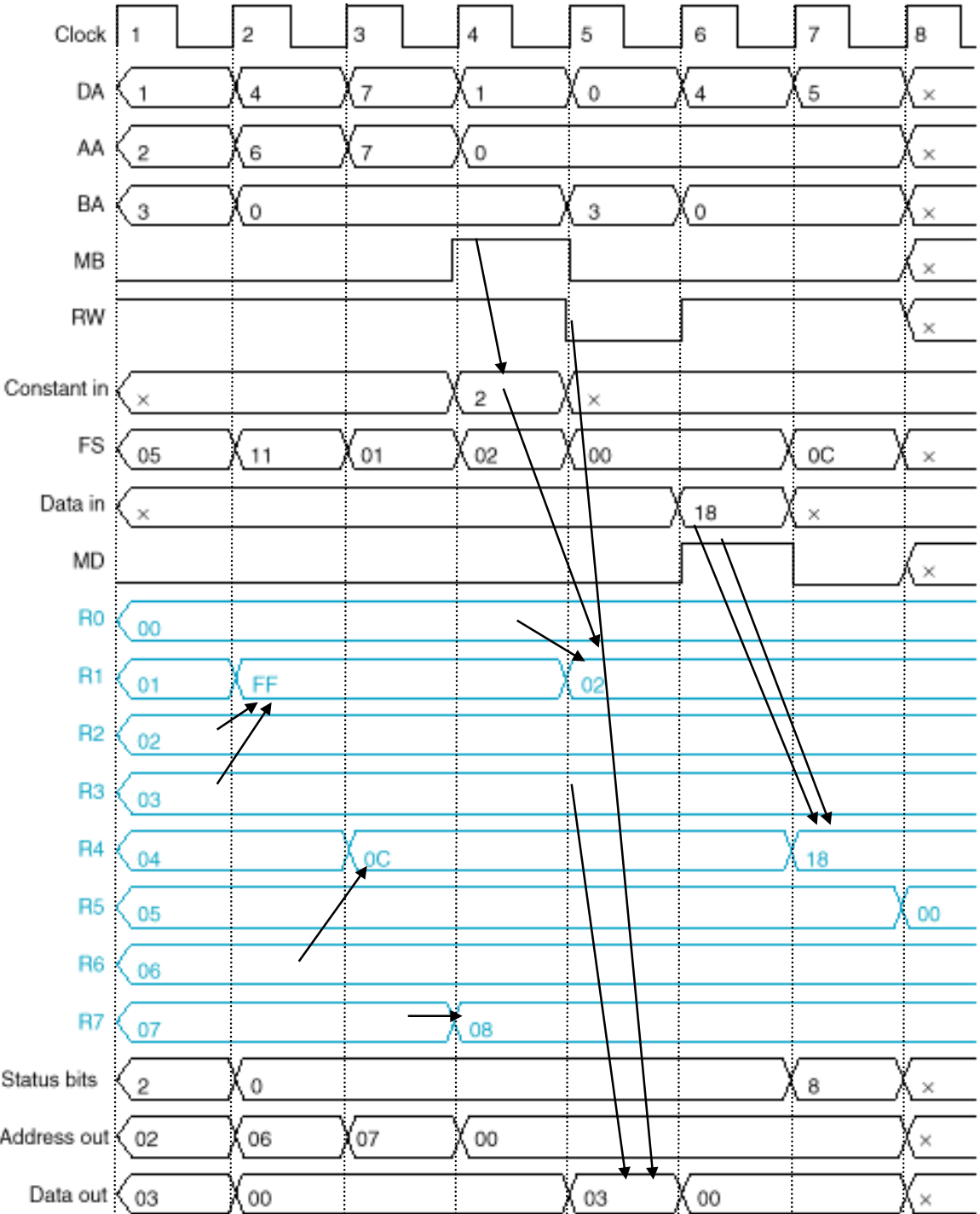
Symbolic Notation

DA, AA, BA		MB		FS		MD		RW	
Function	Code	Function	Code	Function	Code	Function	Code	Function	Code
$R0$	000	Register	0	$F = A$	00000	Function	0	No write	0
$R1$	001	Constant	1	$F = A + 1$	00001	Data In	1	Write	1
$R2$	010			$F = A + B$	00010				
$R3$	011			$F = A + B + 1$	00011				
$R4$	100			$F = A + \overline{B}$	00100				
$R5$	101			$F = A + \overline{B} + 1$	00101				
$R6$	110			$F = A - 1$	00110				
$R7$	111			$F = A$	00111				
				$F = A \wedge B$	01000				
				$F = A \vee B$	01010				
				$F = A \oplus B$	01100				
				$F = \overline{A}$	01110				
				$F = \text{sr } A$	10000				
				$F = \text{sl } A$	10001				

Symbolic Notation to Control Word

Micro-operation	DA	AA	BA	MB	FS	MD	RW
$R1 \leftarrow R2 - R3$	$R1$	$R2$	$R3$	Register	$F = A + \overline{B} + 1$	Function	Write
$R4 \leftarrow$ sl R6	$R4$	$R6$	—	Register	$F =$ sl A	Function	Write
$R7 \leftarrow R7 + 1$	$R7$	$R7$	—	Register	$F = A + 1$	Function	Write
$R1 \leftarrow R0 + 2$	$R1$	—	—	Constant	$F = A + B$	Function	Write
Data out $\leftarrow R3$	—	—	$R3$	Register	—	—	No Write
$R4 \leftarrow$ Data in	$R4$	—	—	—	—	Data in	Write
$R5 \leftarrow 0$	$R5$	$R0$	$R0$	Register	$F = A \oplus B$	Function	Write

Micro-operation	DA	AA	BA	MB	FS	MD	RW
$R1 \leftarrow R2 - R3$	001	010	011	0	00101	0	1
$R4 \leftarrow$ sl R6	100	110	000	0	10001	0	1
$R7 \leftarrow R7 + 1$	111	111	000	0	00001	0	1
$R1 \leftarrow R0 + 2$	001	000	000	1	00010	0	1
Data out $\leftarrow R3$	000	000	011	0	00000	0	0
$R4 \leftarrow$ Data in	100	000	000	0	00000	1	1
$R5 \leftarrow 0$	101	000	000	0	01100	0	1



Timing Diagram

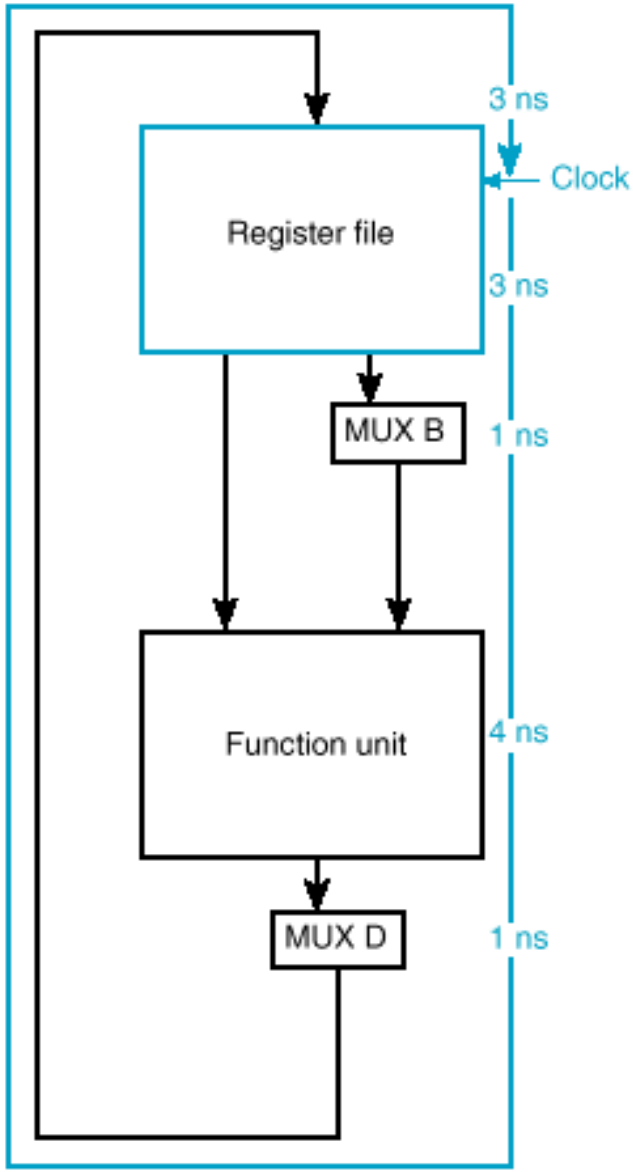
Micro-operation	DA	AA	BA
$R1 \leftarrow R2 - R3$	001	010	011
$R4 \leftarrow$ sl R6	100	110	000
$R7 \leftarrow R7 + 1$	111	111	000
$R1 \leftarrow R0 + 2$	001	000	000
Data out $\leftarrow R3$	000	000	011
$R4 \leftarrow$ Data in	100	000	000
$R5 \leftarrow 0$	101	000	000

