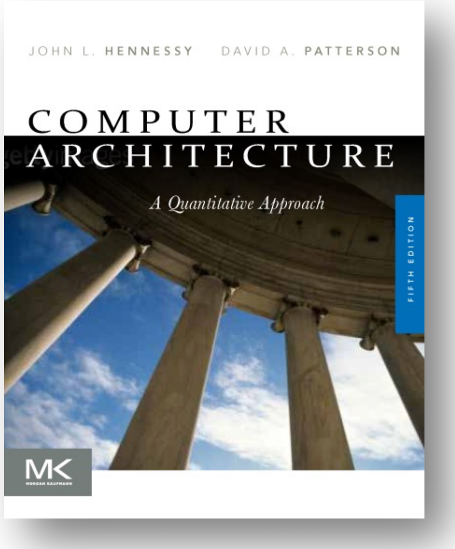
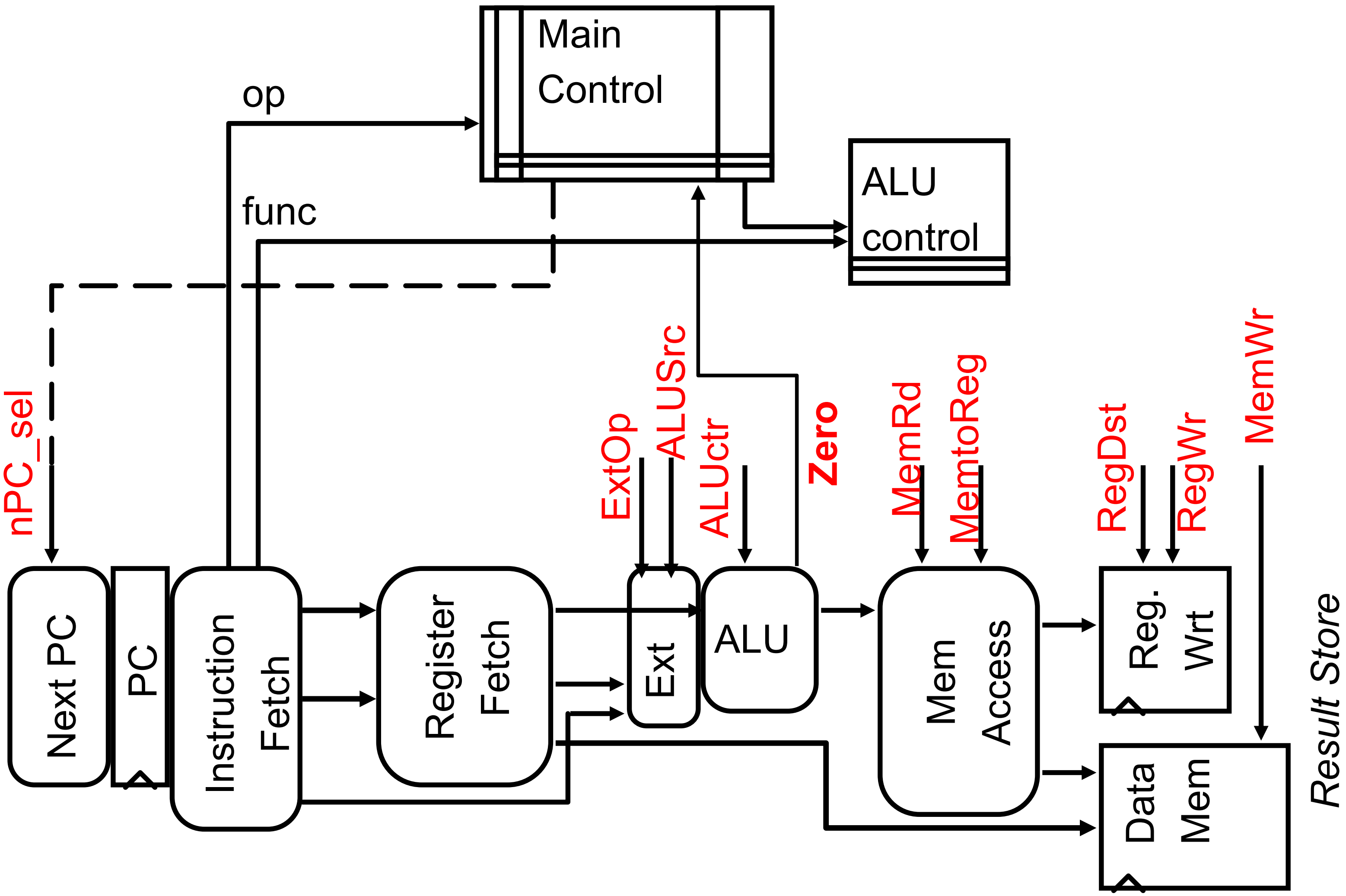


