Processor Micro-Architecture: Part 1

Mark McDermott

Fall 2009

Agenda

- Taxonomy of computing elements
- Datapath Design
- Control Unit Design

TLAs

- TLA – Three Letter Acronym
- RTN – Register Transfer Notation
- RTL – Register Transfer Language
- MAC – Multiply Accumulate
- ISA – Instruction Set Architecture
- CPI – Clocks per Instruction
- IPC – Instructions per clock
- ALU – Arithmetic Logic Unit
- FSM – Finite State Machine
- L2 – Level Two Cache
- GPP - General Purpose Processor
- FPGA – Field Programmable Gate Array
- ASIC - Application Specific Integrated Circuits
- VM – Virtual Memory
- VA – Virtual Address
- PA – Physical Address
- CPI – Clocks per Instruction
- BTC – Branch Target Cache
- BTAC – Branch Target Address Cache
- LRU – Least Recently Used
- FIFO - First-In-First-Out
- NMRU - Not most recently used
- PMI – Page miss per instruction
- PF – Page Fault

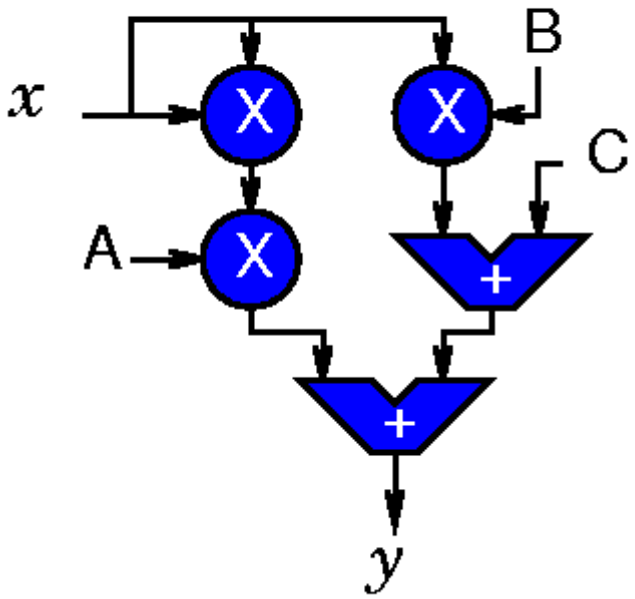
Taxonomy of Computing Elements

- **General Purpose Processors (GPPs)**
  - Intended for general purpose computing (desktops, servers, clusters..)
- **Application-Specific Processors (ASPs)**
  - Processors with ISAs and architectural features tailored towards specific application domains:
    - Digital Signal Processors (DSPs), Network Processors (NPs), Media Processors, Graphics Processing Units (GPUs), Vector Processors ...
- **Co-Processors: A hardware implementation of specific algorithms with limited programming interface.**
- **Configurable Hardware:**
  - Field Programmable Gate Arrays (FPGAs)
  - Configurable array of simple processing elements
- **Application Specific Integrated Circuits (ASICs)**
  - A custom VLSI hardware solution for a specific computational task

Spatial vs. Temporal Computing

Spatial

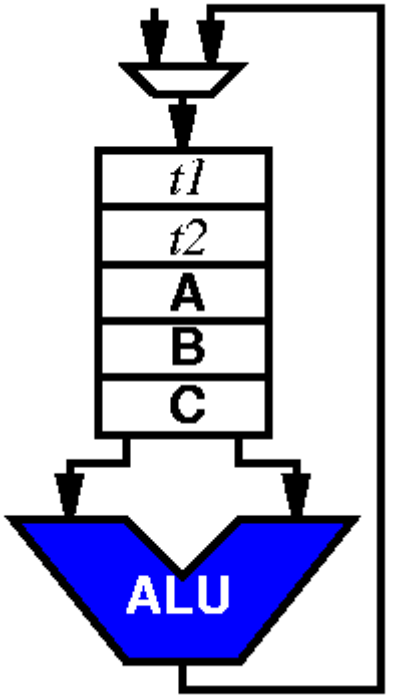
(Hardware Accelerator)



Temporal

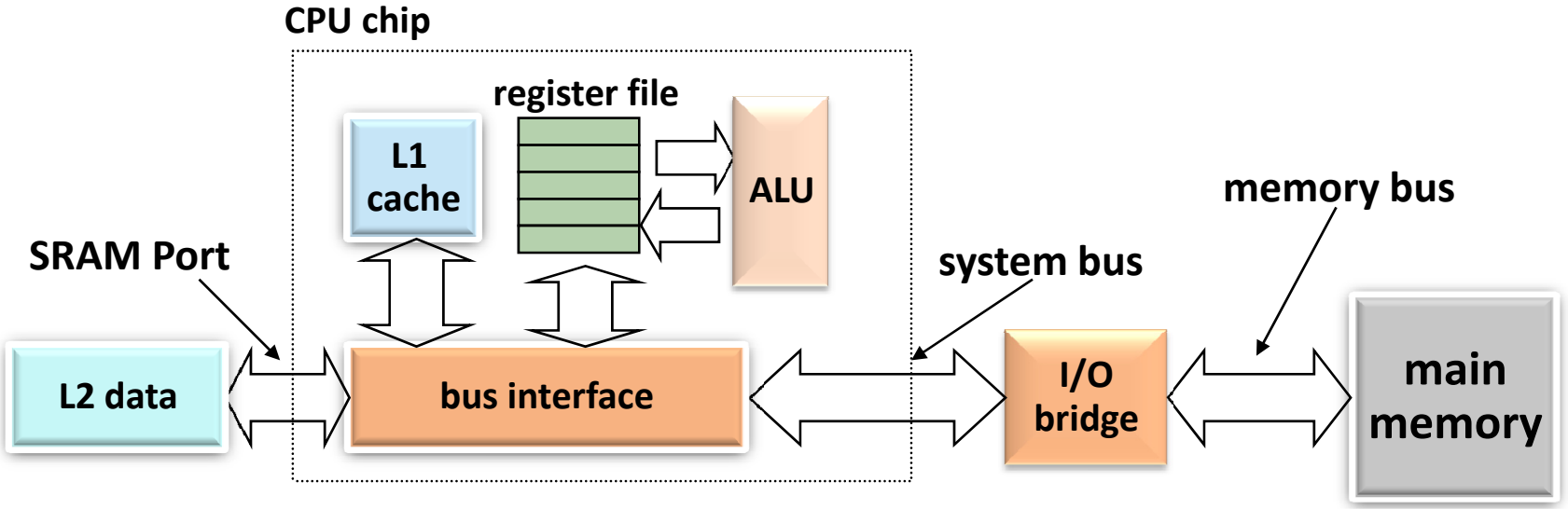
(software/program running on a processor)

$t1 \leftarrow x$
$t2 \leftarrow \mathbf{A} \times t1$
$t2 \leftarrow t2 + \mathbf{B}$
$t2 \leftarrow \mathbf{t2} \times t1$
$y \leftarrow \mathbf{t2} + \mathbf{C}$



The Von Neumann Computer Model

- **Partitioning of the programmable computing engine into components:**
  - Central Processing Unit (CPU): Control Unit (instruction decode , sequencing of operations), Datapath (registers, arithmetic and logic unit, buses).
  - Memory: Instruction and data operand storage.
  - Input/Output (I/O) sub-system: I/O bus, interfaces, devices.
  - The stored program concept: Instructions from an instruction set are fetched from a common memory and executed one at a time

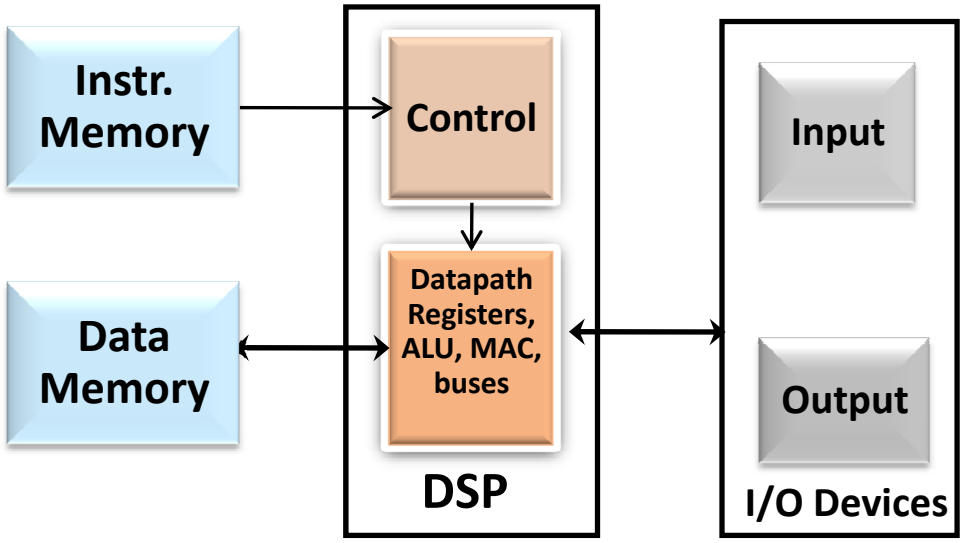


EECC550-Shaaban, RIT

Harvard Computer Model

- Similar to Von Neumann computer model with the following exceptions:
  - Instruction memory and data memory are separate
  - Self modifying code is not possible

- Digital Signal Processors are usually Harvard machines.



CPU µarchitecture Design Steps

Datapath

- 1. Analyze instruction set operations using independent RTN

  ISA => RTN => datapath requirements.
  - This provides the the required datapath components and how they are connected to meet ISA requirements.
- 2. Select required datapath components, connections & establish clock methodology (e.g clock edge-triggered).
- 3. Assemble datapath meeting the requirements.

Control

- 4. Identify and define the function of all control points or signals needed by the datapath.
  - Analyze implementation of each instruction to determine setting of control points that affects its operations and register transfer.
- 5. Design & assemble the control logic.
  - Hard-Wired: Finite-state machine implementation.
  - Micro-programmed.

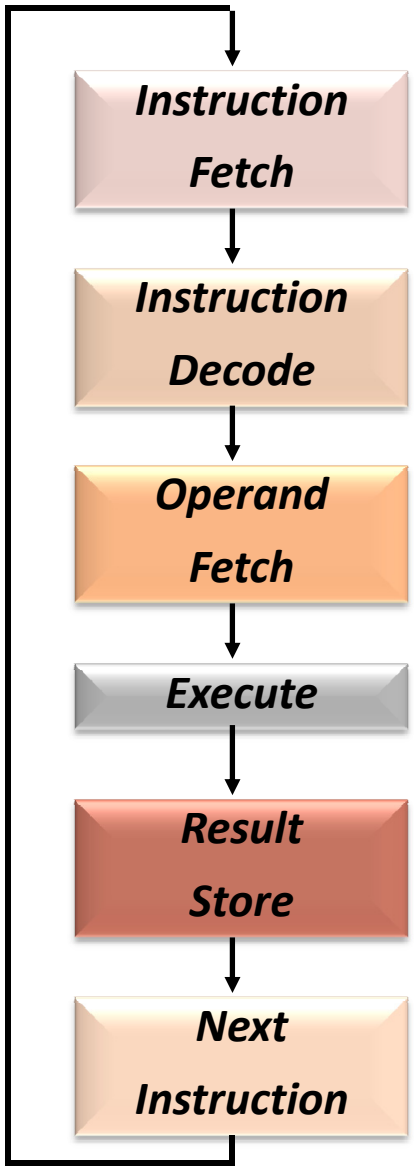
CPU Design Considerations

- **Datapath Design:**
  - Number of pipeline stages
  - Capabilities & performance characteristics of principal Functional Units (FUs): Registers, ALU, Shifters, Logic Units, ...
  - Ways in which these components are interconnected (buses connections, multiplexors, etc.).