MB	FS	MD	RW
0	00101	0	1
0	10001	0	1
0	00001	0	1
1	00010	0	1
0	00000	0	0
0	00000	1	1
0	01100	0	1

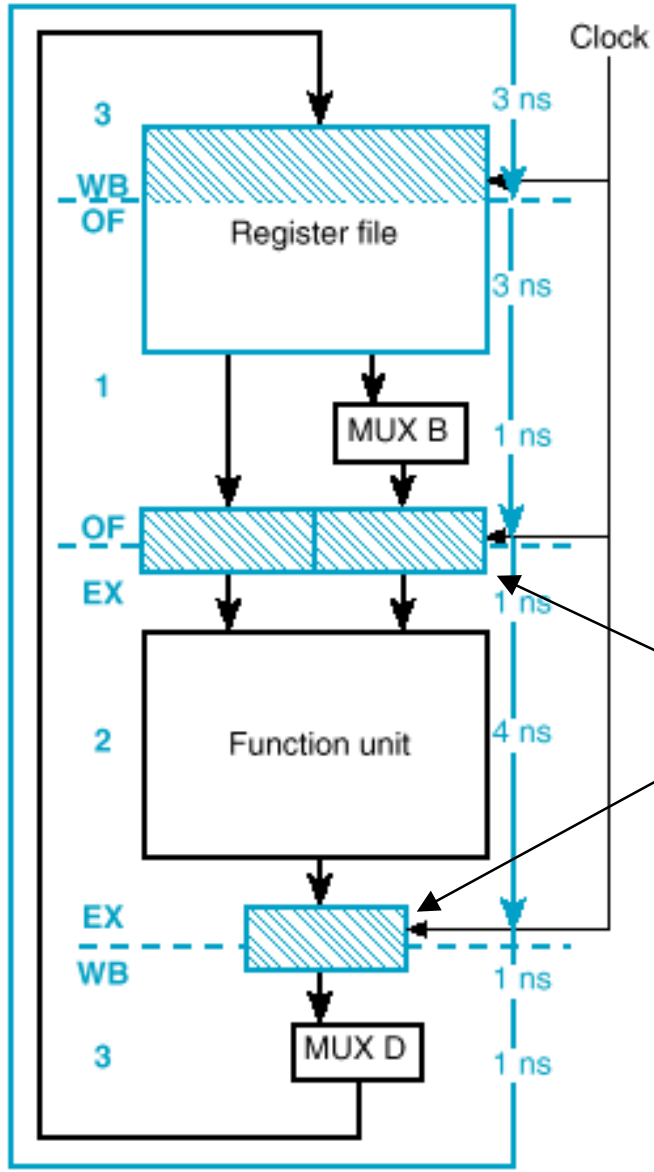
Pipelined Datapath

- Datapath throughput can be increased by breaking up the datapath with registers and increasing the clock speed.
  - This is known as pipelining the datapath.
- Throughput vs. Turnaround time
  - Throughput: The number of operations (units) per second (time period).
  - Turnaround time: The amount of time which elapses from the beginning of the operation's execution (unit processing) to its completion.
  - Consider an assembly line (or fast-food drive-thru!)
- Basic stages of a datapath microoperation:
  - Operand Fetch (OF)
  - Execute Function (EX)
  - Write Back Results (WB)

Datapath Timing



(a) Conventional



(b) Pipelined

Stage One

Stage Two

Pipeline Platforms

Stage Three

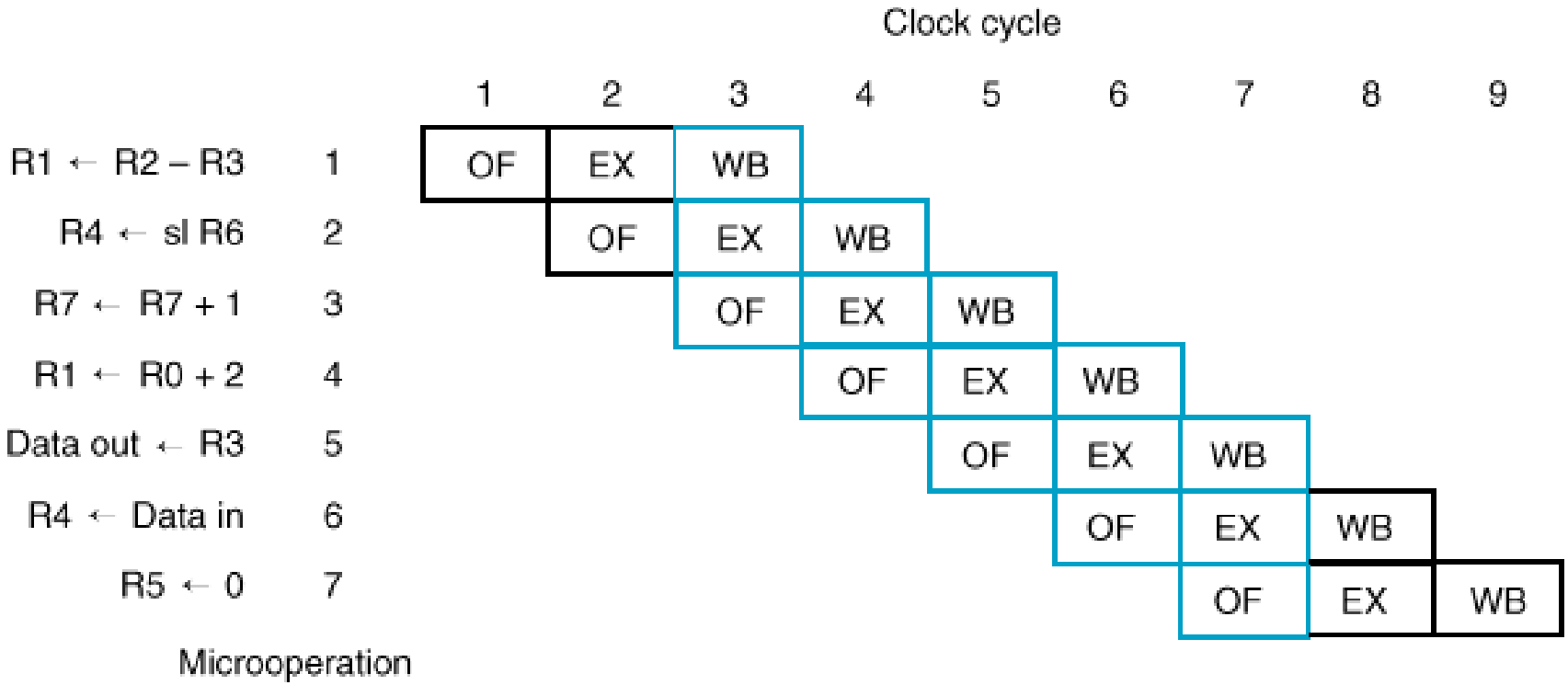
Min. Clock Period = 12 ns

83.3 MHz

Min. Clock Period = 5 ns

200 MHz

Pipeline Execution Pattern



- Not all of the pipelined units are necessarily active at all times
  - Filling and emptying the pipeline "wastes" time
  - Hazards exist!

The Control Unit

- The control unit generates the signals for sequencing the operations in the datapath
  - A sequential circuit with states that *dictate the control signals* for the system
  - Using status conditions and control inputs, the sequential control unit *determines the next state* in which additional microoperations are activated.
- Hardwired Control
  - The control unit is implemented to provide a particular digital function
- Microprogrammed Control
  - The control unit's binary control values are stores as words in a microprogrammed control (usually ROM).
  - Each word in the control contains a microinstruction
  - A sequence of microinstructions constitutes a microprogram
  - Firmware!