ESE 545 Computer Architecture

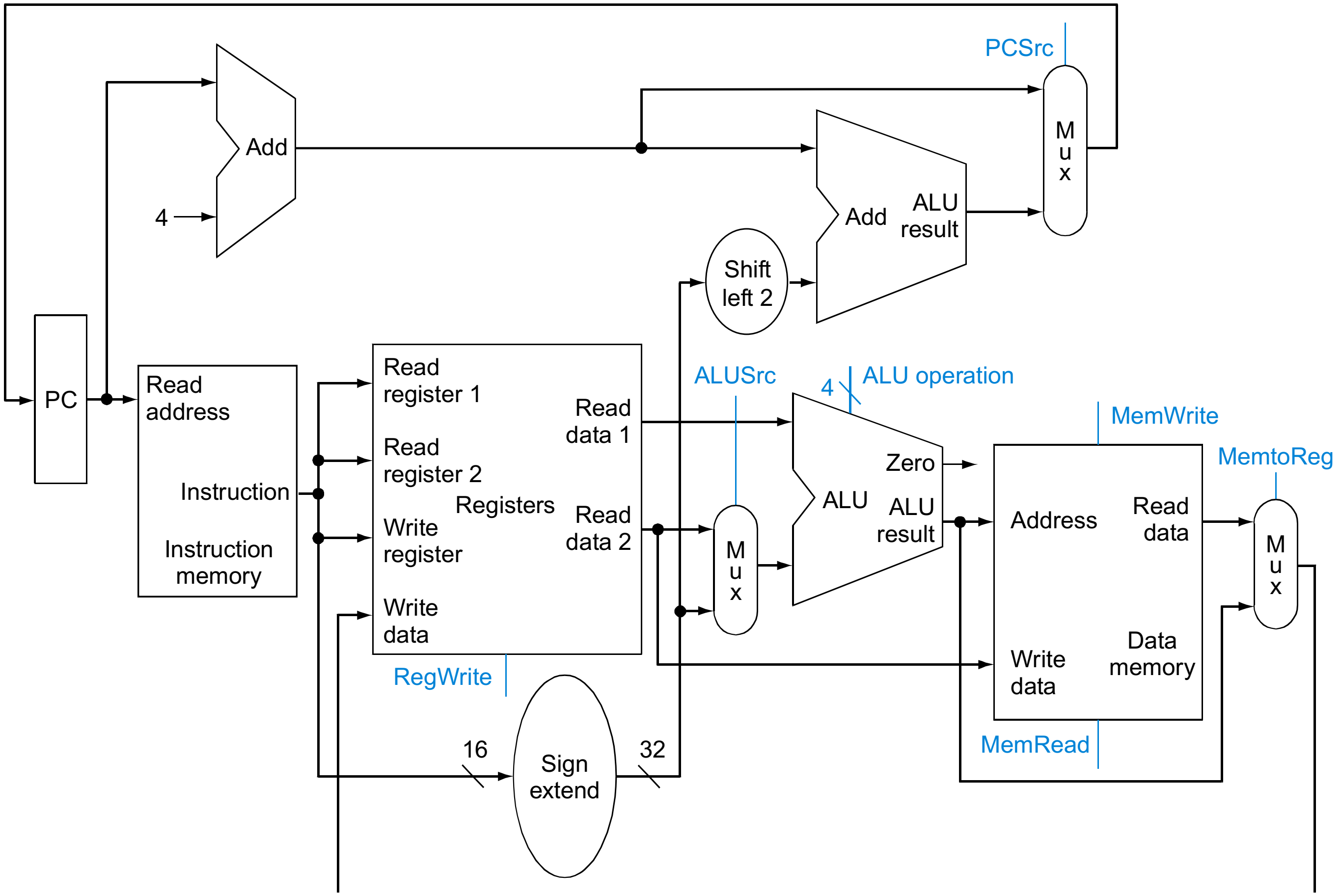
Designing a Multicycle Processor

Abstract View of a Single Cycle Processor

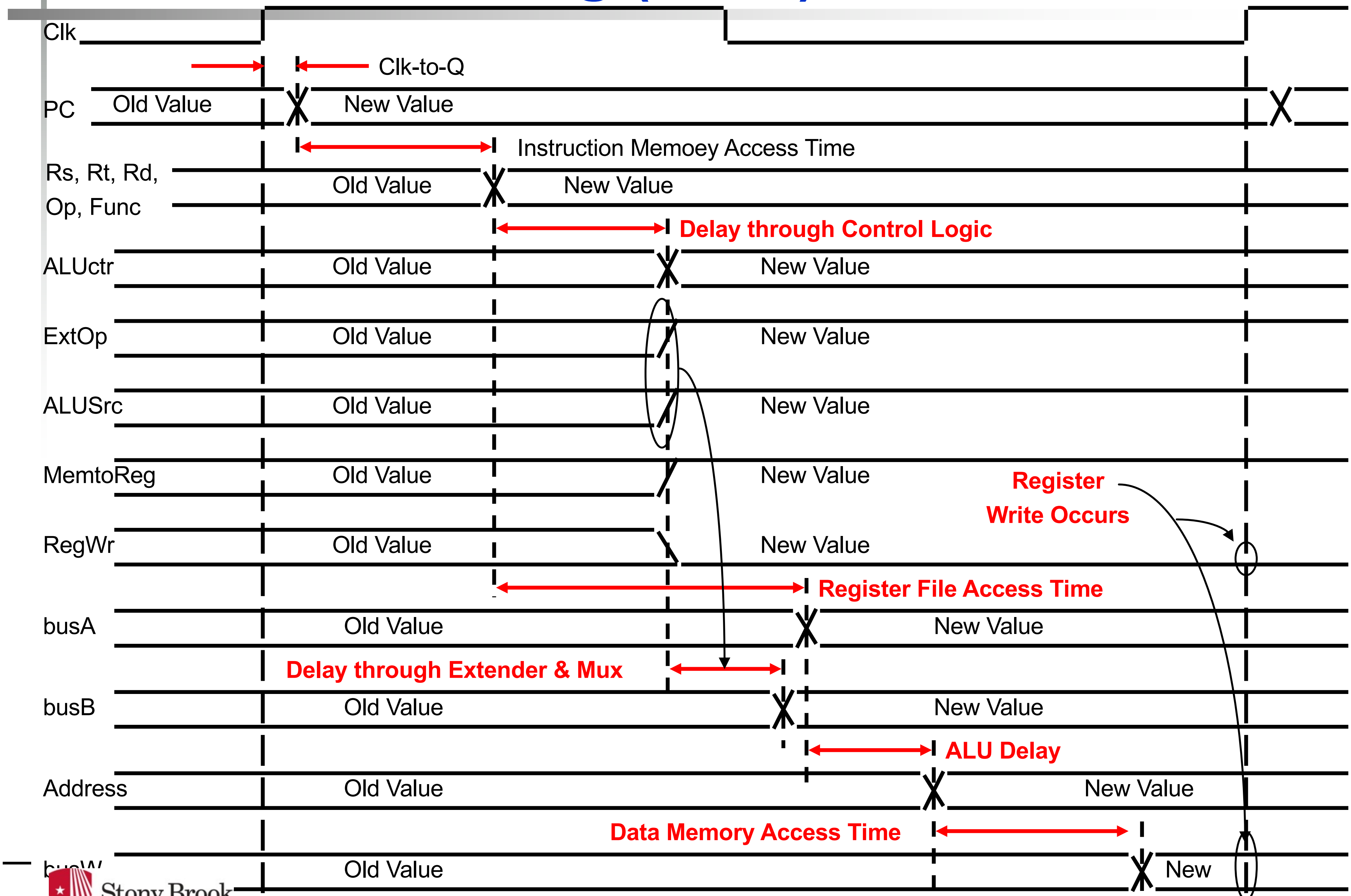


Single Cycle Implementation

- Calculate cycle time assuming negligible delays except:
  - memory (200ps),
    ALU and adders (100ps),
    register file access (50ps)



Worst Case Timing (Load)