- **Control Unit Design:**
  - Logic and means by which such information flow is controlled.
  - Control and coordination of FUs operation to realize the targeted Instruction Set Architecture (ISA) to be implemented.
    - Can either be implemented using a hard coded finite state machine or a micro-coded machine.

Datapath Design

Generic CPU Machine Instruction Processing Steps



Obtain instruction from program storage

Determine required actions and instruction size

Locate and obtain operand data

Compute result value or status

Deposit results in storage for later use

Determine successor or next instruction

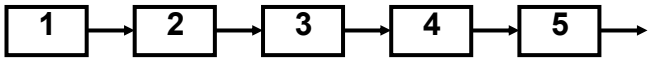
NOTE: These steps can be performed in a single clock cycle. There are numerous issues with single cycle machines.

Drawbacks of Single-Cycle Processor

- **Long cycle time:**
  - All instructions must take as much time as the slowest:
    - Cycle time for load is longer than needed for all other instructions.
  - Real memory is not as well-behaved as idealized memory
    - Cannot always complete data access in one (short) cycle.
- **Impossible to implement complex, variable-length instructions and complex addressing modes in a single cycle.**
  - e.g. indirect memory addressing.
- **Very high hardware resource requirements**
  - Any hardware functional unit cannot be used more than once in a single cycle (e.g. ALUs).
- **Cannot pipeline (overlap) the processing of one instruction with the previous instructions.**

Instruction Pipelining

- Instruction pipelining is a CPU implementation technique where multiple operations on a number of instructions are overlapped.
  - Instruction pipelining exploits Instruction-Level Parallelism (ILP)
- An instruction execution pipeline involves a number of steps, where each step completes a part of an instruction. Each step is called a pipeline stage or a pipeline segment.
- The stages or steps are connected in a linear fashion: one stage to the next to form the pipeline -- instructions enter at one end and progress through the stages and exit at the other end.



- The time to move an instruction one step down the pipeline is is equal to the machine cycle and is determined by the stage with the longest processing delay.

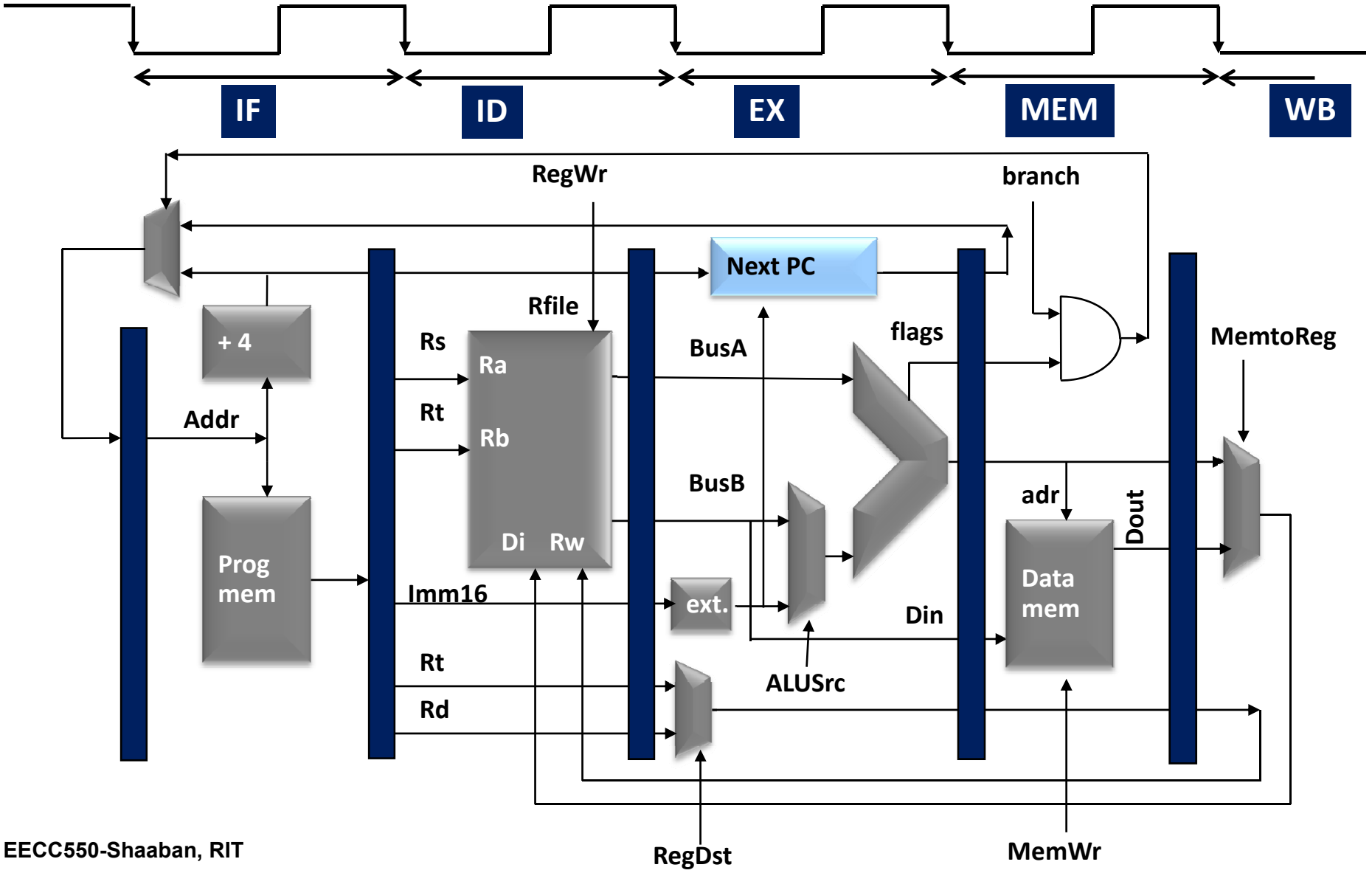
Instruction Pipelining (cont)

- Pipelining increases the CPU instruction throughput: The number of instructions completed per cycle.
  - Under ideal conditions (no stall cycles), instruction throughput is one instruction per machine cycle, or ideal CPI = 1

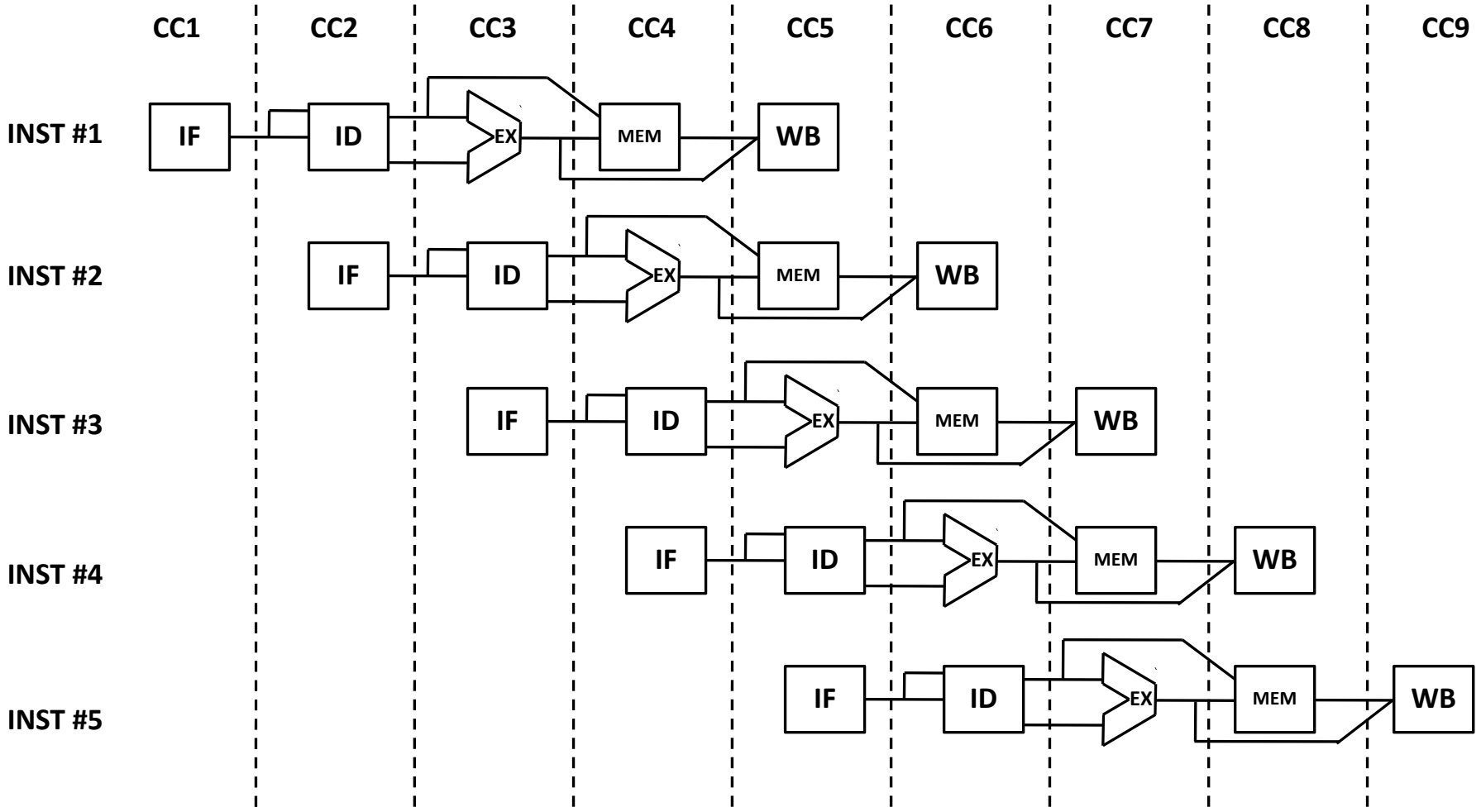
- Pipelining does not reduce the execution time of an individual instruction: The time needed to complete all processing steps of an instruction (also called instruction completion latency).
  - Minimum instruction latency = n cycles, where n is the number of pipeline stages

- What is the difference between Latency and Throughput?
- What is the Throughput of a pipelined machine with a CPI=1?
- What is the Latency of a 5 stage pipelined machine with a IPC=2?

Typical 5 Stage Pipelined Integer Datapath



Temporal view of a Pipelined Datapath



Pipelining Performance Example

Example: For an unpipelined CPU:

Clock cycle = 1ns, 4 cycles for ALU operations and branches and 5 cycles for memory operations with instruction frequencies of 40%, 20% and 40%, respectively. The average instruction execution time is:

Clock cycle x Average CPI = 1 ns x ((40% + 20%) x 4 + 40%x 5) = 1 ns x 4.4 = 4.4 ns

If pipelining adds 0.2 ns to the machine clock cycle (for the flip-flops) and a five stage pipeline is used then the average instruction execution time is:

1 ns + 0.2 ns = 1.2 ns

The speedup in instruction execution from pipelining is:

$$\text{Speedup from pipelining} = \frac{\text{Instruction time unpipelined}}{\text{Instruction time pipelined}} = \frac{4.4\text{ns}}{1.2\text{ns}} \quad 3.7\text{X}$$

Pipeline Hazards

- **Hazards are situations in pipelining which prevent the next instruction in the instruction stream from executing during the designated clock cycle possibly resulting in one or more stall (or wait) cycles.**
- **Hazards reduce the ideal speedup (increase CPI > 1) gained from pipelining and are classified into three classes:**
  - **Structural hazards: Arise from hardware resource conflicts when the available hardware cannot support all possible combinations of instructions.**
  -
  - **Data hazards: Arise when an instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline**
  - **Control hazards: Arise from the pipelining of conditional branches and other instructions that change the PC**

Performance of Pipelines with Stalls

- Hazard conditions in pipelines may make it necessary to stall the pipeline by a number of cycles degrading performance from the ideal pipelined CPU CPI of 1.