Diagram of a Hardwired Control Unit

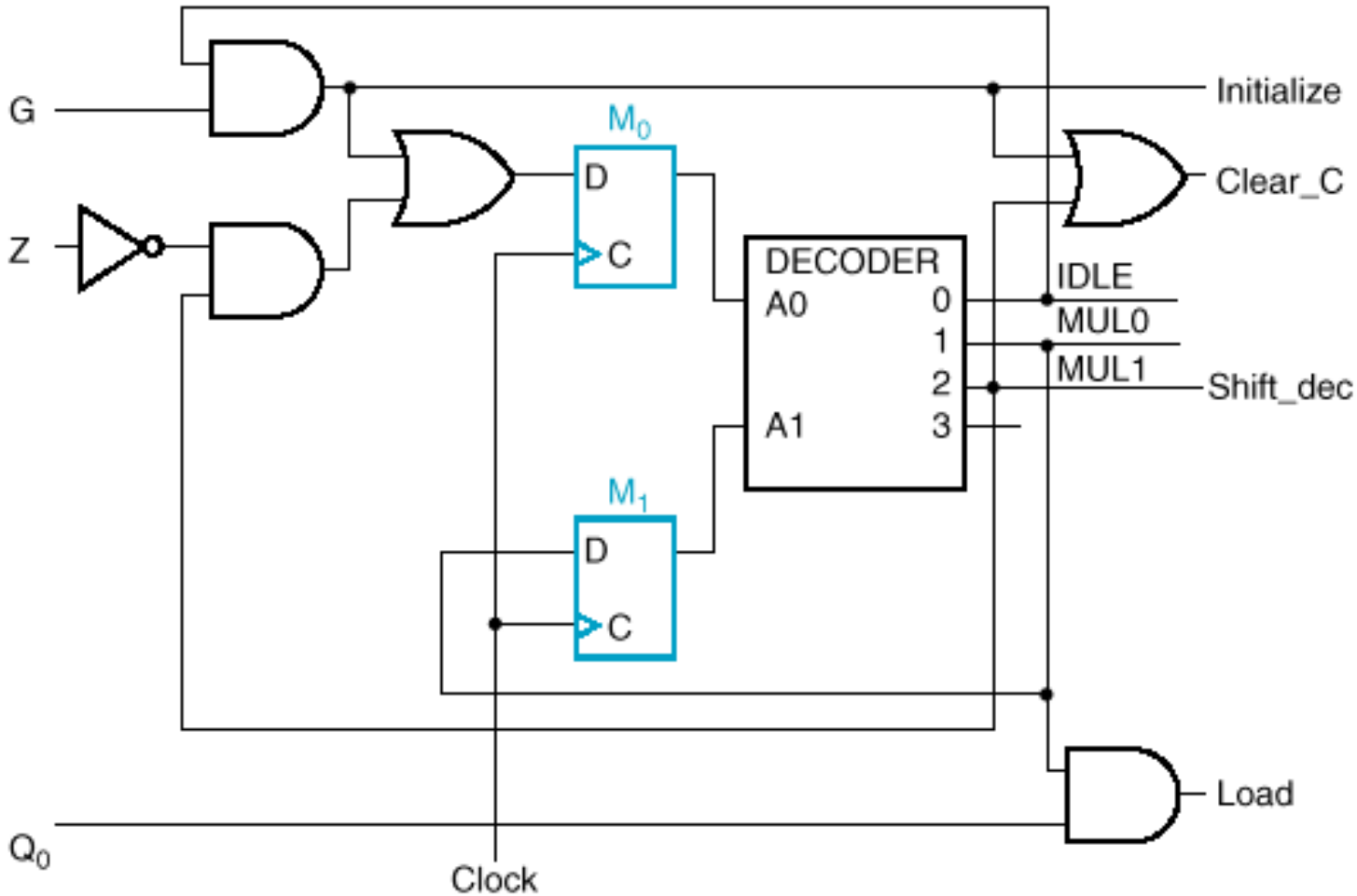
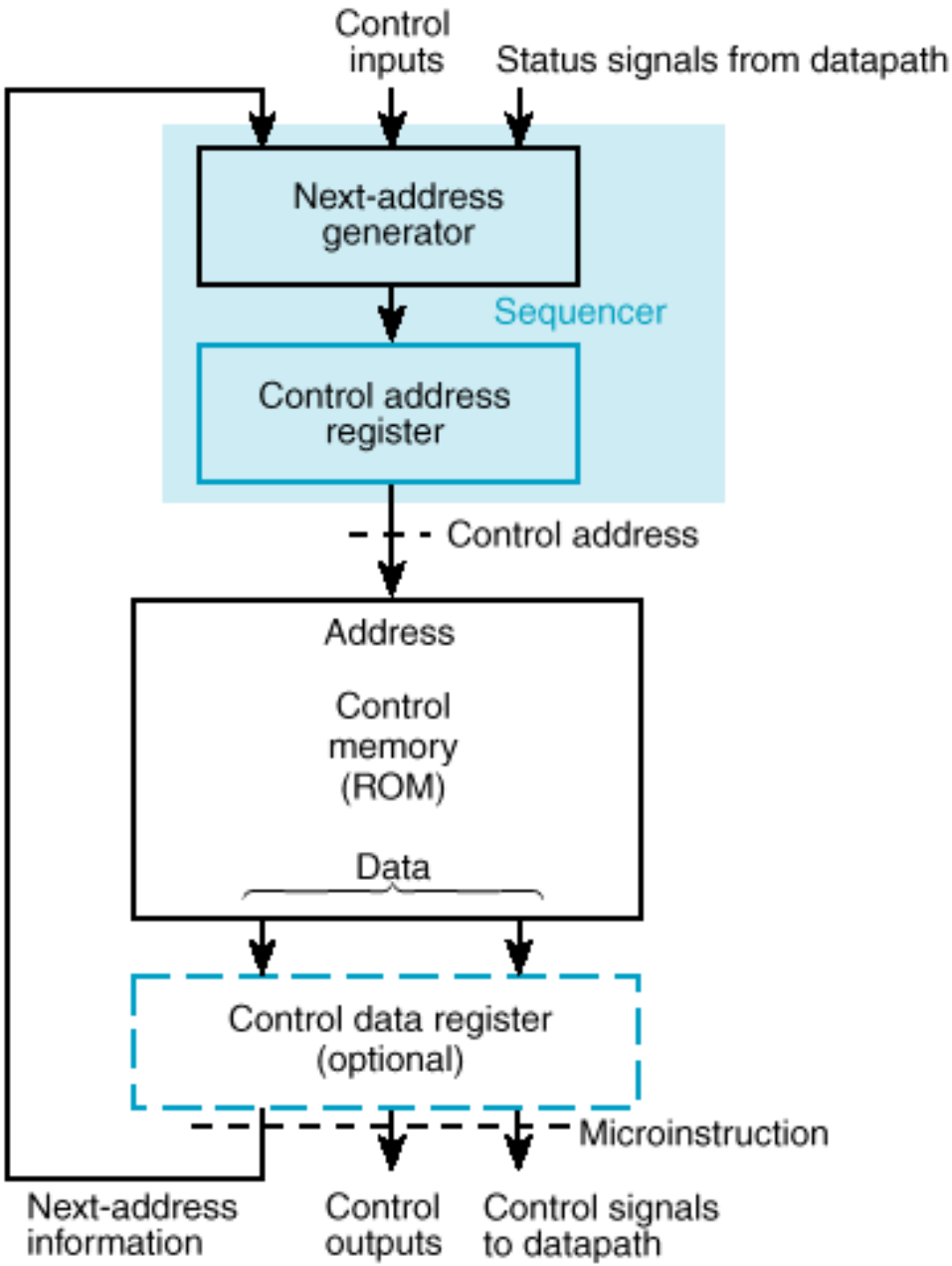


Diagram of a Microprogrammed Control Unit

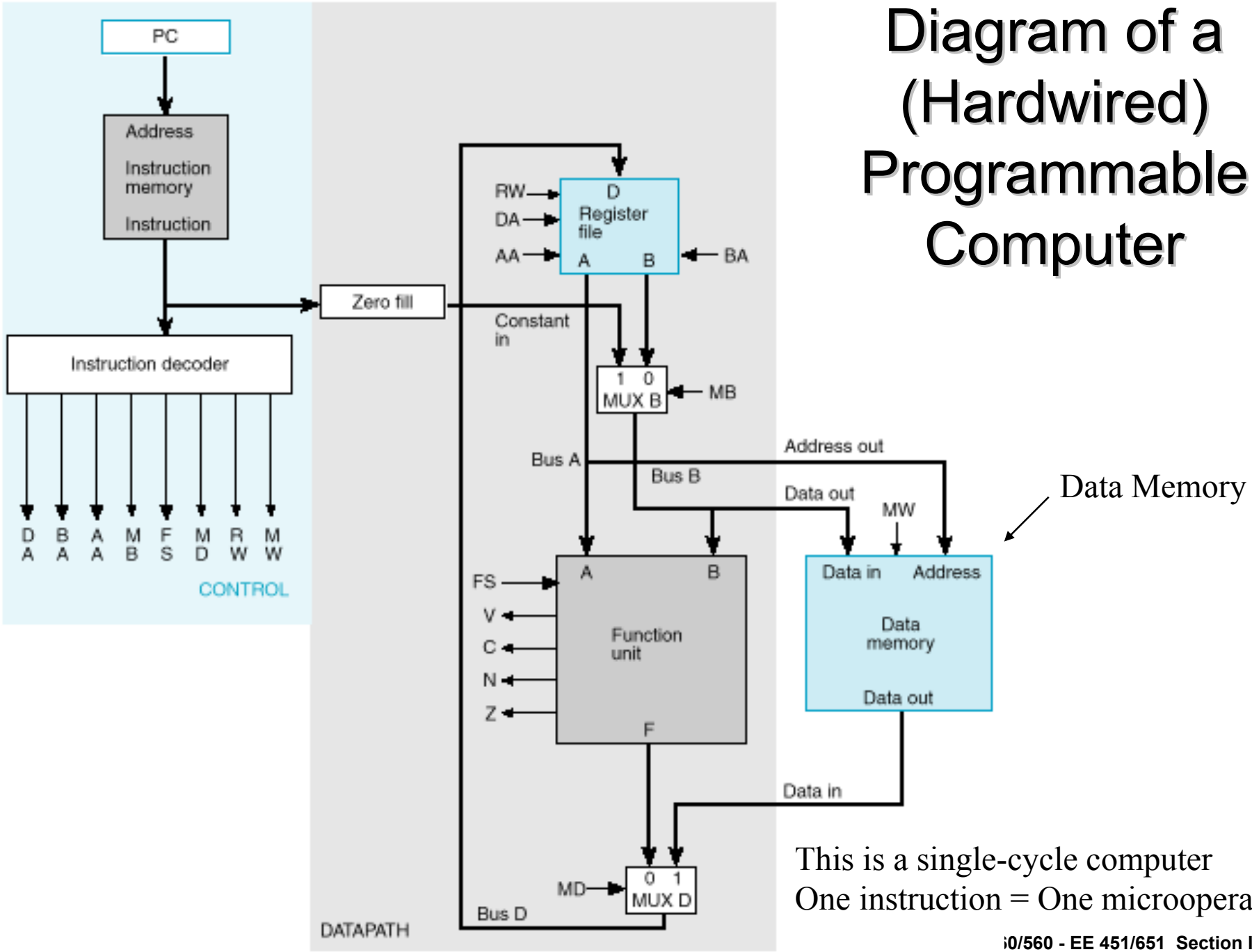


- The values of the control signals (and outputs) are determined by the contents of the Control Memory (a.k.a. the Control Store)
- A portion of the contents of the Control Memory is used (along with the next set of inputs).
  - The "next-address" field maintains internal state