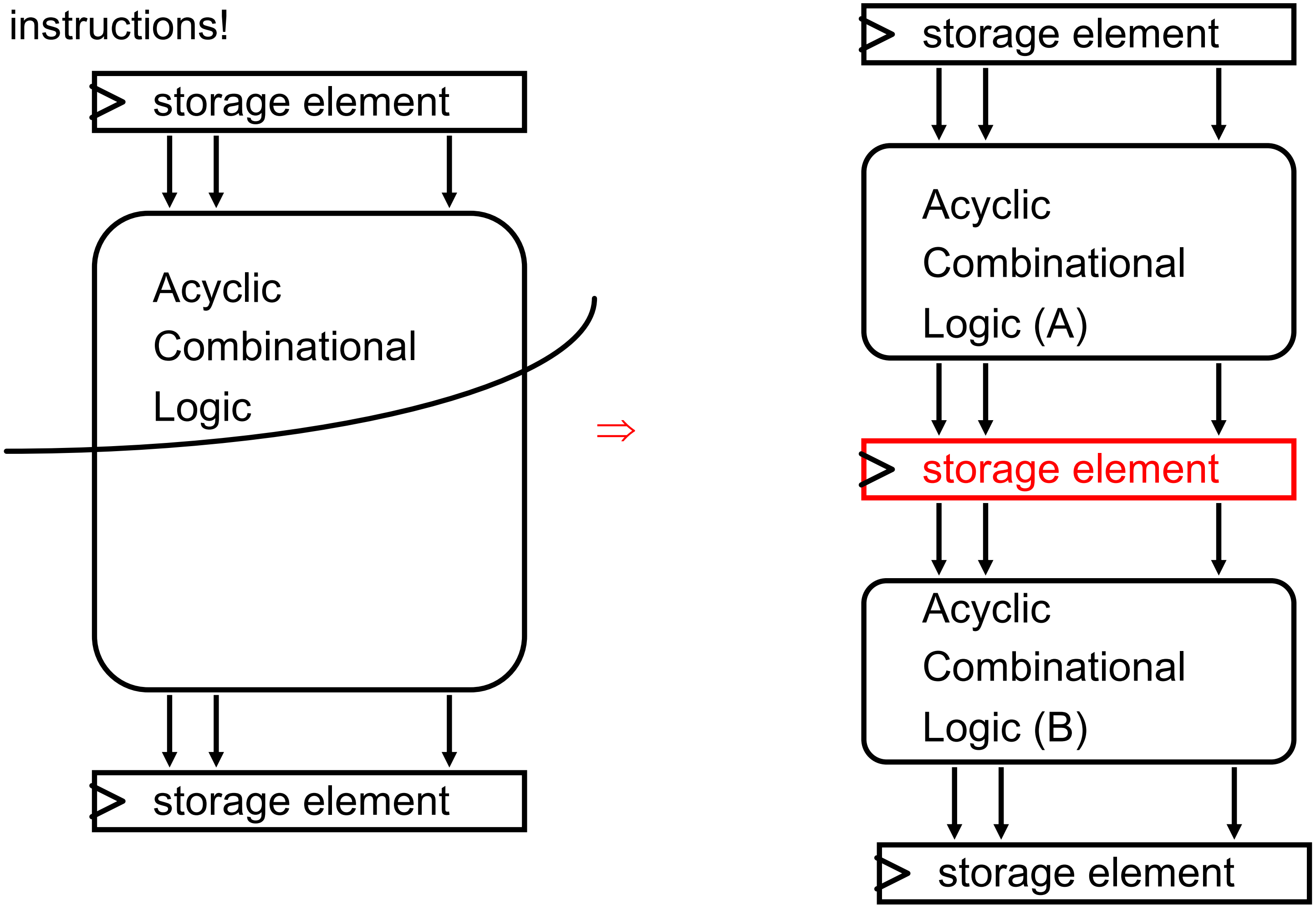


Where We are Headed

- Single Cycle Problems:
  - what if we had a more complicated instruction like floating point?
- One Solution:
  - use a “smaller” cycle time
  - have different instructions take different numbers of cycles
  - a “multicycle” datapath:

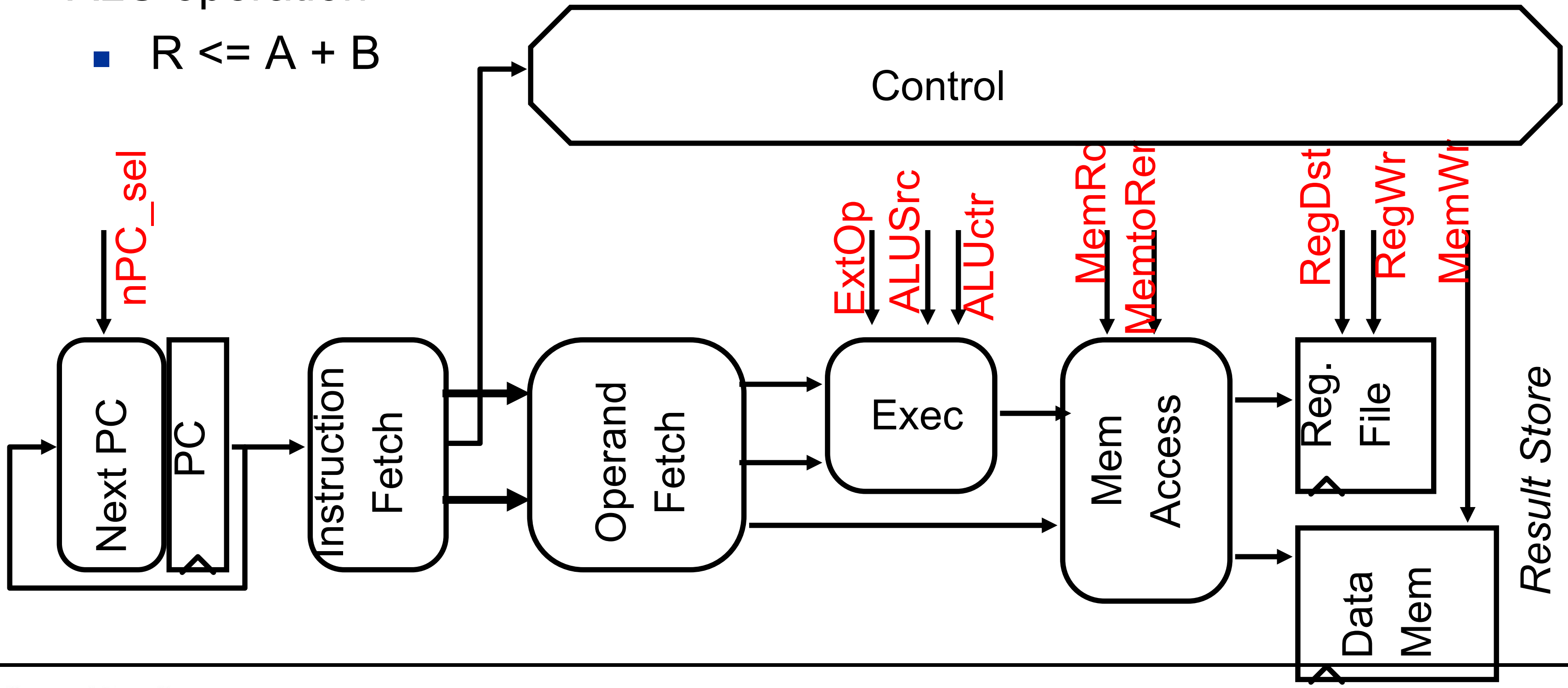
Reducing Cycle Time

- Cut combinational dependency graph and insert register / latch
- Do same work in two fast cycles, rather than one slow one
- May be able to short-circuit path and remove some components for some instructions!