CPI pipelined = Ideal CPI + Pipeline stall clock cycles per instruction
= 1 + Pipeline stall clock cycles per instruction

- If pipelining overhead is ignored and we assume that the stages are perfectly balanced then speedup from pipelining is given by:

Speedup = CPI unpipelined / CPI pipelined
= CPI unpipelined / (1 + Pipeline stall cycles per instruction)

- When all instructions in the multicycle CPU take the same number of cycles equal to the number of pipeline stages then:

Speedup = Pipeline depth / (1 + Pipeline stall cycles per instruction)

Structural (or Hardware) Hazards

- In pipelined machines overlapped instruction execution requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline.

- If a resource conflict arises due to a hardware resource being required by more than one instruction in a single cycle, and one or more such instructions cannot be accommodated, then a structural hazard has occurred, for example:
  - when a pipelined machine has a shared single-memory pipeline stage for data and instructions.
  - stall the pipeline for one cycle for memory data access
  - Solution: Use a Harvard architecture machine.

A Structural Hazard Example

- Given that data references (load/store) are 40% for a specific instruction mix or program, and that the ideal pipelined CPI ignoring hazards is equal to 1.

- A machine with a data memory access structural hazards requires a single stall cycle for data references and has a clock rate 1.05 times higher than the ideal machine. Ignoring other performance losses for this machine:

Average instruction time = CPI X Clock cycle time

Average instruction time = (1 + 0.4 x 1) x $\frac{\text{Clock cycle ideal}}{1.05}$

= 1.33 X Clock cycle time ideal

i.e. CPU without structural hazard is 1.33 times faster

Data Hazards

- Data hazards occur when the pipeline changes the order of read/write accesses to instruction operands in such a way that the resulting access order differs from the original sequential instruction operand access order of the unpipelined machine resulting in incorrect execution.
- Data hazards may require one or more instructions to be stalled to ensure correct execution.

Example:

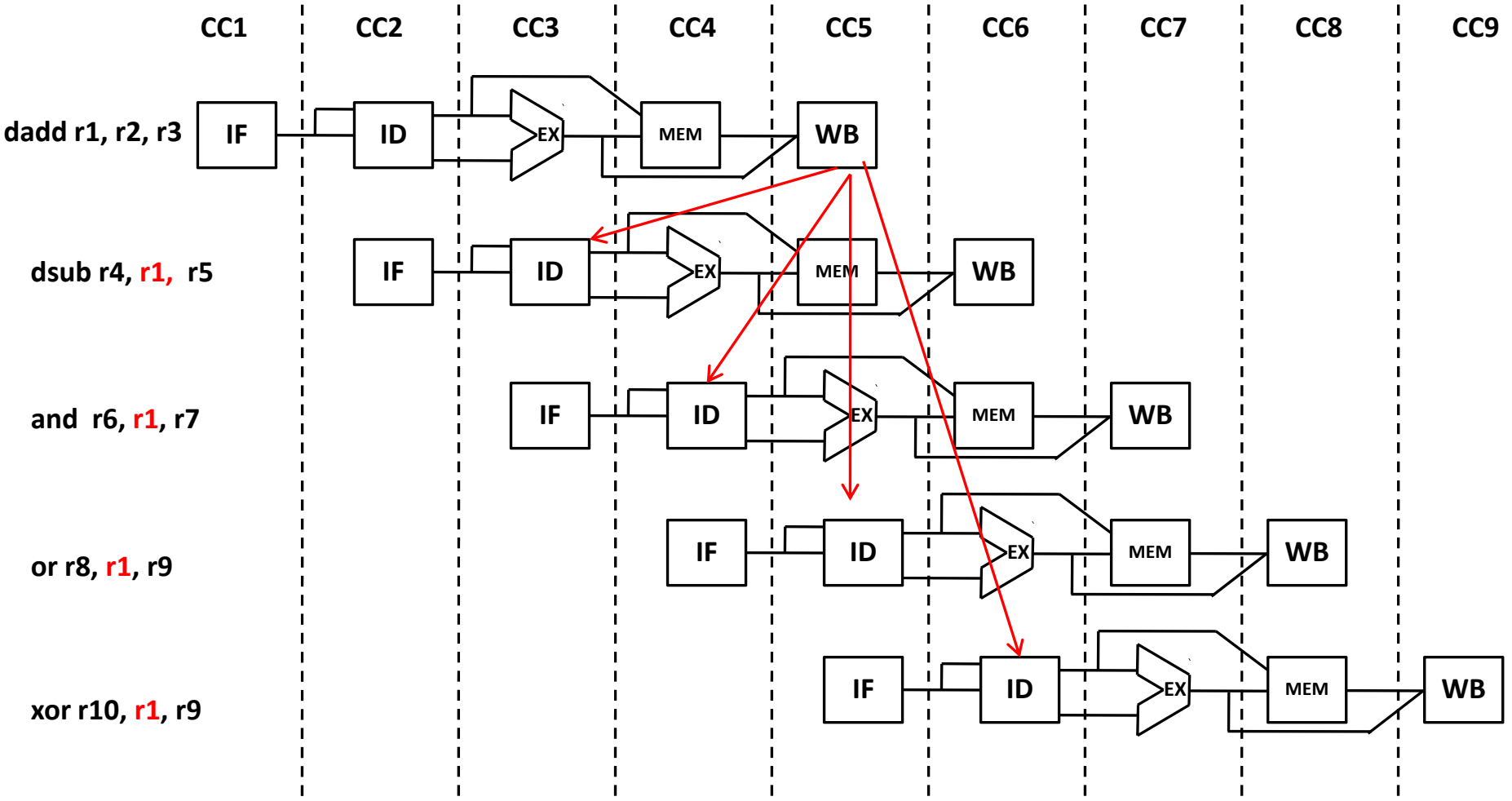
```
DADD   R1, R2, R3
DSUB   R4, R1, R5
AND    R6, R1, R7
OR     R8, R1, R9
XOR    R10, R1, R11
```

Register R1 is used by multiple instruction Instructions that have no dependencies among them are said to be parallel or independent

A high degree of Instruction-Level Parallelism (ILP) is present in a given code sequence if it has a large number of parallel instructions

All the instructions after DADD use the result of the DADD instruction DSUB, AND instructions need to be stalled for correct execution.