Programmable Control Units

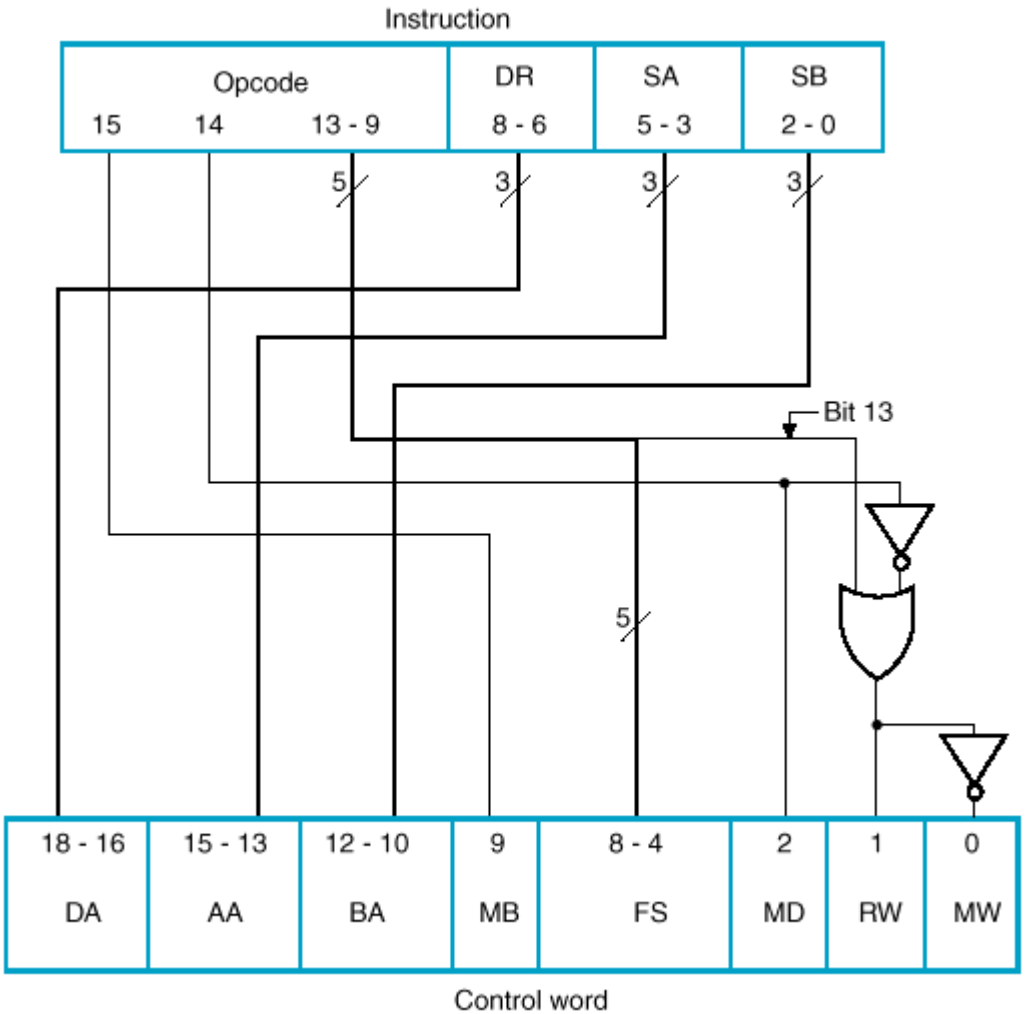
- The binary information stored in a digital computer can be classified as either data or control information (machine language instructions)
- Non-programmable Control Unit
  - A control unit which not responsible for obtaining instructions from memory, it determines the operations to be performed and the sequence of those operations based only upon its inputs and status bits
- Programmable Control Unit:
  - A portion of the input to the system consists of a sequence of instructions.
  - Each instruction contains the information necessary for the control unit to determine a sequence of microoperations or which instruction to execute next
  - The address of the next instruction comes from a Program Counter (PC)
    - The PC can count (+1 instruction, normal operation)
    - The PC can load (branch instruction)
- Control Units (Programmable or Non) can be single-cycle or multi-cycle
  - If any instruction requires more than one microoperation the machine is multi-cycle.
    - How does this complicate the design?

Diagram of a (Hardwired) Programmable Computer



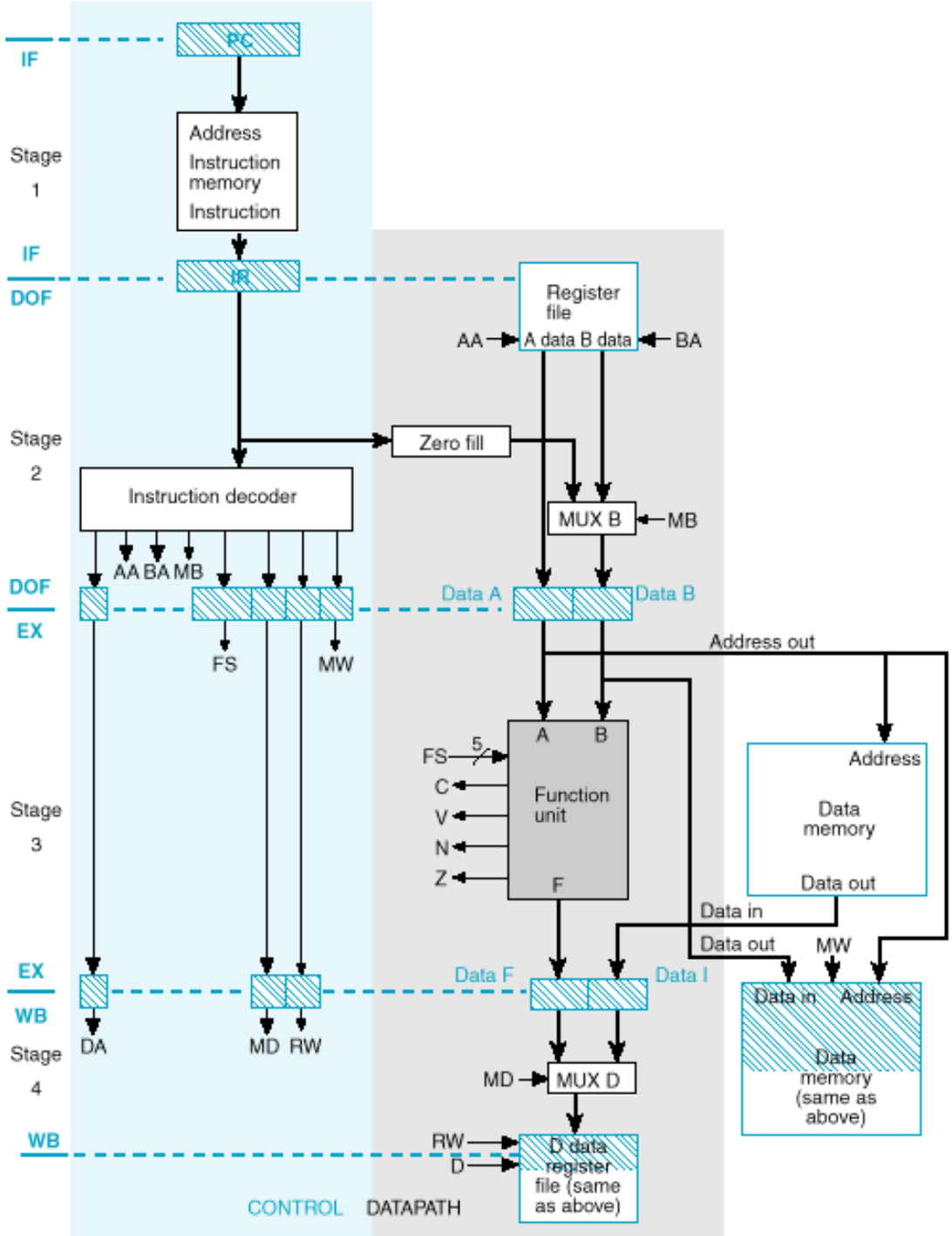
This is a single-cycle computer
One instruction = One microoperation

Diagram of a Instruction Decoder



- The instruction decoder takes the inputs to the control unit (it this case an "instruction") and creates a corresponding control word for a datapath microoperation.
- This might be used in a single-cycle computer

Diagram of a Pipelined Control Unit



Basic stages of instruction execution:

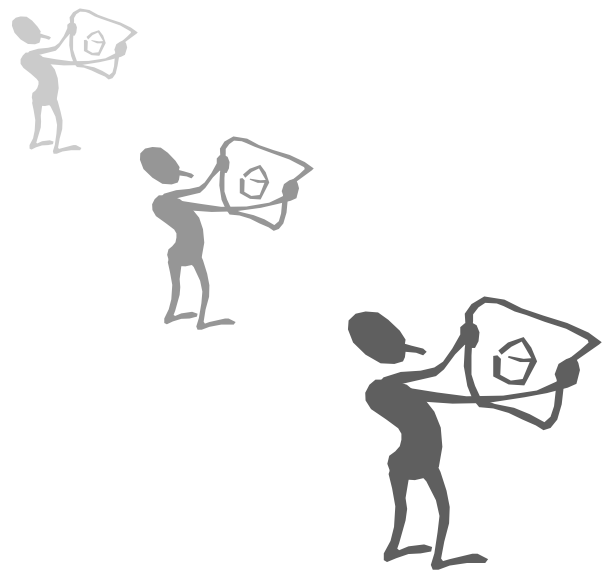
- Instruction Fetch (IF)
- Decode & Operand Fetch (DOF)
- Execute Function (EX)
- Write Back Results (WB)

More stages are possible!