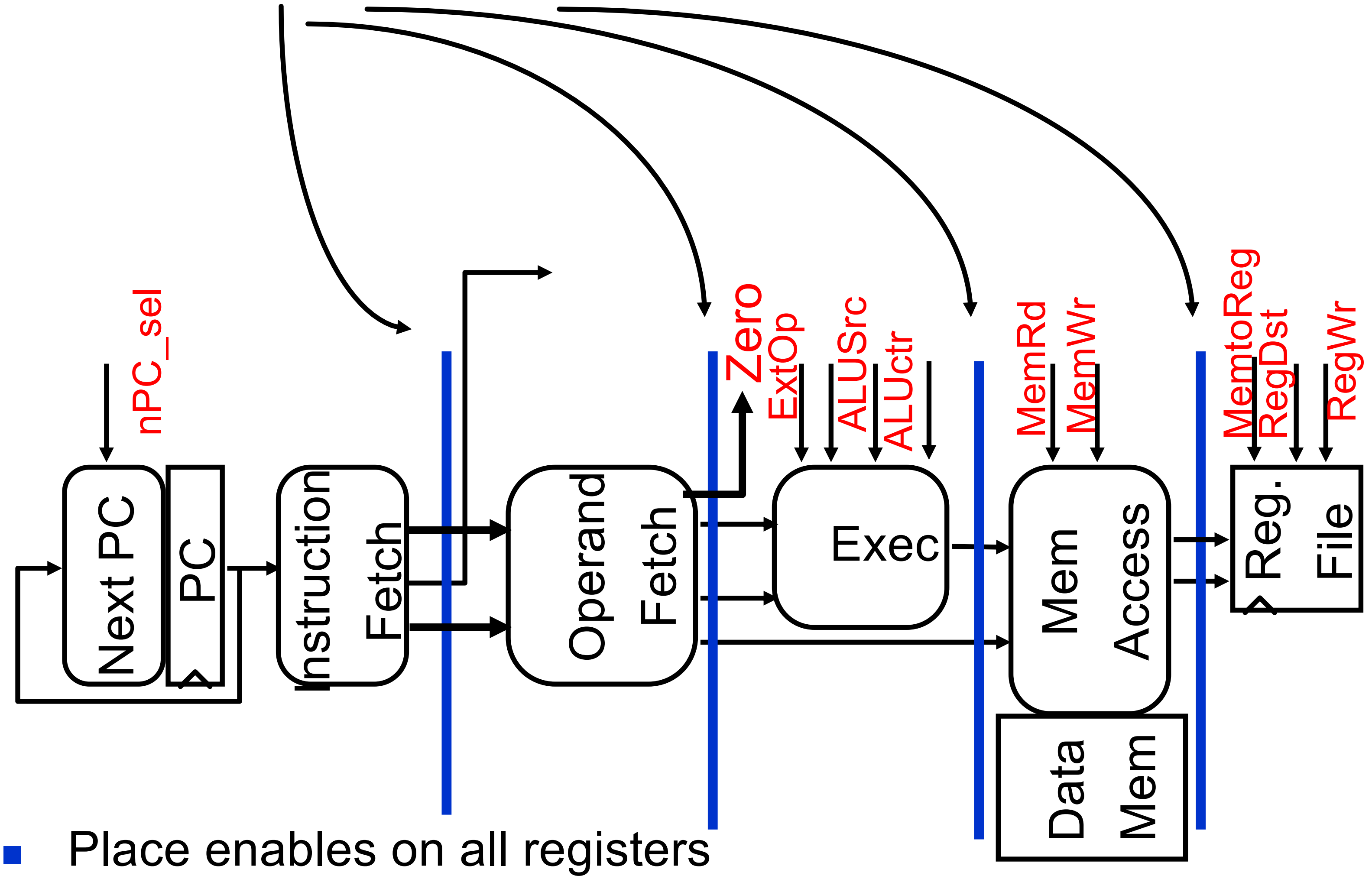
Basic Limits on Cycle Time

- Next address logic
  - PC <= branch ? PC + offset : PC + 4
- Instruction Fetch
  - InstructionReg <= Mem[PC]
- Register Access
  - A <= R[rs]
- ALU operation
  - R <= A + B



Partitioning the CPI=1 Datapath

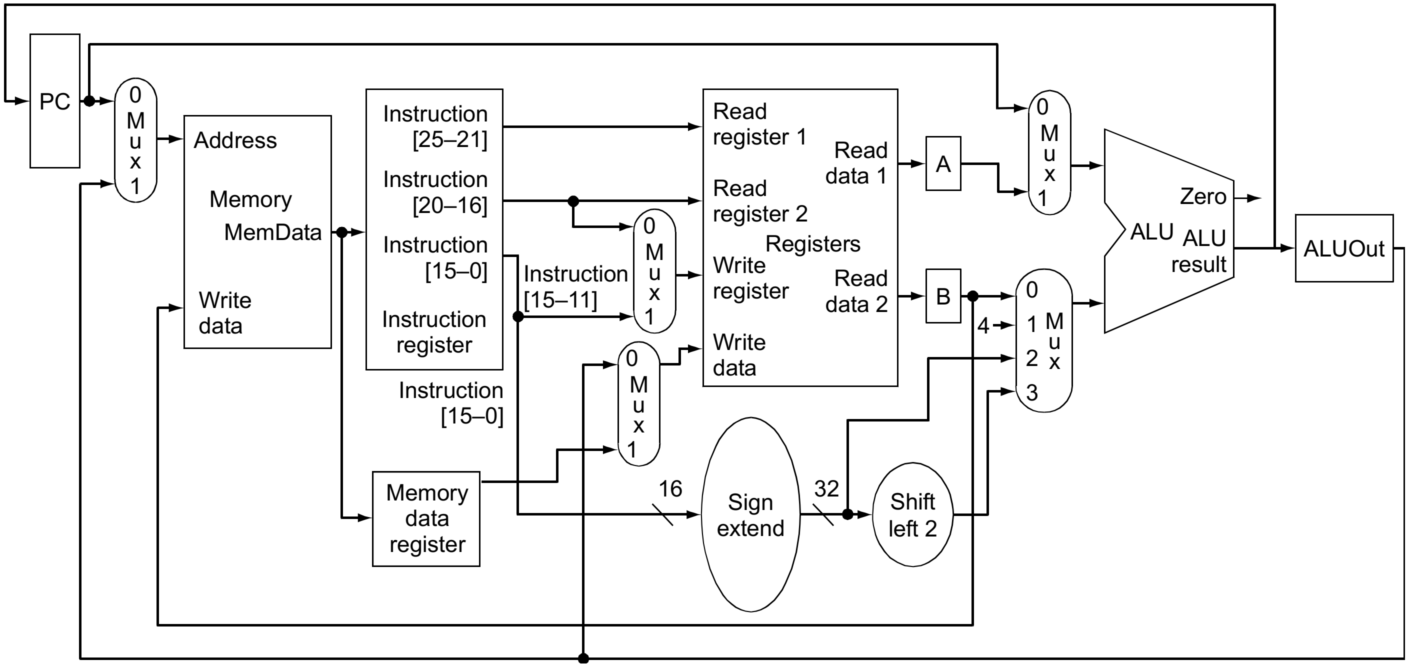
- Add registers between smallest steps



- Place enables on all registers

Multicycle Approach 1/2

- Break up the instructions into steps, each step takes a cycle
  - balance the amount of work to be done
  - restrict each cycle to use only one major functional unit
- At the end of a cycle
  - store values for use in later cycles (easiest thing to do)
  - introduce additional “internal” registers