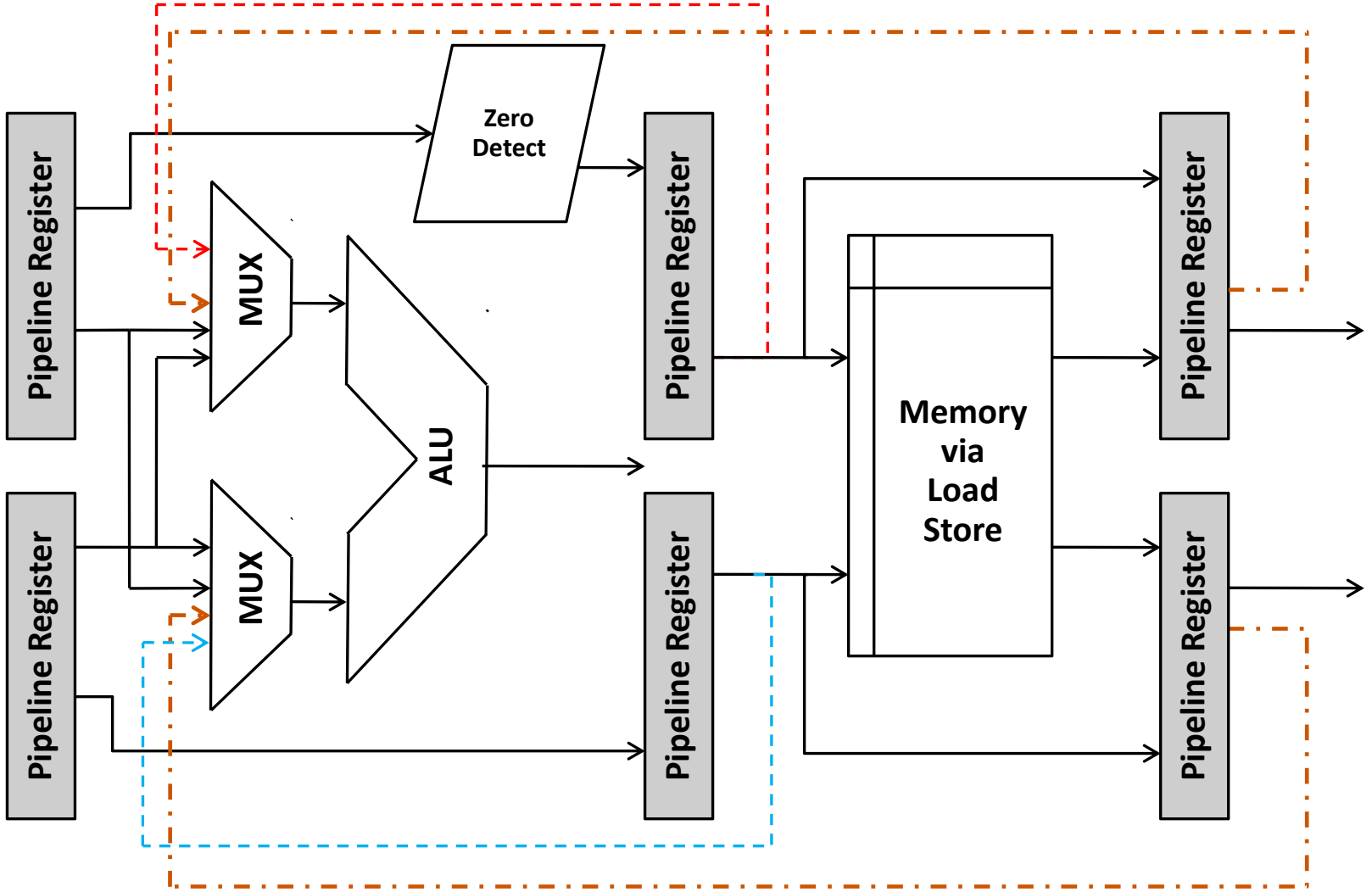
Data Hazard Example



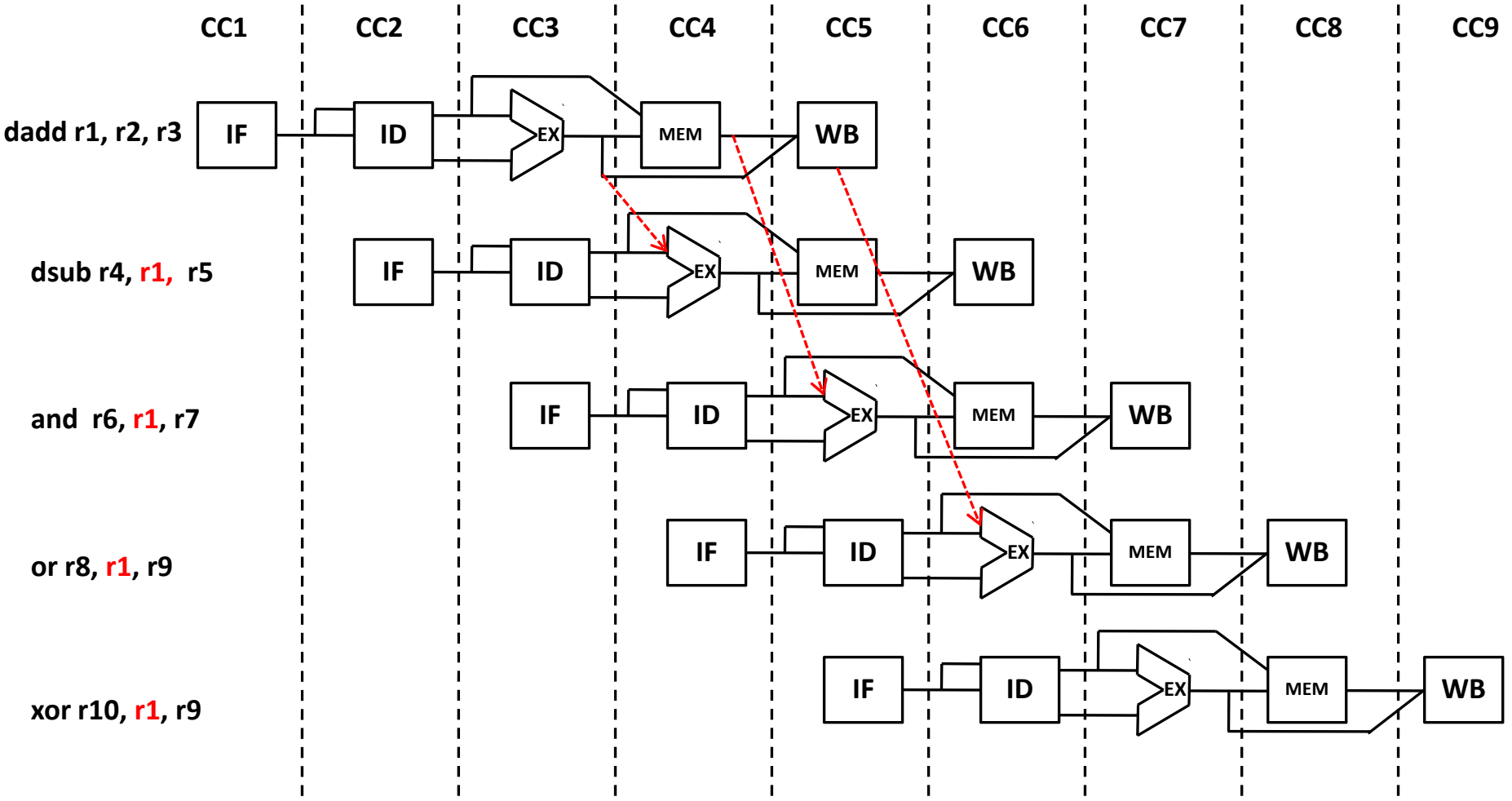
Minimizing Data Hazard Stalls by Forwarding

- Data forwarding is a hardware-based technique (also called register bypassing or short-circuiting) used to eliminate or minimize data hazard stalls.
- Using forwarding hardware, the result of an instruction is copied directly from where it is produced (ALU, memory read port etc.), to where subsequent instructions need it (ALU input register, memory write port etc.)
- For example, in the MIPS integer pipeline with forwarding:
  - The ALU result from the EX/MEM register may be forwarded or fed back to the ALU input latches as needed instead of the register operand value read in the ID stage.
  - Similarly, the Data Memory Unit result from the MEM/WB register may be fed back to the ALU input latches as needed .
  - If the forwarding hardware detects that a previous ALU operation is to write the register corresponding to a source for the current ALU operation, control logic selects the forwarded result as the ALU input rather than the value read from the register file.

Pipeline with Data Forwarding

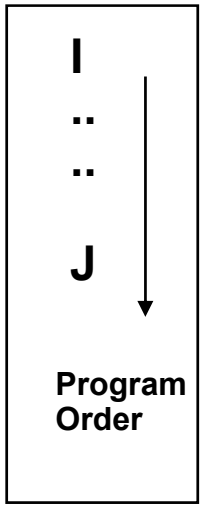


Data Forwarding Example



Data Hazard Classification

Given two instructions *I*, *J*, with *I* occurring before *J* in an instruction stream (program execution order):



RAW (read after write): ***A true data dependence violation***

J tried to read a source before *I* writes to it, so *J* incorrectly gets the old value.

WAW (write after write): ***A name dependence violation***

J tries to write an operand before it is written by *I*

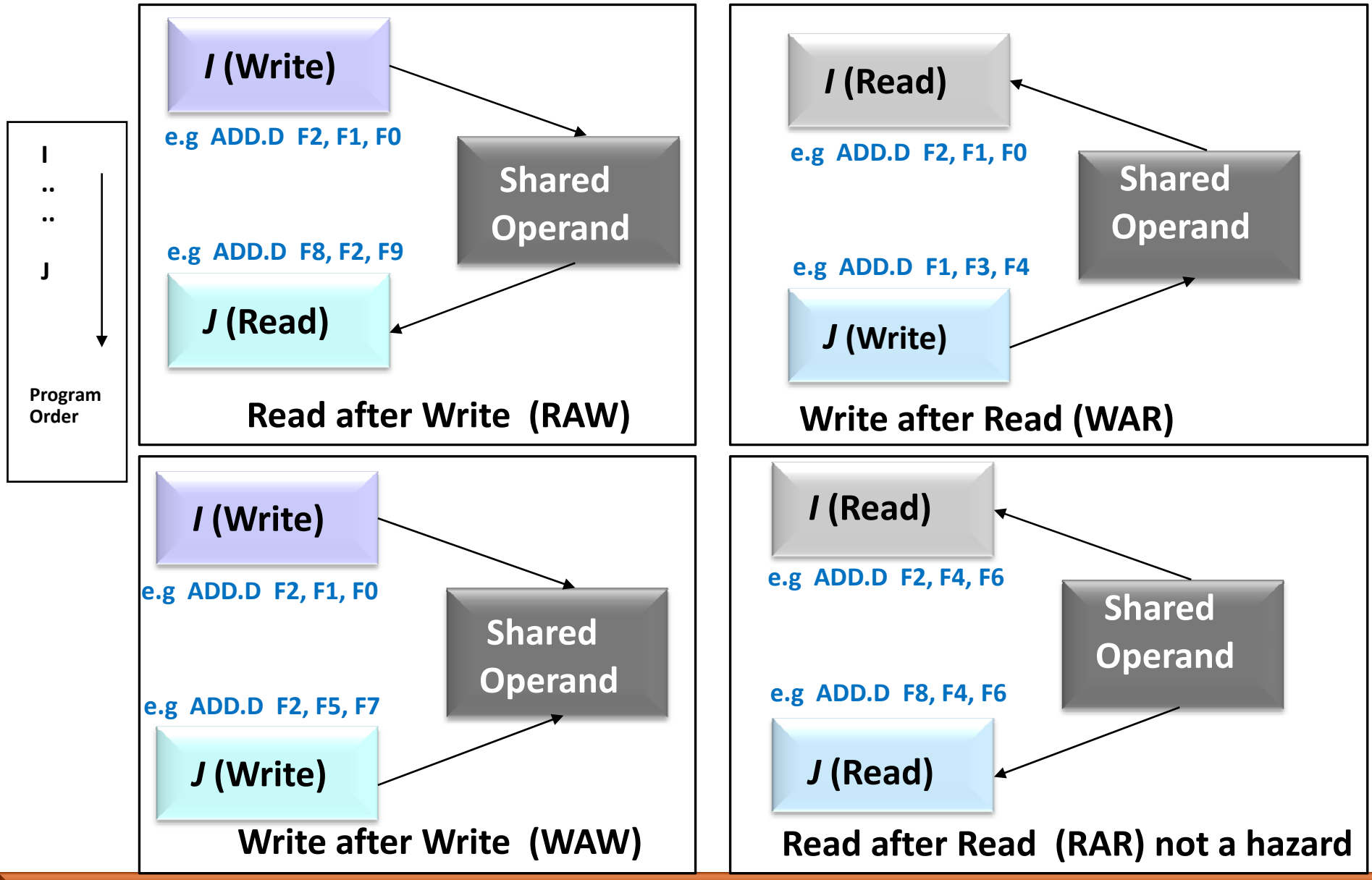
The writes end up being performed in the wrong order.

WAR (write after read): ***A name dependence violation***

J tries to write to a destination before it is read by *I*, so *I* incorrectly gets the new value.

RAR (read after read): Not a hazard.

Data Hazard Classification



More Problems with Pipelining: Exceptions and Interrupts

- **Exception:** An unusual event happens to an instruction during its execution
  - Examples: divide by zero, undefined opcode

- **Interrupt:** Hardware signal to switch the processor to a new instruction stream
  - Example: a sound card interrupts when it needs more audio output samples (an audio "click" happens if it is left waiting)

- **Problem:** It must appear that the exception or interrupt happens between 2 instructions (I_i and I_{i+1})
  - The effect of all instructions up to and including I_i is totally complete
  - No effect of any instruction after I_i can take place

- The interrupt (exception) handler either aborts program or restarts at instruction I_{i+1}

Patterson UCB

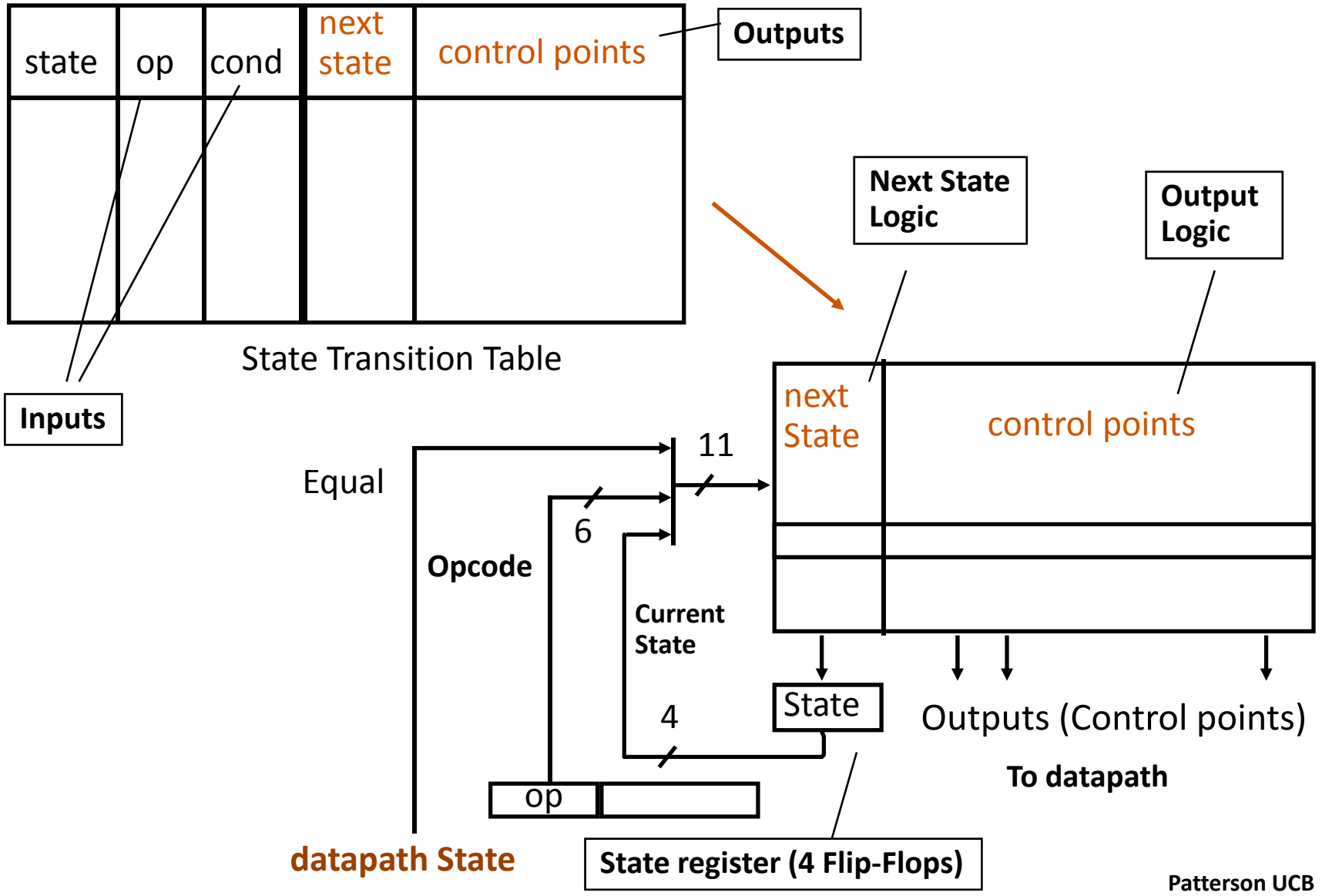
Control Unit Design

Control Unit Design

- There are three basic approaches to building control units for a computing element:
  - Finite State Machine
    - Generally a Moore machine
  - Micro-programmed
    - Implemented as ROMs or Writeable Control Store (WCS)
      - MicroROM vs. NanoROM
    - Horizontal vs. Vertical microcode
  - Hardwired