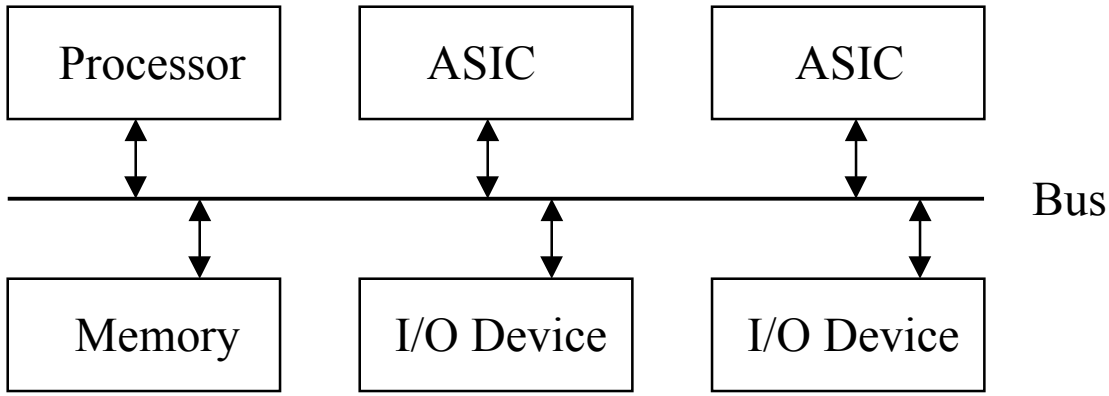
Relate this to the "Machine-cycle"

Outline

- Complex System Design
  - Register-Transfer Level Design
  - Data paths
    - The Control Word
    - Pipelining
  - Control unit
    - Hardwired Control
    - Microprogrammed Control
    - Programmable Control Units
- **Simple Computer Architecture**
  - **Computer Instructions**
  - **Instruction Set Architecture**
- Issues in Computer Design
  - CISC vs. RISC
  - Memory Hierarchy

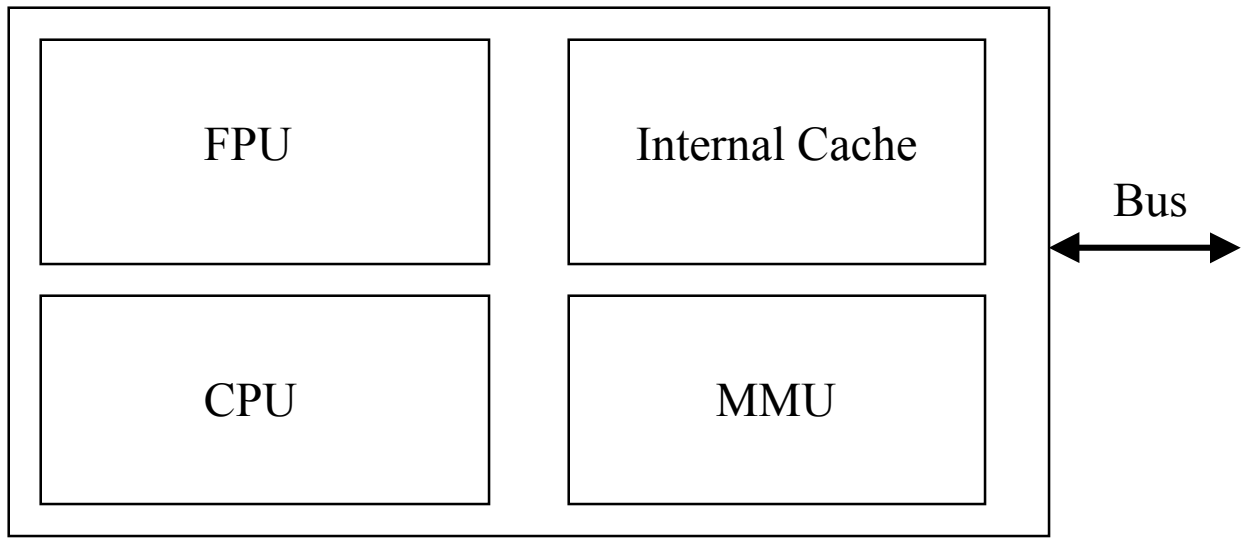


A simple computer architecture



- Generic Computer System.
  - Current architectures are performance driven, and vary widely.
- Processor
  - Uniprocessor systems
- ASIC (Application Specific Integrated Circuit)
  - Performs a specific task, not a general purpose processor (e.g. Voodoo)
- I/O Device
  - Accesses data devices (e.g. Graphics Adapter, Disk Controller, et al.)

A simple microprocessor



- Central Processing Unit
  - Control Unit, Integer Datapath (Load/Store, Integer ALU)
- Floating Point Unit
  - Floating Point Datapath
- Internal Cache
  - SRAM for Instruction Cache (i-cache) and Data Cache (d-cache)
- Memory Management Unit
  - Controls communication with Main Memory and other I/O

Instruction Set Architecture

- Microprocessors can only perform certain operations
  - Users determine which operations will be performed and in what order though the use of a program.
  - A *program* consists of a sequence of machine-executable instructions.
  - An *instruction* is a collection of bits that instructs the computer to perform a specific operation.
  - The set of instructions that a particular microprocessor can execute is its *instruction set.*
  - *Instruction set architecture:* A thorough description of the instruction set of a computer.
- Users can not easily produce meaningful programs using the instruction set directly.
  - Compilers convert programs specified in high-level languages into the instruction set equivalent (a *machine language* program).

Computer Instructions (1)

- High-Level Language - C
  - A = B + C;
  - Memory-Transfer Equivalent
    - Mem[A] ← Mem[B] + Mem[C]
    - Mem[EA00] ← Mem[EA08] + Mem[EA10]
- Machine-Level Equivalent
  - Assembly (human readable) ex: Machine (for a simple architecture)
    - Load R2, B E2EA08
    - Load R3, C E3EA10
    - R2 ← R2 + R3 0223
    - Store A, R2 F2EA00
- The bits of a machine instruction are divided into *fields*
  - eg: E2EA08
  - E: Operation "Load"; 2: Destination Address R2; EA08: Address Field
  - The operation field (*opcode*) defines the format for the instruction

Computer Instructions (2)

- There are three basic types of computer instructions
  - Register Instructions: operate on values stored in registers
    - Arithmetic, Shift, and Logic instructions
  - Move Instructions: move data between memory and registers
    - Load/Store instructions
    - Move/Copy portions of memory
  - Branch Instructions: select one of two possible next instructions to execute
    - Branch on condition, Unconditional branch (Jump)
    - Only one address is *explicit*, the other operand is *implict*
      - e.g.: Beq R2, R3, A
      - If the contents of R2 = R3 then execute the instruction at location A next (explict)
      - otherwise, execute the next instruction in the normal order (using the PC) (implict)

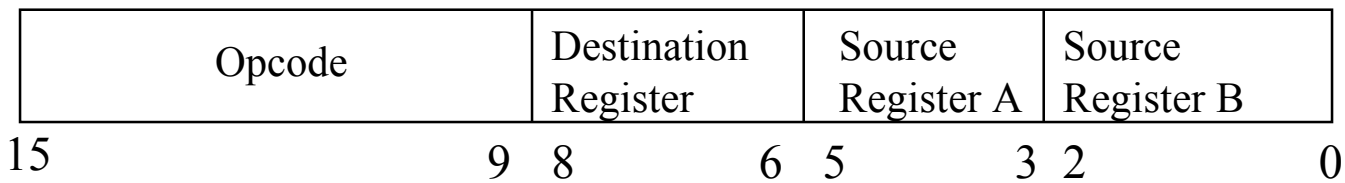
Instructions Vs. Microoperations

- What is the difference between a computer instruction and a hardware microoperation?
  - Computer instruction: an operation stored in binary in the computer's memory
  - The control unit uses the address or addresses provided by the program counter (PC) to retrieve the next instruction from memory
  - The control unit then decodes the instruction fields to perform the required microoperations for the execution of the instruction.
  - Thus, in microprogrammed control, each computer instruction corresponds to a microprogram!