Multicycle Approach 2/2

- We will be reusing functional units
  - ALU used to compute address and to increment PC
  - Memory used for instruction and data
- Our control signals will not be determined directly by instruction
  - e.g., what should the ALU do for a “subtract” instruction?
- We’ll use a finite state machine for control

Recall: Step-by-step Processor Design

Step 1: ISA => Logical Register Transfers

Step 2: Components of the Datapath

Step 3: RTL + Components => Datapath

Step 4: Datapath + Logical RTs => Physical RTs

Step 5: Physical RTs => Control

Instructions from ISA Perspective

- Consider each instruction from perspective of ISA (at the logical register-transfer level).
- Example:
  - The add instruction changes a register.
  - Register specified by bits 15:11 of instruction.
  - Instruction specified by the PC.
  - New value is the sum ("op") of two registers.
  - Registers specified by bits 25:21 and 20:16 of the instruction
    Reg[Memory[PC][15:11]] <= Reg[Memory[PC][25:21]] op
    Reg[Memory[PC][20:16]]
  - In order to accomplish this we must break up the instruction.
    (kind of like introducing variables when programming)

Breaking Down an Instruction

- ISA definition of arithmetic:

  Reg[Memory[PC][15:11]] <= Reg[Memory[PC][25:21]] op
  Reg[Memory[PC][20:16]]

- Could break down to:
  - IR <= Memory[PC]
  - A <= Reg[IR[25:21]]
  - B <= Reg[IR[20:16]]
  - ALUOut <= A op B
  - Reg[IR[20:16]] <= ALUOut

- And do not forget an important part of the definition of arithmetic!
  - PC <= PC + 4

Idea Behind a Multicycle Approach

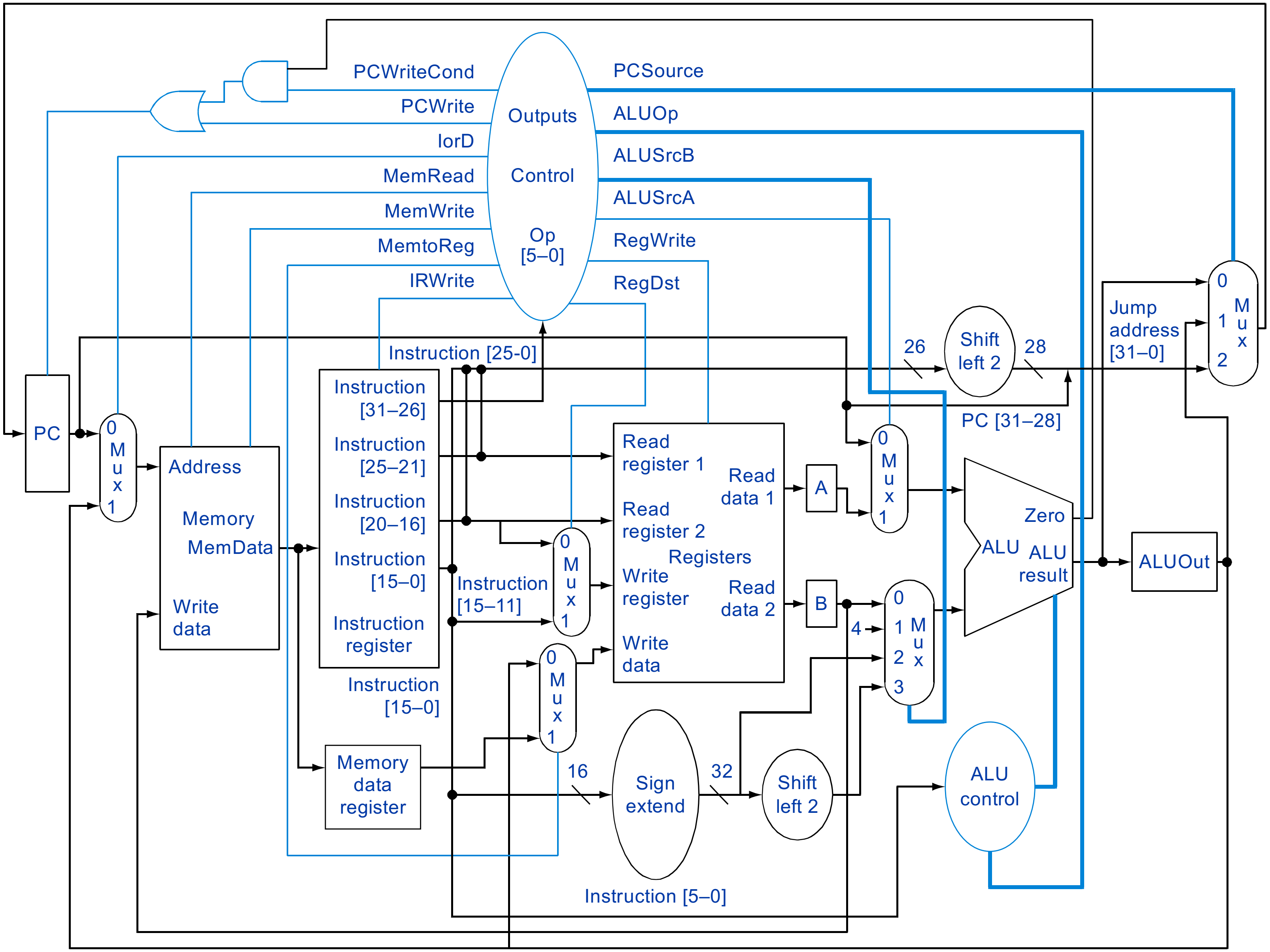
- We define each instruction from the ISA perspective (logical RTL)
- Break it down into steps following our rule that data flows through at most one major functional unit (e.g., balance work across steps)
- Introduce new registers as needed (e.g, A, B, ALUOut, MDR, etc.)
- Finally try and pack as much work into each step

  (avoid unnecessary cycles)

  while also trying to share steps where possible

  (minimizes control, helps to simplify solution)
- Result: Our multicycle Implementation!