Patterson UCB

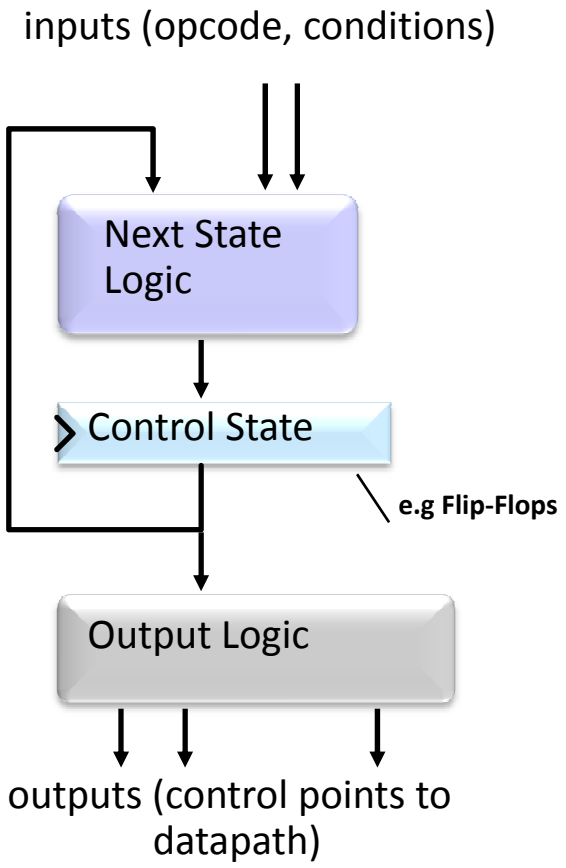
Traditional FSM Controller

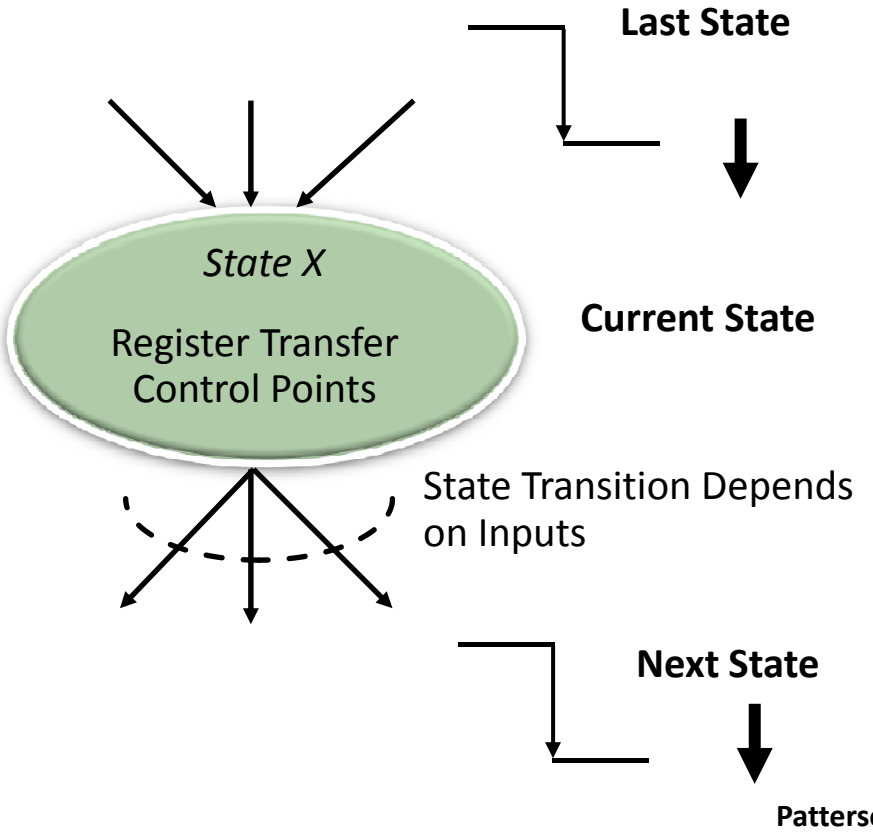


Patterson UCB

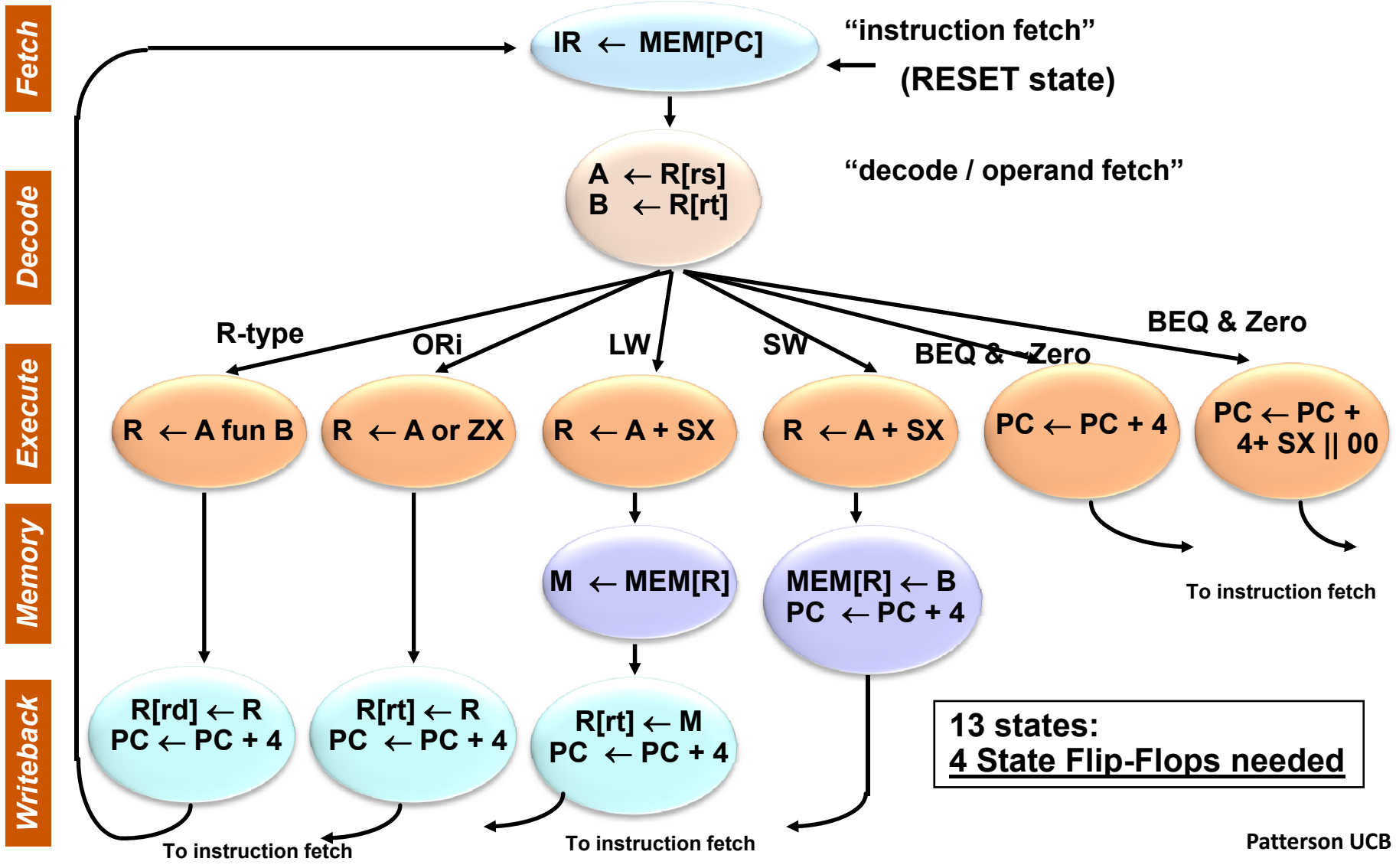
Finite State Machine (FSM) Control Model

- State specifies control points (outputs) for Register Transfer.
- Control points (outputs) are assumed to depend only on the current state and not inputs (i.e. Moore finite state machine)
- Transfer (register/memory writes) and state transition occur upon exiting the state on the falling edge of the clock.

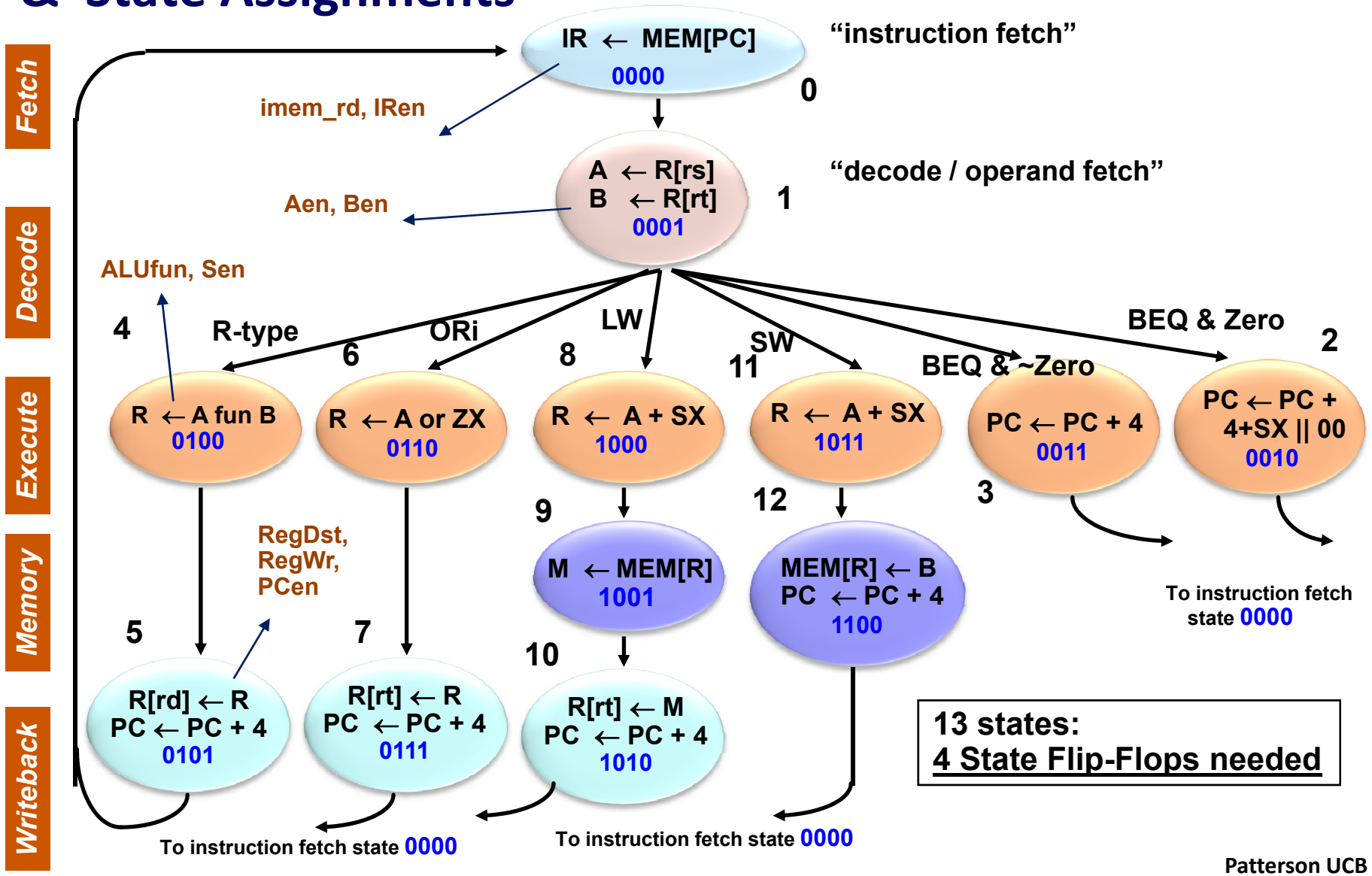




Control Specification For Multi-cycle CPU
Finite State Machine (FSM) - State Transition Diagram