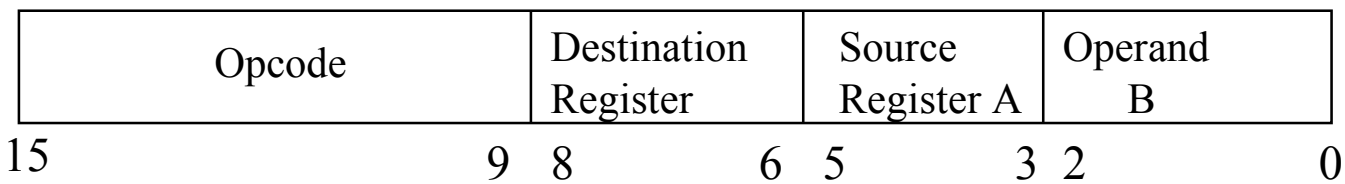
Instruction Formats

- Different Instructions have different types of instruction formats
  - Register, Implied, Immediate, Direct, Indirect, Relative, Indexed

- Register: operands are hardware registers (e.g. Add R3, R2, R1)



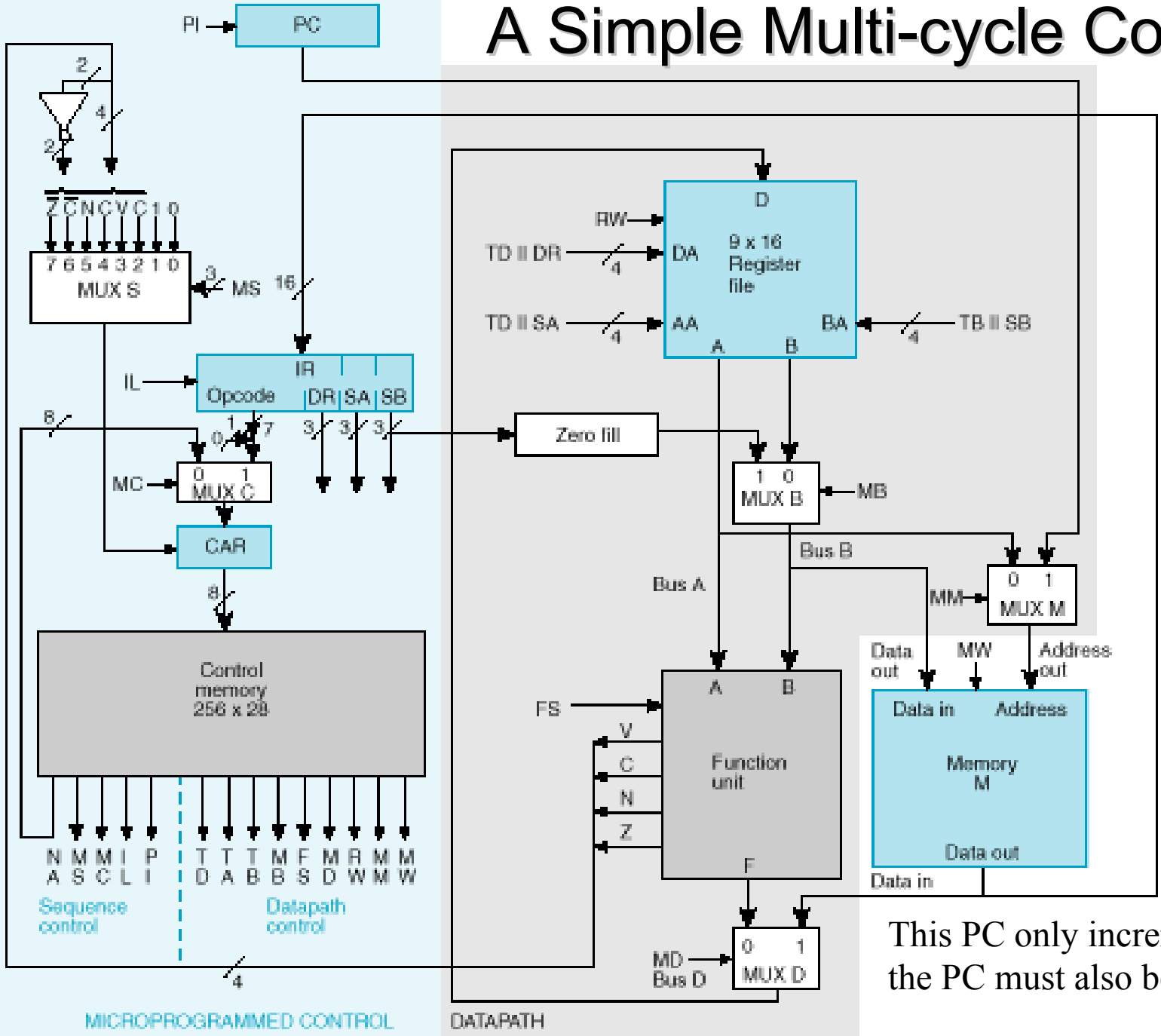
- Immediate: one operand is a constant (eg. Add R2, 3)



The Instruction-execution Cycle

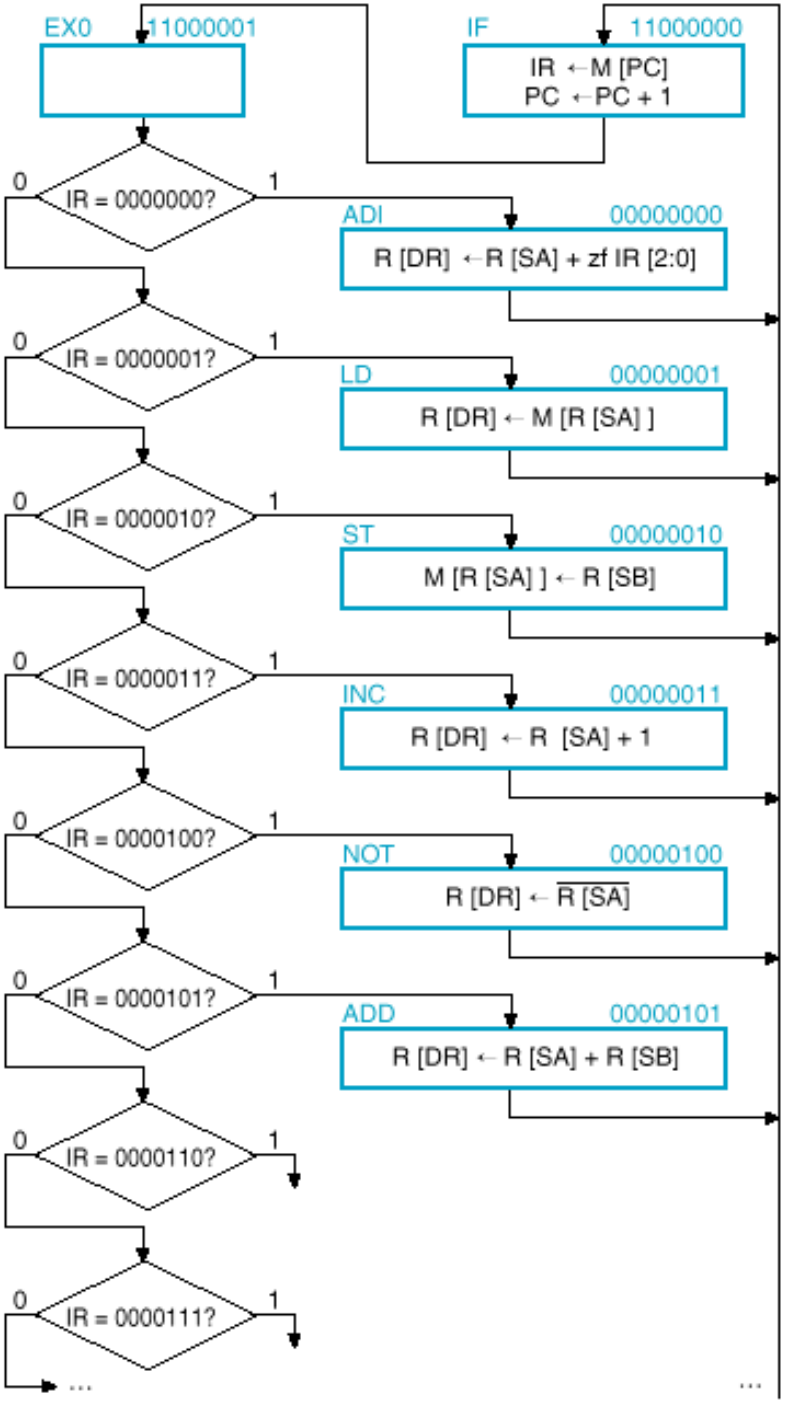
- Instruction Fetch (IF) stage
  - Get next instruction from the memory address referenced by the PC
  - Place the new instruction in the Instruction Register
  - Increment the PC to the next instruction address
- Instruction decode (ID) stage
  - The instruction is recognized, or decoded
  - Determine the instruction format by examining the opcode
- Operand fetch (OF) stage
  - Perform any calculations necessary to fetch the operand values
  - If necessary, fetch operands from memory to temporary registers
- Execute operation (EX) stage
  - Execute the operation specified in the opcode
    - Branch instructions may update the PC
- Writeback (WB) stage
  - Store the result of the operation in as determined by the instruction
- Repeat the instruction-execution cycle (a.k.a. the machine cycle).