Multicycle Processor



Five Execution Steps

- Instruction Fetch
- Instruction Decode and Register Fetch
- Execution, Memory Address Computation, or Branch Completion
- Memory Access or R-type instruction completion
- Write-back step

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

Step 1: Instruction Fetch

- Use PC to get instruction and put it in the Instruction Register.
- Increment the PC by 4 and put the result back in the PC.
- Can be described succinctly using RTL "Register-Transfer Language"

```
    IR <= Memory[PC];
    PC <= PC + 4;
```

Can we figure out the values of the control signals?

What is the advantage of updating the PC now?

Step 2: Instruction Decode and Register Fetch

- Read registers rs and rt in case we need them
- Compute the branch address in case the instruction is a branch
- (Physical) RTL:

```
A <= Reg[IR[25:21]];
B <= Reg[IR[20:16]];
ALUOut <= PC + (sign-extend(IR[15:0]) << 2);
```

- We aren't setting any control lines based on the instruction type

  (we are busy "decoding" it in our control logic)

Step 3 (Instruction Dependent)

- ALU is performing one of three functions, based on instruction type
- Memory Reference:

```
ALUOut <= A + sign-extend(IR[15:0]);
```

- R-type:

```
ALUOut <= A op B;
```

- Branch:

```
if (A==B) PC <= ALUOut;
```

Step 4 (R-type or Memory-Access) and Write-Back Step 5

- **Step 4**
- Loads and stores access memory

```
MDR <= Memory[ALUOut];
          or
Memory[ALUOut] <= B;
```

- R-type instructions finish

```
Reg[IR[15:11]] <= ALUOut;
```

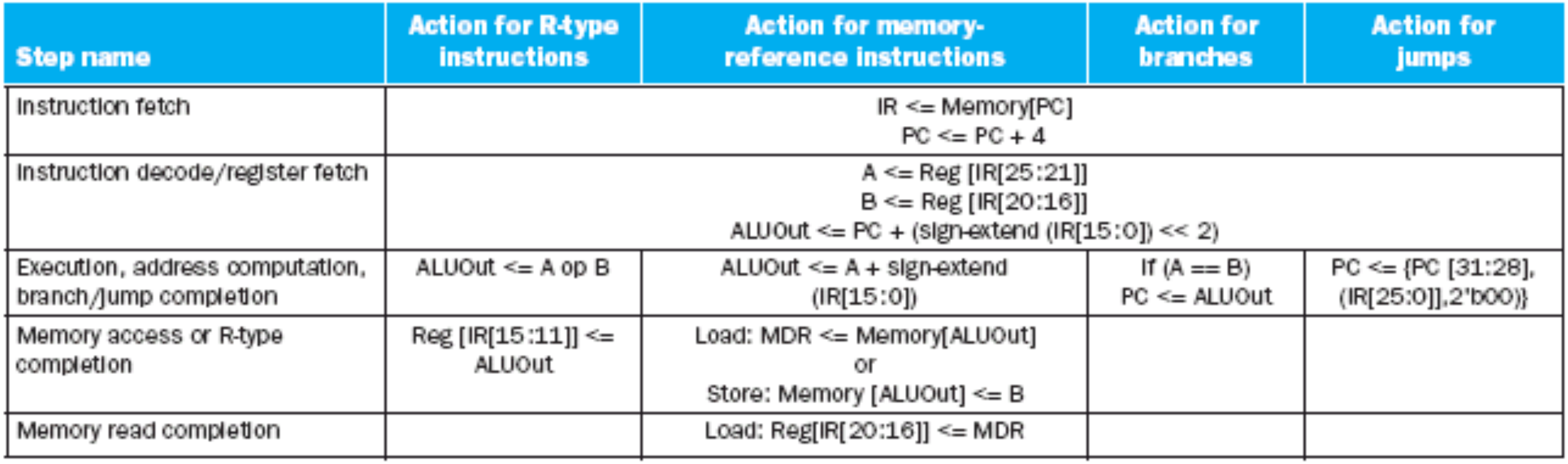
The write actually takes place at the end of the cycle on the edge

- **Write-back step 5**

- Reg[IR[20:16]] <= MDR;

Which instruction needs this?

Summary:



Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR <= Memory[PC] PC <= PC + 4			
Instruction decode/register fetch	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC + (sign-extend (IR[15:0]) << 2)			
Execution, address computation, branch/jump completion	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	If (A == B) PC <= ALUOut	PC <= {PC [31:28], (IR[25:0]],2'b00)}
Memory access or R-type completion	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Memory read completion		Load: Reg[IR[20:16]] <= MDR		

Simple Questions

- How many cycles will it take to execute this code?

```
                lw $t2, 0($t3)
                lw $t3, 4($t3)
                beq $t2, $t3, Label                #assume not taken
                add $t5, $t2, $t3
                sw $t5, 8($t3)
Label:          ...
```

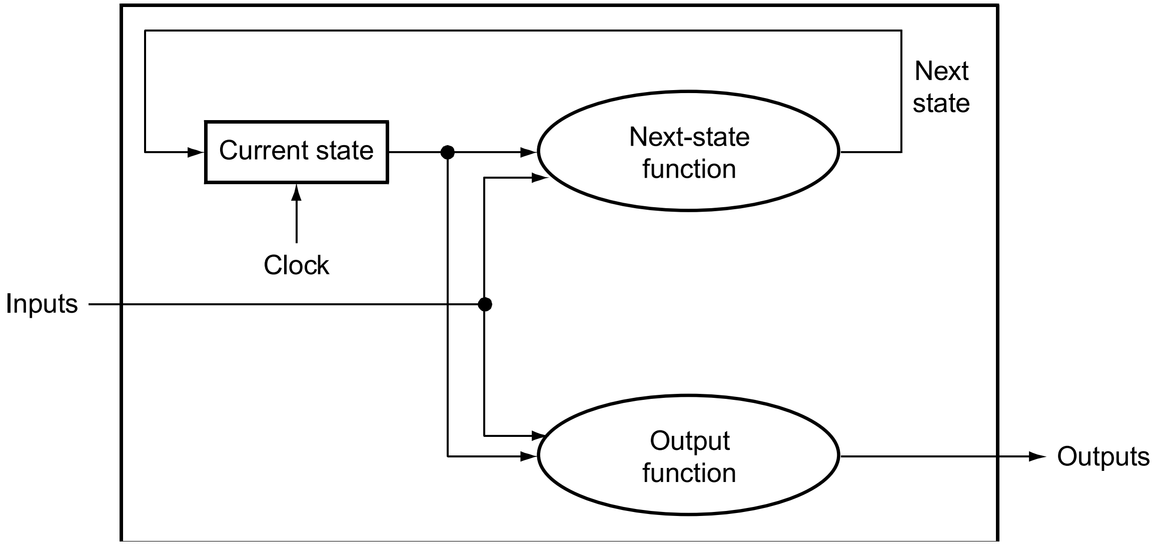
- What is going on during the 8th cycle of execution?
- In what cycle does the actual addition of $t2 and $t3 takes place?