Mapping RTNs To Control Points Examples & State Assignments

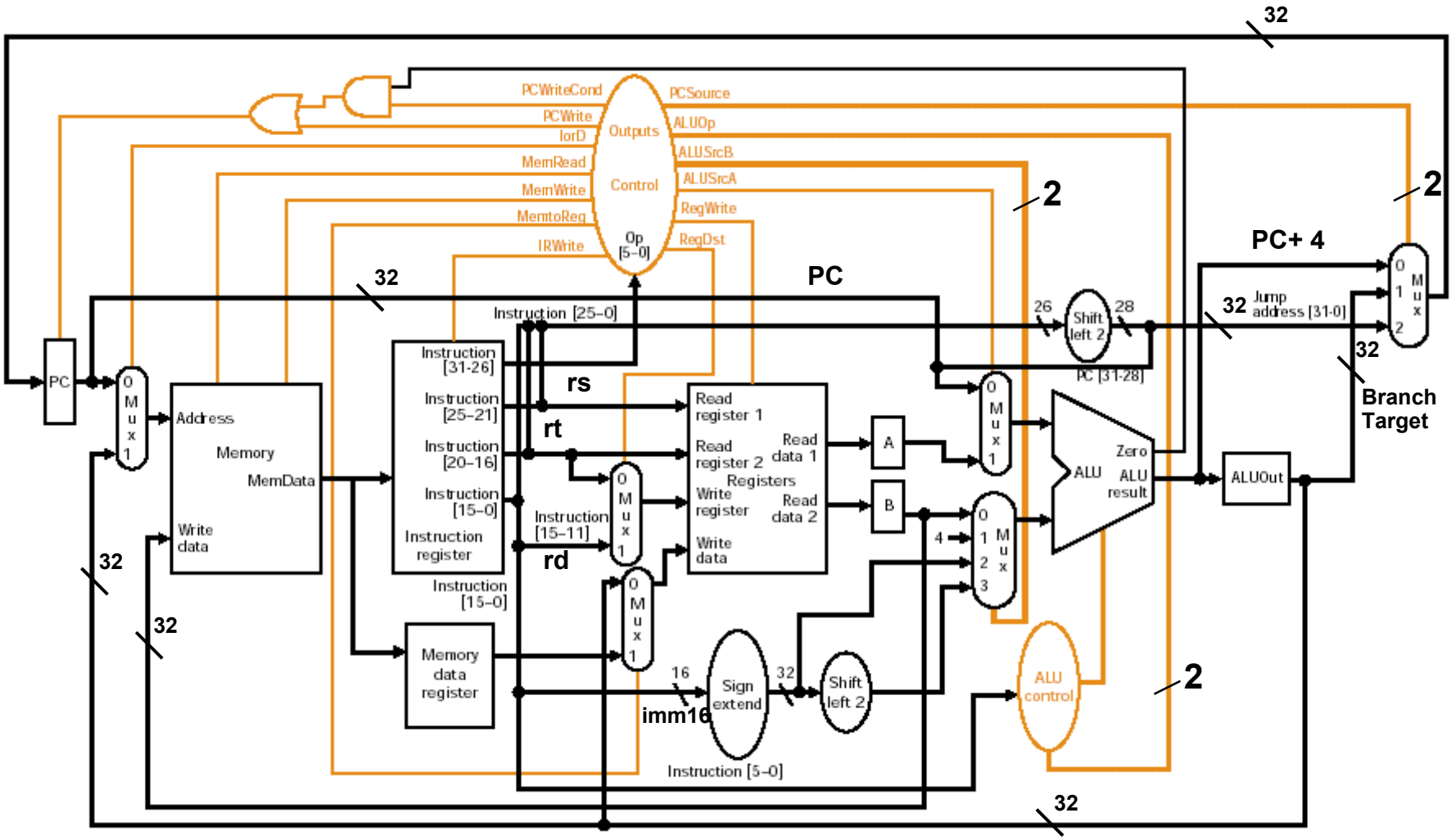


Detailed Control Specification - State Transition Table

	Current State	Op field	Z	Next	IR	PC en	PC sel	Ops A	Ops B	Exec Ex	Exec Sr	Exec ALU	Exec S	Mem R	Mem W	Mem M	Write-Back M-R	Write-Back Wr	Write-Back Dst
IF	0000	??????	?	0001	1														
	0001	BEQ	0	0011				1	1										
	0001	BEQ	1	0010				1	1										
ID	0001	R-type	x	0100				1	1										
	0001	orI	x	0110				1	1										
	0001	LW	x	1000				1	1										
	0001	SW	x	1011				1	1										
BEQ	0010	xxxxxx	x	0000		1	1	Can be combined into one state											
	0011	xxxxxx	x	0000		1	0												
R	0100	xxxxxx	x	0101						0	1	fun	1						
	0101	xxxxxx	x	0000		1	0										0	1	1
ORI	0110	xxxxxx	x	0111						0	0	or	1						
	0111	xxxxxx	x	0000		1	0										0	1	0
LW	1000	xxxxxx	x	1001						1	0	add	1						
	1001	xxxxxx	x	1010										1	0	1			
	1010	xxxxxx	x	0000		1	0										1	1	0
SW	1011	xxxxxx	x	1100						1	0	add	1						
	1100	xxxxxx	x	0000		1	0							0	1				

Patterson UCB

Multiple Cycle Datapath With Control Block



Patterson UCB

1-HOT Control Signals

Signal Name	Effect when = 0	Effect when = 1
RegDst	The register destination number for the write register comes from the rt field (instruction bits 20:16).	The register destination number for the write register comes from the rd field (instruction bits 15:11).
RegWrite	None	The register on the write register input is written with the value on the Write data input.
ALUSrcA	The first ALU operand is the PC	The First operand is register A (i.e R[rs])
MemRead	None	Content of memory specified by the address input are put on the memory data output.
MemWrite	None	Memory contents specified by the address input is replaced by the value on the Write data input.
MemtoReg	The value fed to the register write data input comes from ALUOut register.	The value fed to the register write data input comes from data memory register (MDR).
IorD	The PC is used to supply the address to the memory unit.	The ALUOut register is used to supply the address to the memory unit.
IRWrite	None	The output of the memory is written into Instruction Register (IR)
PCWrite	None	The PC is written; the source is controlled by PCSource
PCWriteCond	None	The PC is written if the Zero output of the ALU is also active.

Patterson UCB

Encoded Control Signals

Signal Name	Value	Action
ALUOp	00	The ALU performs an add operation
	01	The funct field of the instruction determines the ALU operation (R-Type)
	10	The ALU performs a subtract operation
ALUSrcB	00	The second input of the ALU comes from register B
	01	The second input of the ALU is the constant 4
	10	The second input of the ALU is the sign-extended 16-bit immediate (imm16) field of the instruction in IR
	11	The second input of the ALU is is the sign-extended 16-bit immediate field of IR shifted left 2 bits
PCSource	00	Output of the ALU (PC+4) is sent to the PC for writing
	01	The content of ALUOut (the branch target address) is sent to the PC for writing
	10	The jump target address (IR[25:0] shifted left 2 bits and concatenated with PC+4[31:28] is sent to the PC for writing