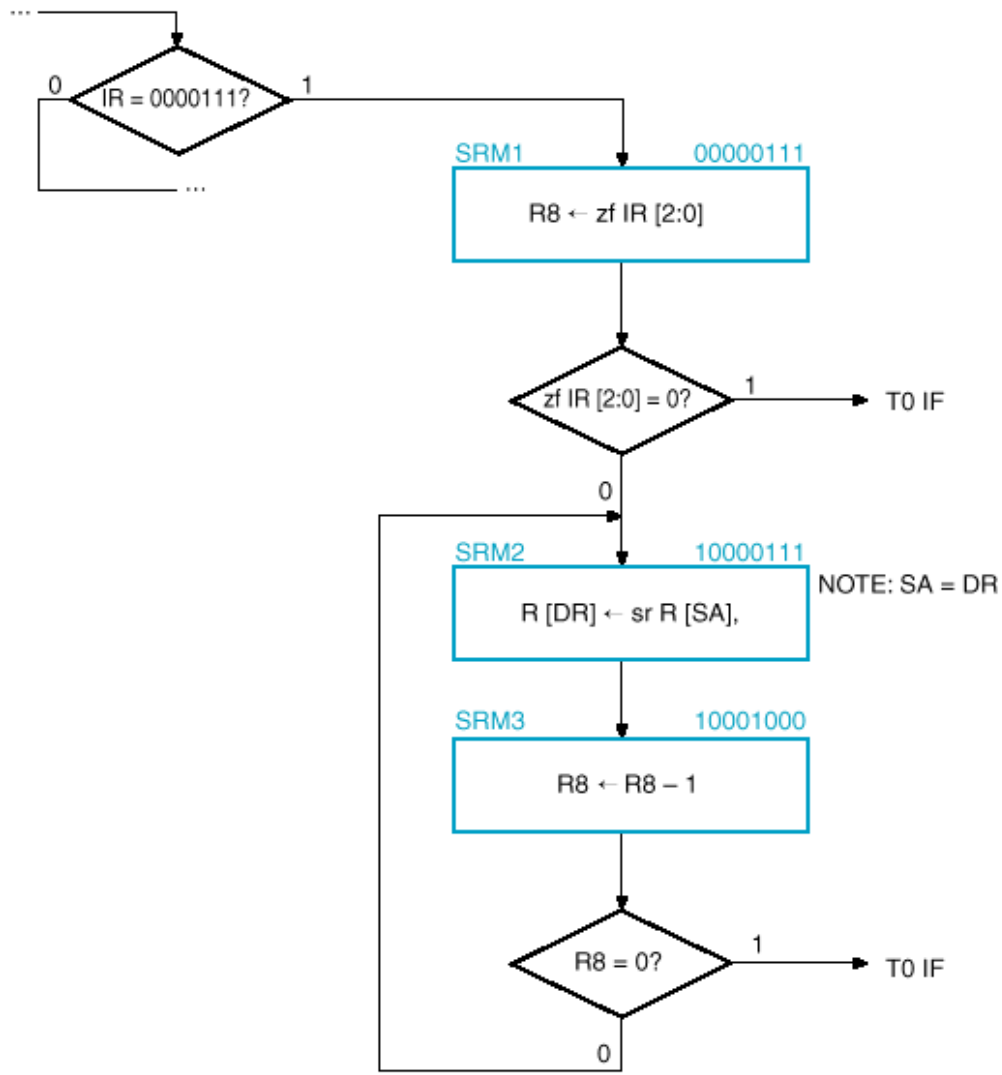
A Simple Multi-cycle Computer



This PC only increments… usually the PC must also be able to load

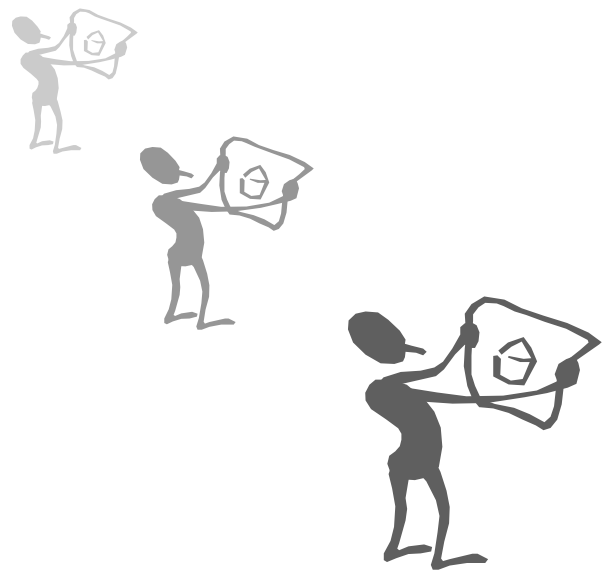
Microprogram Design



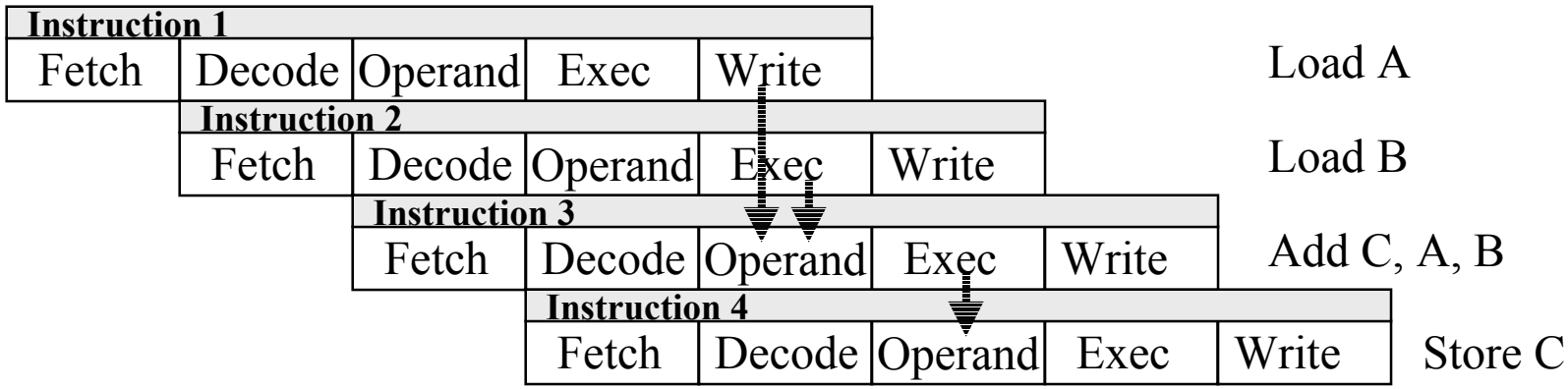


Outline

- Complex System Design
  - Register-Transfer Level Design
  - Data paths
    - The Control Word
    - Pipelining
  - Control unit
    - Hardwired Control
    - Microprogrammed Control
    - Programmable Control Units
- Simple Computer Architecture
  - Computer Instructions
  - Instruction Set Architecture
- **Issues in Computer Design**
  - **CISC vs. RISC**
  - **Memory Hierarchy**

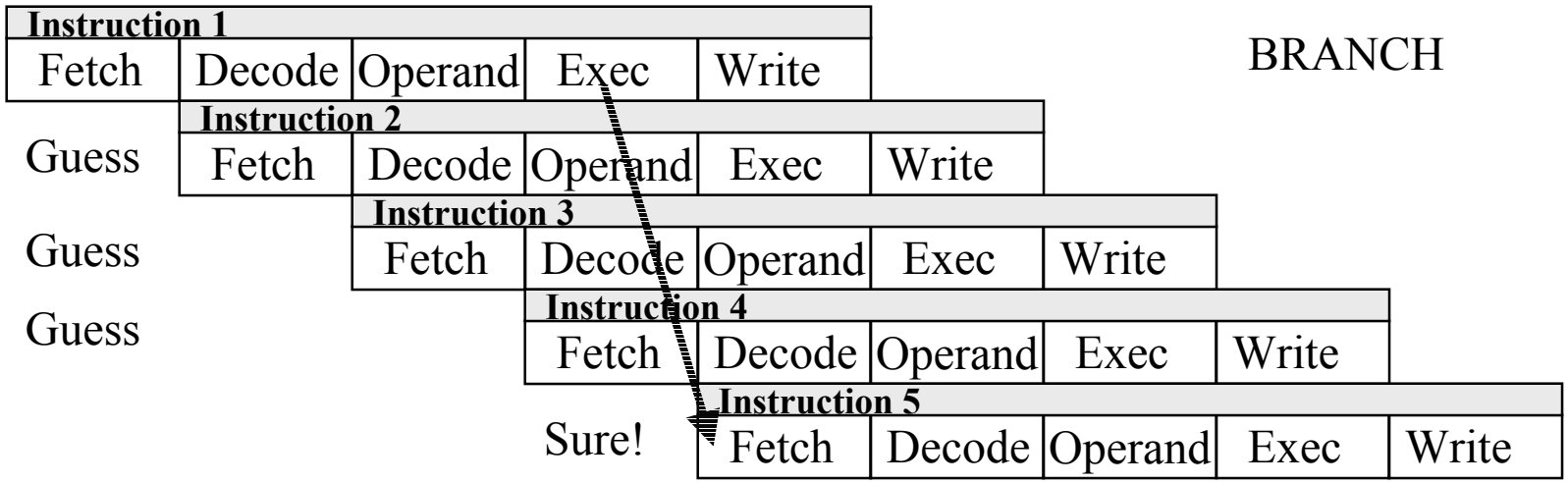


Pipeline Hazards (1)



- Basic Pipelining Hazards
  - Data Dependency - occurs whenever one instruction generates a value that is used by one or more successive instructions.
    - Data Forwarding - a technique to avoid delays due to data dependency hazards by allowing data to skip one or more pipeline stages

Pipeline Hazards (2)



- Control Dependency - occurs whenever a branch instruction is encountered; the address of the "next" instruction can not be determined until the branch conditions is evaluated during the execute phase.
  - Branch Prediction - reduces delays due to control dependencies by making a educated guess and "anulling" the operations in the pipeline if the guess is incorrect.
  - Most branches backwards are "taken", most branches forward are "not taken".
  - 2-bit history (traditional technique): If the guess is wrong twice in a row, change the future guess for that instruction in the future.
  - Trace caches: proposed designs use instruction traces to keep track of past program flow. Some proposals use 50% of the chip area for this cache. Why?