Implementing the Control for a Multicycle Processor

- Value of control signals is dependent upon:
  - what instruction is being executed
  - which step is being performed
- Use the information we've accumulated to specify a finite state machine
  - specify the finite state machine graphically, or
  - use microprogramming
- Implementation can be derived from specification

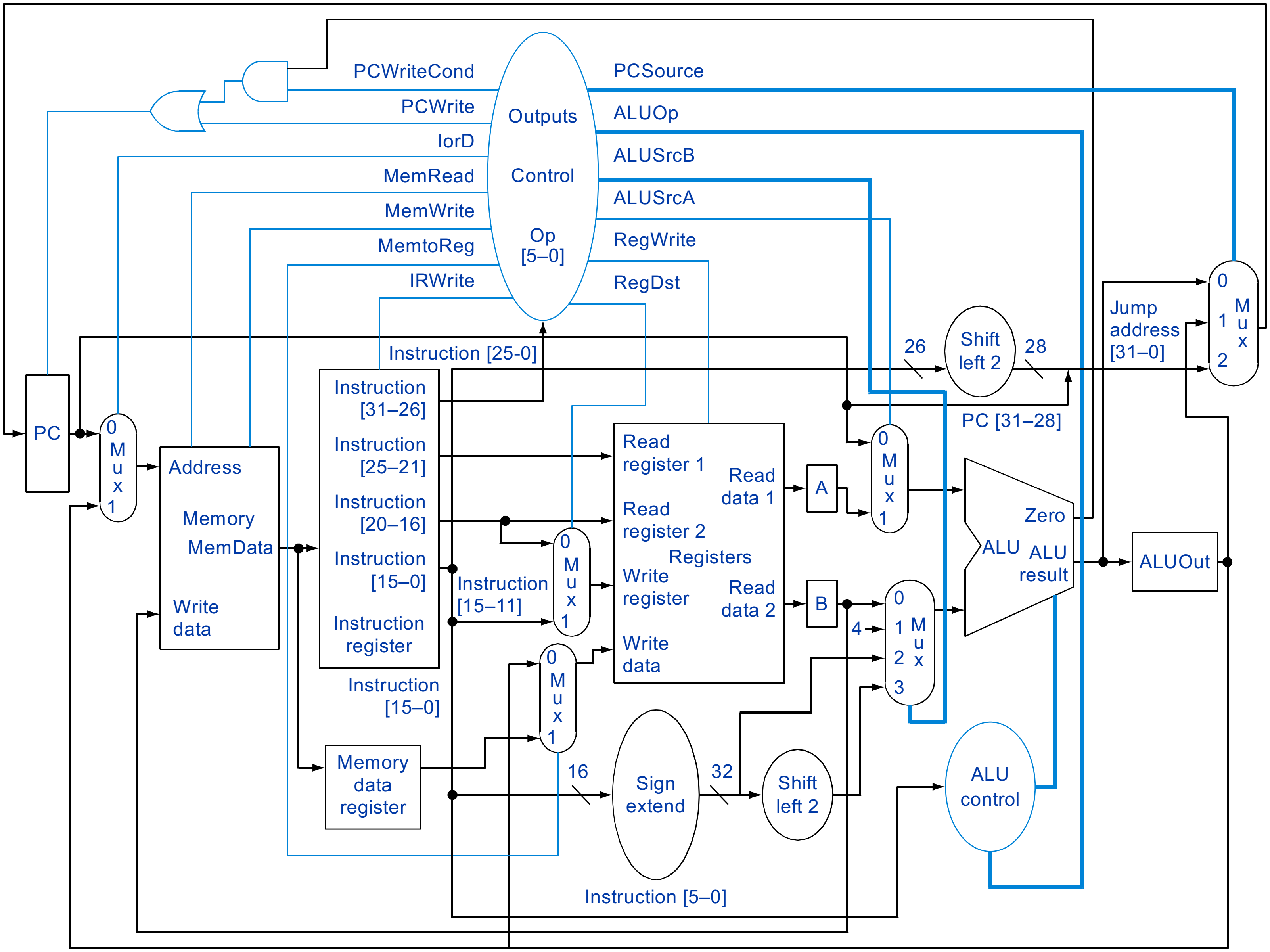
Review: Finite State Machines

- Finite state machines:
  - a set of states and
  - next state function (determined by current state and the input)
  - output function (determined by current state and possibly input)



- We’ll use a Moore machine for the output function
  - output based only on current state