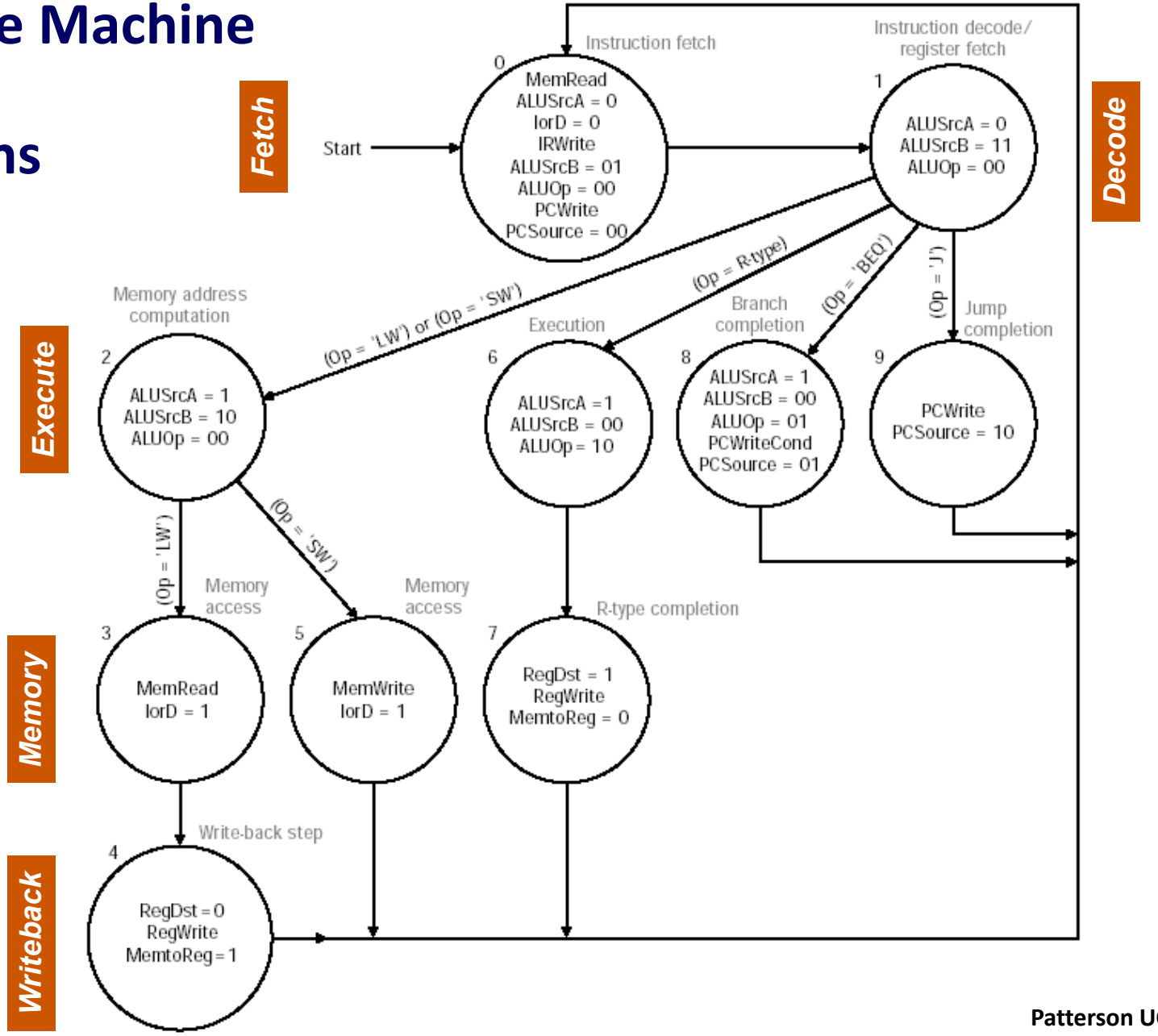
Patterson UCB

Operations (Dependent RTN) for Each Cycle

	R-Type	Load	Store	Branch	Jump
Instruction Fetch	IR ¬ Mem[PC] PC ¬ PC + 4	IR ¬ Mem[PC] PC ¬ PC + 4	IR ¬ Mem[PC] PC ¬ PC + 4	IR ¬ Mem[PC] PC ¬ PC + 4	IR ¬ Mem[PC] PC ¬ PC + 4
Instruction Decode	A ¬ R[rs] B ¬ R[rt] ALUout ¬ PC + (SignExt(imm16) x4)	A ¬ R[rs] B ¬ R[rt] ALUout ¬ PC + (SignExt(imm16) x4)	A ¬ R[rs] B ¬ R[rt] ALUout ¬ PC + (SignExt(imm16) x4)	A ¬ R[rs] B ¬ R[rt] ALUout ¬ PC + (SignExt(imm16) x4)	A ¬ R[rs] B ¬ R[rt] ALUout ¬ PC + (SignExt(imm16) x4)
Execution	ALUout ¬ A funct B	ALUout ¬ A + SignEx(Imm16)	ALUout ¬ A + SignEx(Imm16)	Zero ¬ A - B Zero: PC ¬ ALUout	PC ¬ Jump Address
Memory		MDR ¬ Mem[ALUout]	Mem[ALUout] ¬ B		
Write Back	R[rd] ¬ ALUout	R[rt] ¬ MDR			

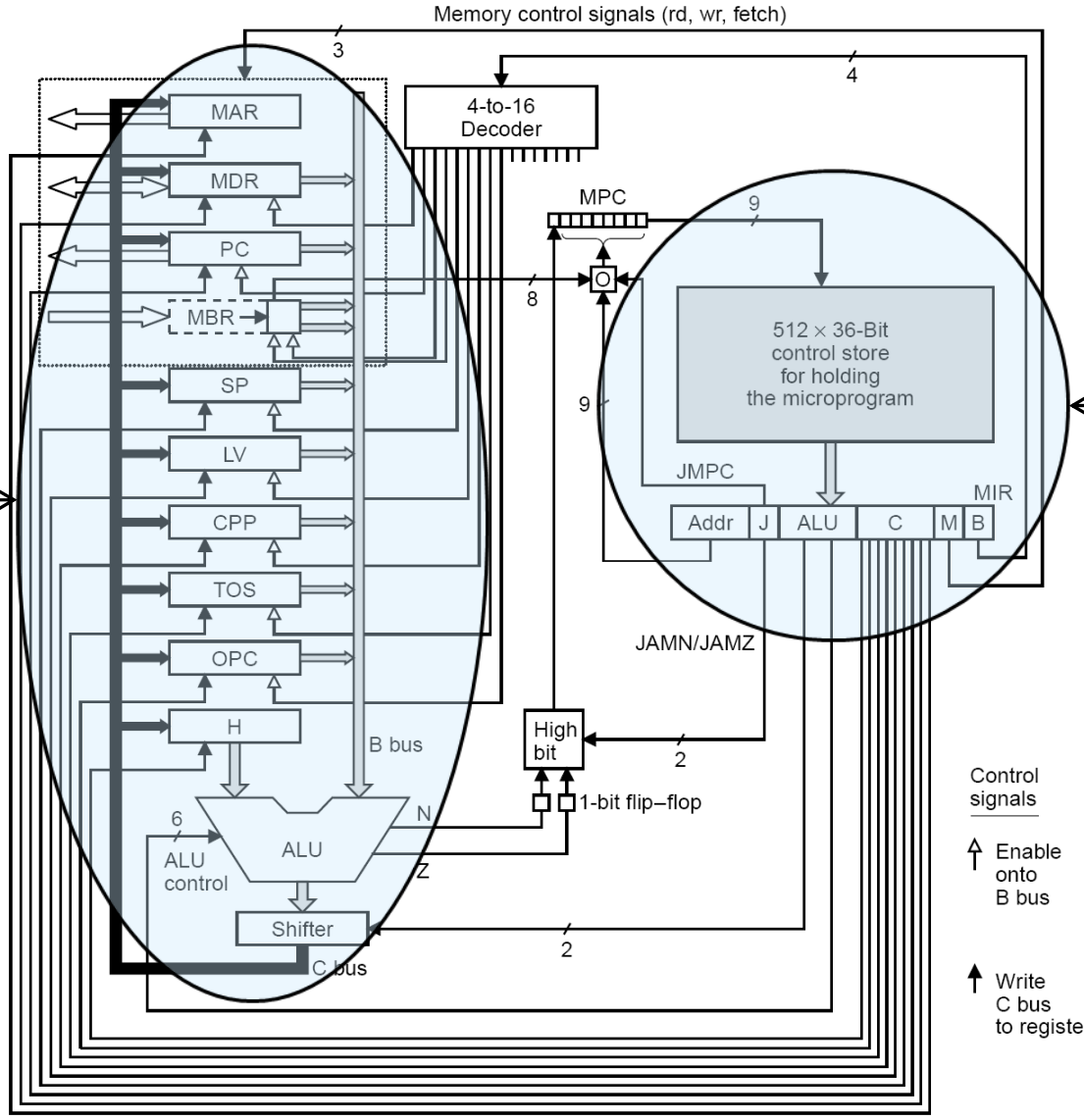
Patterson UCB

Finite State Machine w/control annotations



Patterson UCB

Micro-programmable Datapath Example

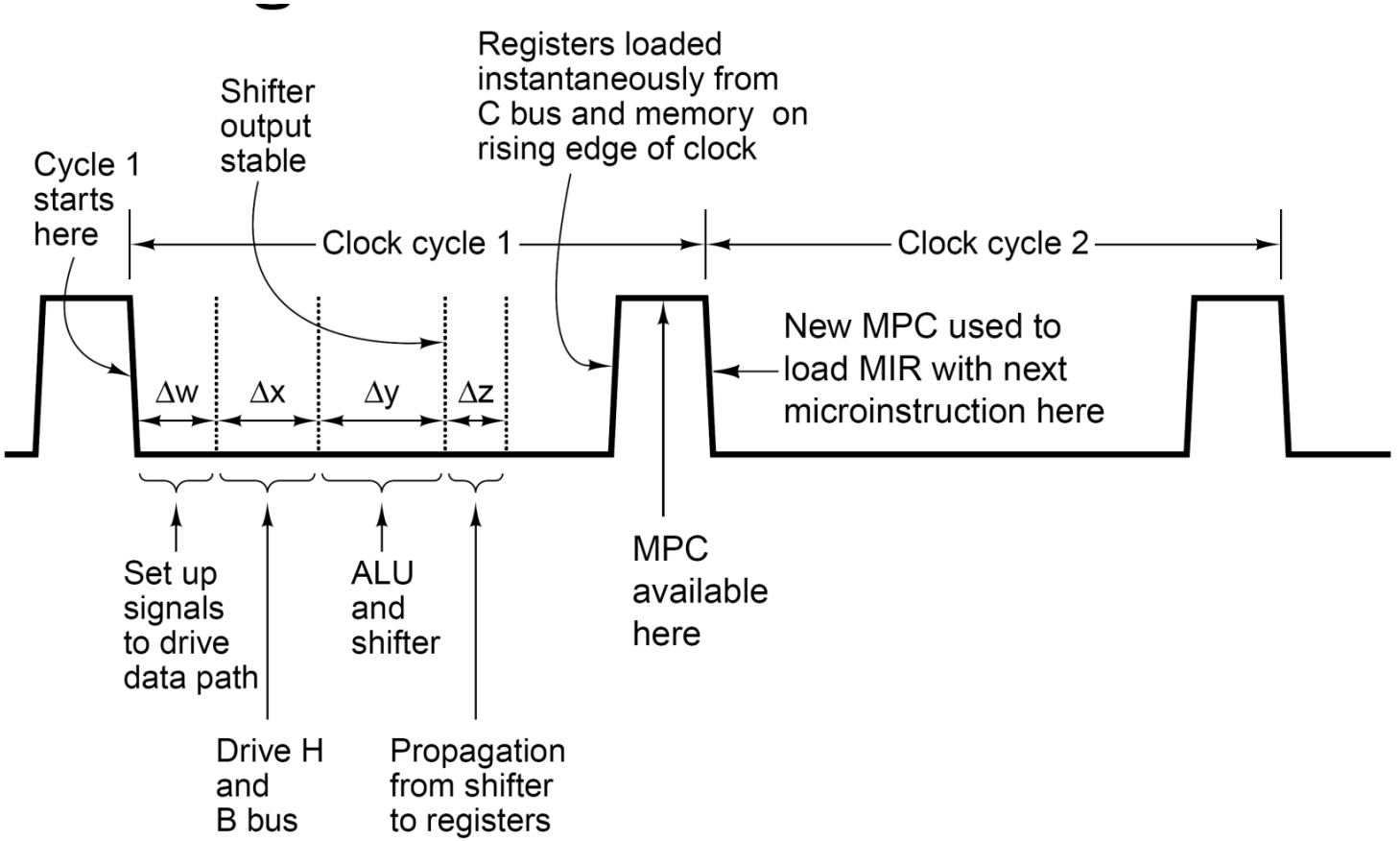


DATAPATH

CONTROL STORE PROGRAM

Tannenbaum

Datapath Timing



Tannenbaum

Data Path Timing

- **Short pulse is produced at start of each clock cycle.**
  - **Various sub-cycles:**
    - **Control signals are set up (w).**
    - **Registers are loaded onto the B bus (x).**
    - **The ALU and shifter operate (y).**
    - **The results propagate along the C bus to the registers (z).**

- **Sub-cycles are implicitly determined by circuit delays:**
  - **Bits need time to become stable.**
  - **ALU has some signal propagation time.**
    - **Until w + x, ALU input is garbage.**
    - **Until w + x + y, ALU output is garbage.**

- **Operation depends on rigid timing of all elements.**

Memory operations

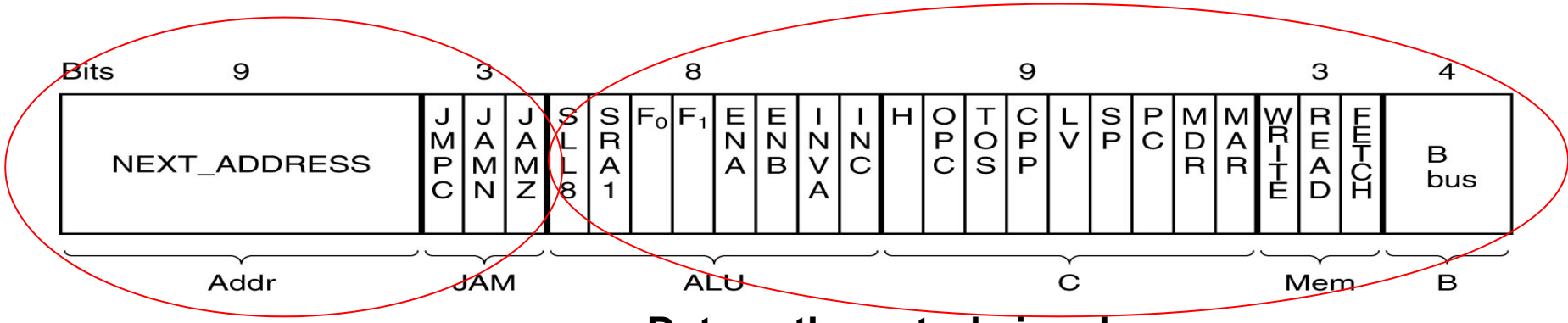
- **32-bit word-addressable memory port.**
  - Controlled by MAR (Memory Address Register) and MDR (Memory Data Register).
  - MAR contains address of word (word 0, word 1, . . .) to be read into MDR.
  - Reading and writing of ISA-level data words.
- **8-bit byte-addressable memory port.**
  - Controlled by PC (Program Counter).
  - PC contains addresses of byte (byte 0, byte 1, . . .) to be read into low-order 8 bits of MBR.
  - Sign extension of MBR (signed, unsigned) is determined by two control lines.
- **Each register is driven by one or two control signals.**
  - Control signal that enables register's output onto B bus (open arrow).
  - Control signal that loads the register from the C bus (solid arrow).
  - Reading and writing of ISA-level program (byte stream).