CISC

- Complex Instruction Set Computers (CISC) - 1950's +
  - Made up of powerful primitives (close to the primitives of high-level languages).
  - Had *many* computer instructions.
  - Why?
    - Early compilers did not produce the fastest, most memory efficient code!
    - Most "respectable" programmers understood and occasional programmed modules in the machine's instruction set.
    - Powerful primitives made this task user-friendly.
  - Performance/Costs
    - The memory footprint of the code was very small!
    - The hardware to implement these instructions was very complicated.
    - Due to the large number of different tasks, CISC machines are difficult to fully pipeline.

RISC

- Reduced Instruction Set Computers (RISC) - 1980's +
  - Minimal instruction set
    - all instructions use simple addressing modes to reduce decoding difficulty
    - most instructions require one clock tick to execute (simplifies pipelining)
  - Why?
    - More chip space is devoted to making the most common instructions as fast as possible. Some less common instructions may be slower, but overall performance is increased.
    - Compilers have improved to the point that well-optimized sequences of simple (very fast) commands often outperform the more complicated multi-cycle instructions.
    - RISC designs could fit on a single chip, which reduced cost, increased reliability, and increased clock speeds!
  - Performance/Costs
    - Larger code footprints, more/larger instructions that CISC.
    - Initially RISC ISA could not include support for floating point instructions. They had to be performed in software and were very slow. As chip densities increased floating point instructions were added to the RISC ISAs.

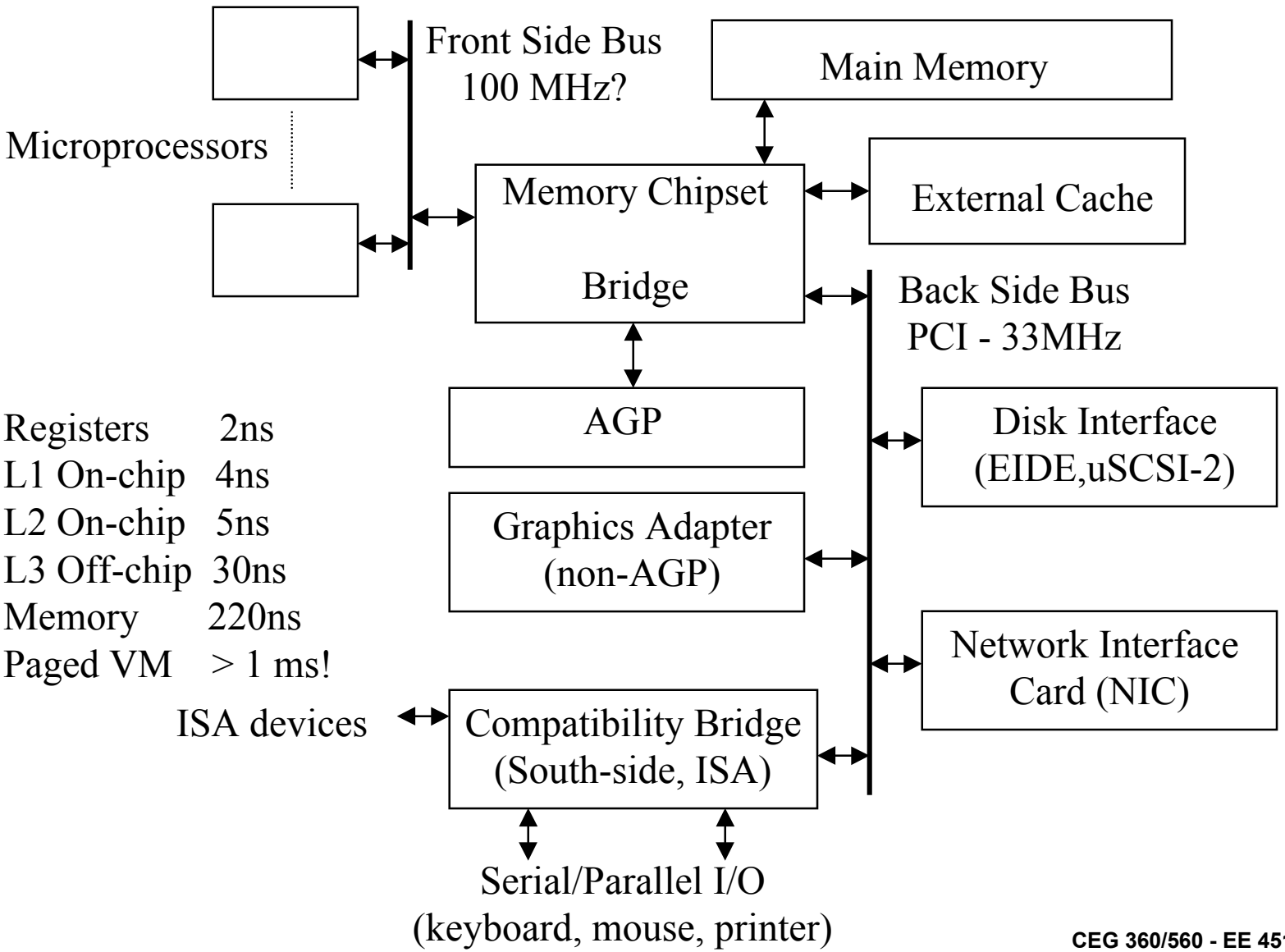
Characteristics of RISC (Vs. CISC) (1)

- RISC is a design philosophy (Fast is good!)
- Common RISC features:
  - Instruction pipelining
  - Uniform instruction length
    - This assumption simplifies instruction fetch, decode, and pipeline design
  - Delayed branching
    - Branches are not decided until the execute phase… anything in the pipeline after the branch may have to be annulled if the "next address" is incorrect.
    - A branch occurs every five to ten instructions in many programs.
    - If we executed a branch every fifth instruction and only half of our branches fell through, the lost efficiency due to restarting the pipeline after the branches would be 20%!
    - Increased pipeline efficiency by requiring an instruction in the branch delay slot (which may be a No-Op).
    - Instruction in the branch delay slot is executed regardless of the branch decision.
    - Branch prediction and the ability to annul instructions in the pipeline has greatly increased the efficiency of branches.

Characteristics of RISC (Vs. CISC) (2)

- Load/Store architecture
  - The only memory access allowed are Load and Store! All other operations are register-register.
  - Otherwise it would be very difficult to have uniform length instructions
  - Instructions that did both a memory fetch and a operation would require two execution stages, one to calculate the address, and one to perform the operation. This way, each instruction has exactly one executionstage.
  - Load/Stores can take more time than other instructions, if we are careful to load registers well before we need them, we can often reorder instructions so that the pipeline is always full.
- Simple addressing modes
  - Necessary to reduce the ISA and allow uniform instruction lengths.
  - Some memory references may take more real instructions than they might have taken on a CISC machine, and be more difficult for a human to understand, but because everything executes more quickly, it is generally still a performance win.

A contemporary RISC architecture



Registers	2ns
L1 On-chip	4ns
L2 On-chip	5ns
L3 Off-chip	30ns
Memory	220ns
Paged VM	> 1 ms!

Memory Organization (1)

- Goal: Access your instructions and data as quickly as possible
- Problems:
  - The larger the memory, the slower the access time
  - The larger the memory, the more expensive the memory
  - The faster the memory, the more expensive the memory
- The Principles of Temporal and Spatial Locality:
  - At any given point in its execution, a program tends to utilize a small portion of memory (its *working set*)
- The Memory Hierarchy
  - The memory subsystem is composed of increasingly larger, slower memory.
  - Keep the *working set* of the program as close to the CPU as possible!
    - Data consistency becomes a problem! How do we keep multiple copies of the same information consistent?

Memory Organization (2)

- An example hierarchy:
  - General Purpose 32-bit Registers: register file, 2ns
  - L1: 16k instruction cache, 16k data cache: SRAM, 4ns
  - L2: 256k on-chip cache: SRAM (farther from CPU), 5ns
  - L3: 512k off-chip cache: SRAM, 30ns (delay due to overhead)
  - 128MB main memory: 60ns EDO DRAM, 220ns (delay due to overhead)
  - 8GB secondary storage: 10ms hard disk drive, >> 1ms
  - The Internet?: 100 Mbs NIC, >> 1s

- Memory is not advancing as quickly as microprocessor technology.
- The performance bottleneck is now memory bandwidth.

Modern RISC Processors

- Second-generation RISC processors have taken advantage of improved manufacturing processes to:
  - 1. Make the clock rate faster.
  - 2. Duplicate functional units to allow parallel execution of instructions.
    - Superscaler!
  - 3. Increase the number of stages in the pipeline.
    - Superpipelined!
- Modern RISC processors are now introducing:
  - 1. More addressing modes
  - 2. Specialized instructions (MMX)
  - 3. Out-of-order speculative execution!
- What will the next generation introduce?
  - Speculative Data, Processor in Memory (PIM), Trace Caches, et al.
  - ??