Datapath Control Signals

- **29 control signals determine data path.**
  - 9 signals to control writing data from C bus into registers.
  - 9 signals to control enabling registers onto the B bus for ALU input.
  - 8 signals to control ALU and shifter operation.
  - 2 signals to indicate memory read/write via MAR/MDR.
  - 1 signal to indicate memory fetch via PC/MBR.
- **Signal values specify operations for one cycle of data path.**
  - Put values from registers to C bus, propagate signals through ALU and shifter on C bus, write results into appropriate register(s).
- **Memory read data signal is asserted in cycle k:**
  - Memory operation is started at end of cycle k (after MAR has been loaded).
  - Memory data are available in MDR at the very end of cycle k + 1.
  - Memory data can be used in cycle k + 2.

Microinstructions

- The ADDR bits contain the address of the potential next microinstruction
- JAM determines how the next microinstruction is selected
- The ALU group control the ALU and the shifter
- The C bits cause individual registers to load the ALU output from the C bus
- The M bit controls memory operation
- The 4 B bits drive the decoder that determines what goes onto the B bus



Microprogram control signals

Data path control signals

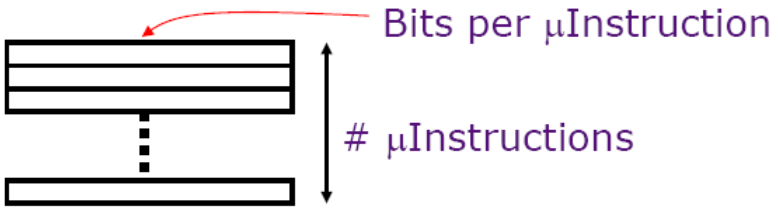
B bus registers

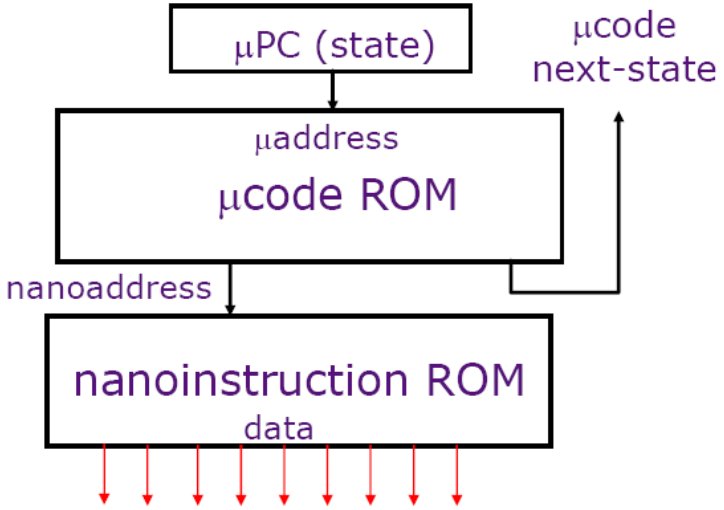
0 = MDR	5 = LV
1 = PC	6 = CPP
2 = MBR	7 = TOS
3 = MBRU	8 = OPC
4 = SP	9-15 none

Tannenbaum

Horizontal vs. Vertical Microcode

- **Horizontal µ-code has wider micro-instructions**
  - Multiple parallel operations per micro-instruction
  - Fewer steps per macroinstruction
  - Sparser encoding ⇒ more bits
- **Vertical µ-code has narrower micro-instructions**
  - Typically a single datapath operation per micro-instruction
    - Separate micro-instruction for branches
  - More steps to per macroinstruction–More compact ⇒ less bits
- **Nanocoding**
  - Tries to combine best of horizontal and vertical µ-code



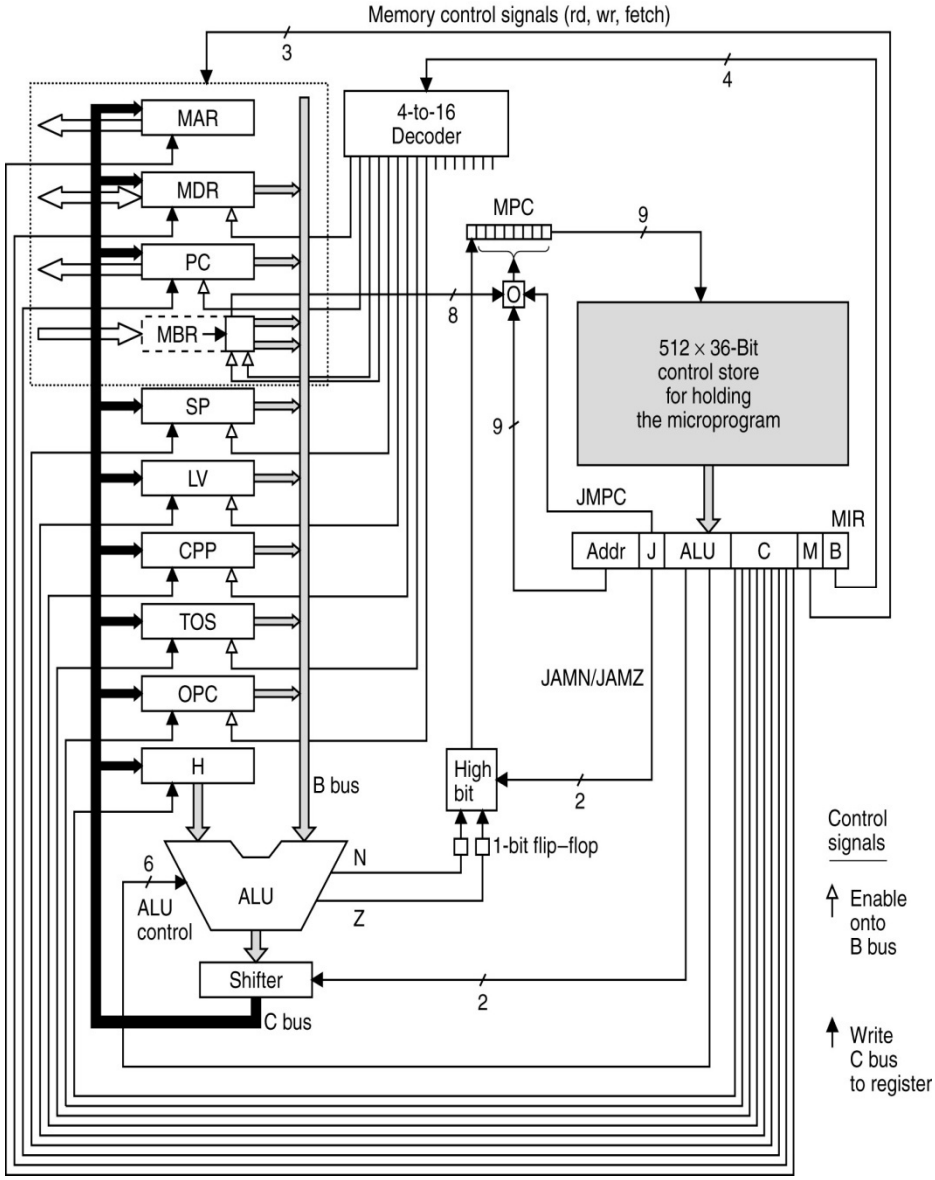


Tannenbaum

Micro-instruction control flow

Sequencer steps through microinstructions.
- Determines state of every control signal.
- Determines address of microinstruction to be executed next.

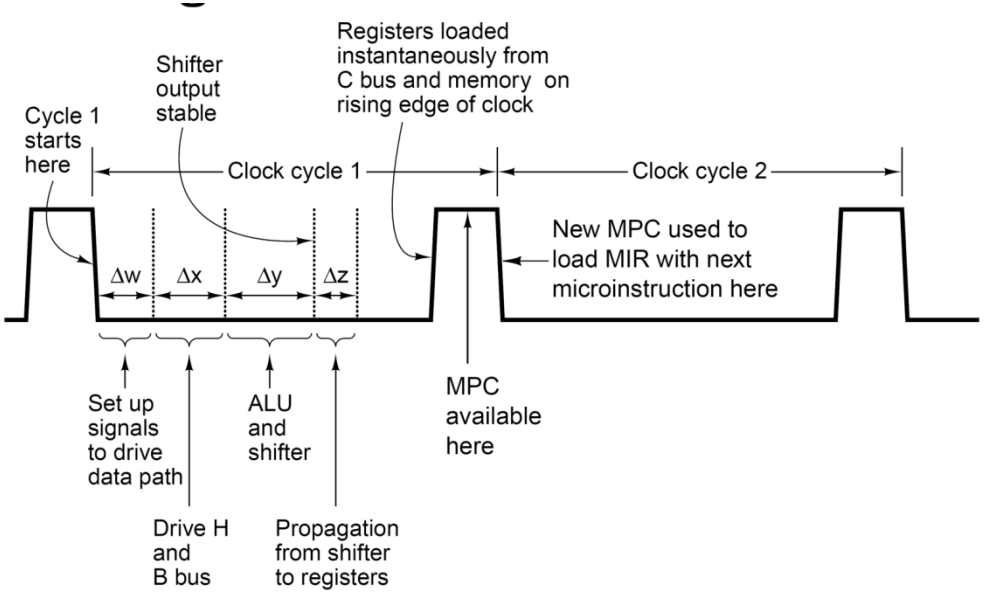
- **Control store holds complete microprogram.**
  - Like program memory, but microinstructions instead of ISA instructions.
  - 512 words containing 36-bit microinstructions.
  - Each microinstruction determines next microinstruction to be executed.
- **MPC (Micro-Program Counter), MIR (Micro-Instruction Counter)**
  - MPC: Address of next microinstruction to be fetched from memory.
  - MIR: Current microinstruction whose bits drive control signals of data path.



Tannenbaum

Micro-instruction control flow (cont)

- MIR is loaded from the word in control store pointed to by MPC.
  - By Δw, MIR is loaded.
- Control signals propagate from MIR into the data path.
  - One register is put onto the B bus.
  - ALU is told which operation to perform.
  - By Δw + Δx, ALU inputs are stable.
- ALU and shifter execute.
  - By Δ w + Δ x + Δ y, ALU and shifter output are stable.
- Output is written into registers.
  - By Δ w + Δ x + Δ y + Δ z, shifter output has reached registers and N and Z flip-ops.



After sub-cycle 4, MPC for next microinstruction is determined.

Tannenbaum