Multicycle Processor



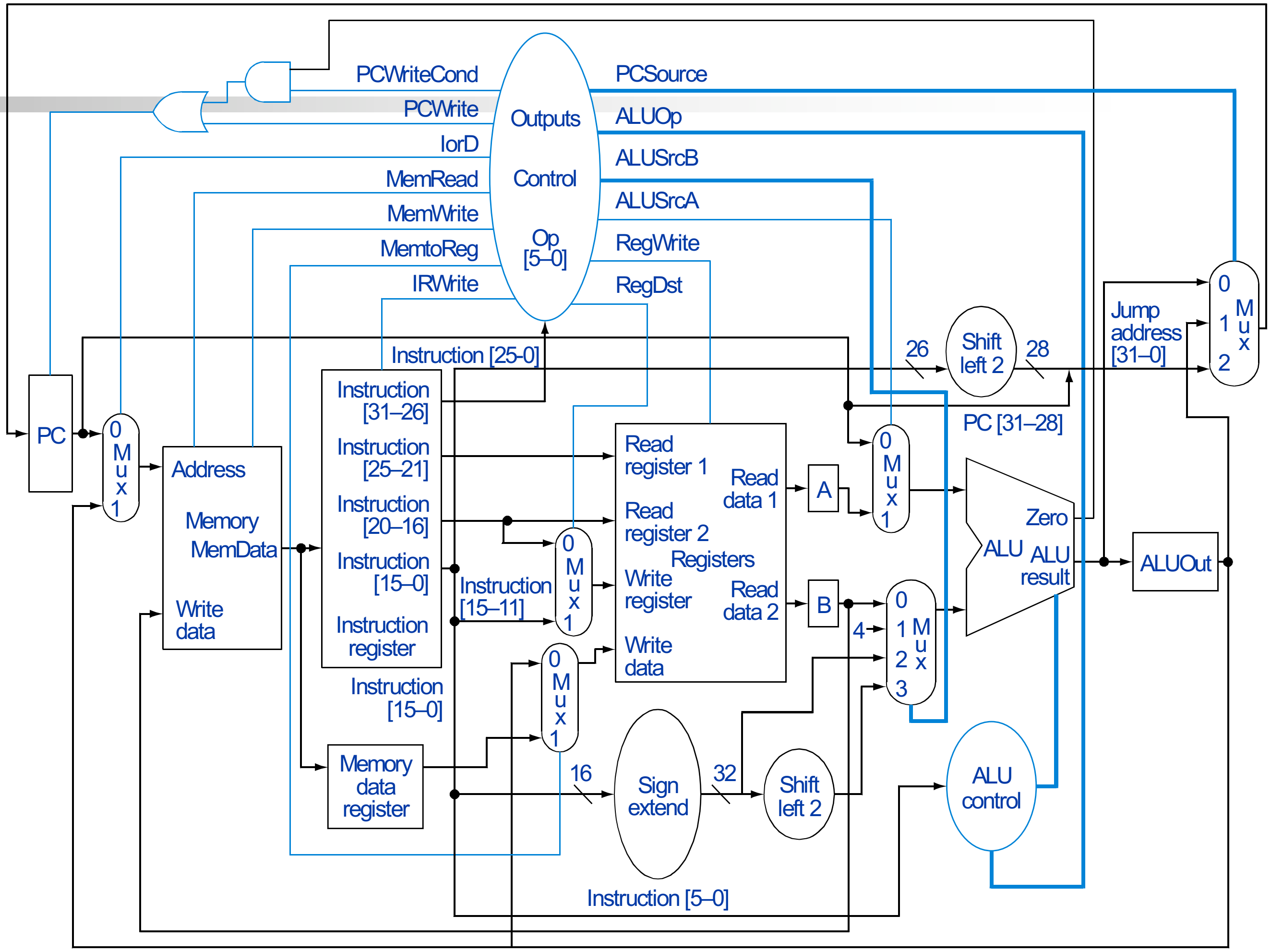
Multicycle Processor

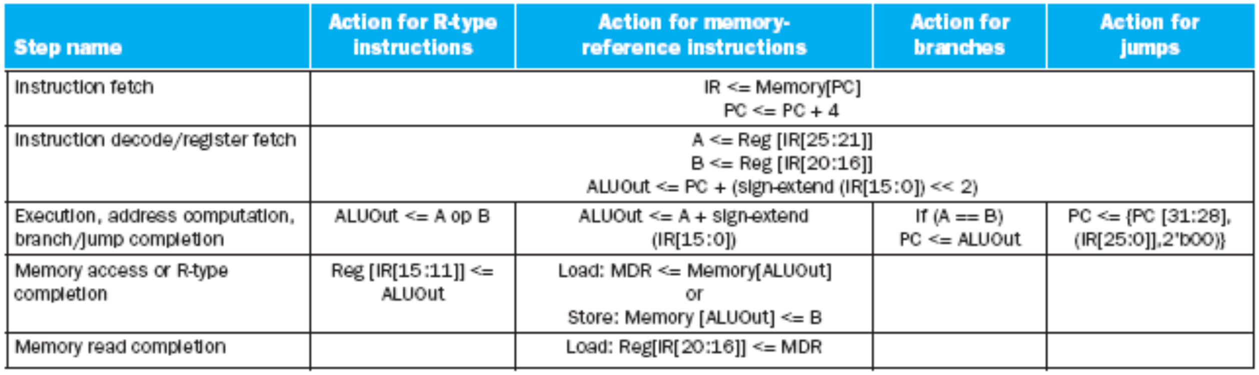
State 0 (IF & PC+4)

- IorD =0
- MemRead=1
- IRWrite=1
- ALUSrcA=0
- ALUScrB=01
- ALUop=00(add)
- PCSource =00
- PCWrite=1

State 1 (IDcd+RF)

- ALUSrcA=0
- ALUScrB=11
- ALUop=00(add)

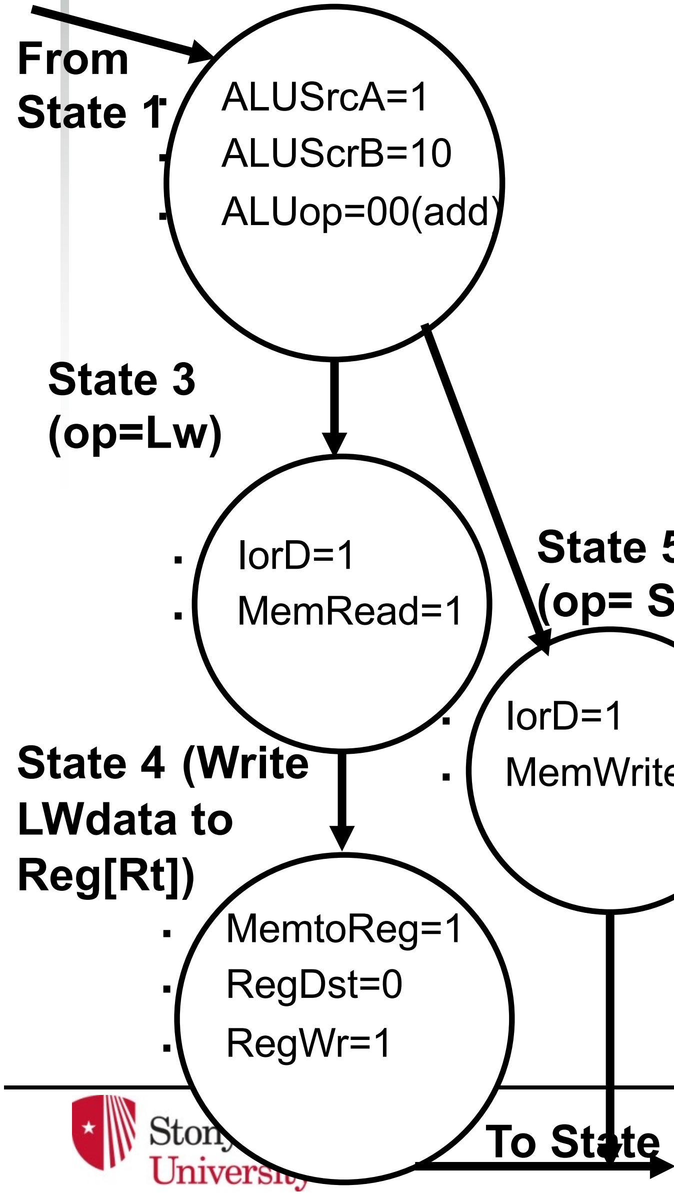




Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR <= Memory[PC] PC <= PC + 4			
Instruction decode/register fetch	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC + (sign-extend (IR[15:0]) << 2)			
Execution, address computation, branch/jump completion	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	If (A == B) PC <= ALUOut	PC <= {PC [31:28], (IR[25:0]],2'b00)}
Memory access or R-type completion	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Memory read completion		Load: Reg[IR[20:16]] <= MDR		

Multicycle Processor

State 2 (op=Lw/Sw)



From State 1

- ALUSrcA=1
- ALUScrB=10
- ALUop=00(add)

State 3 (op=Lw)

- IorD=1
- MemRead=1

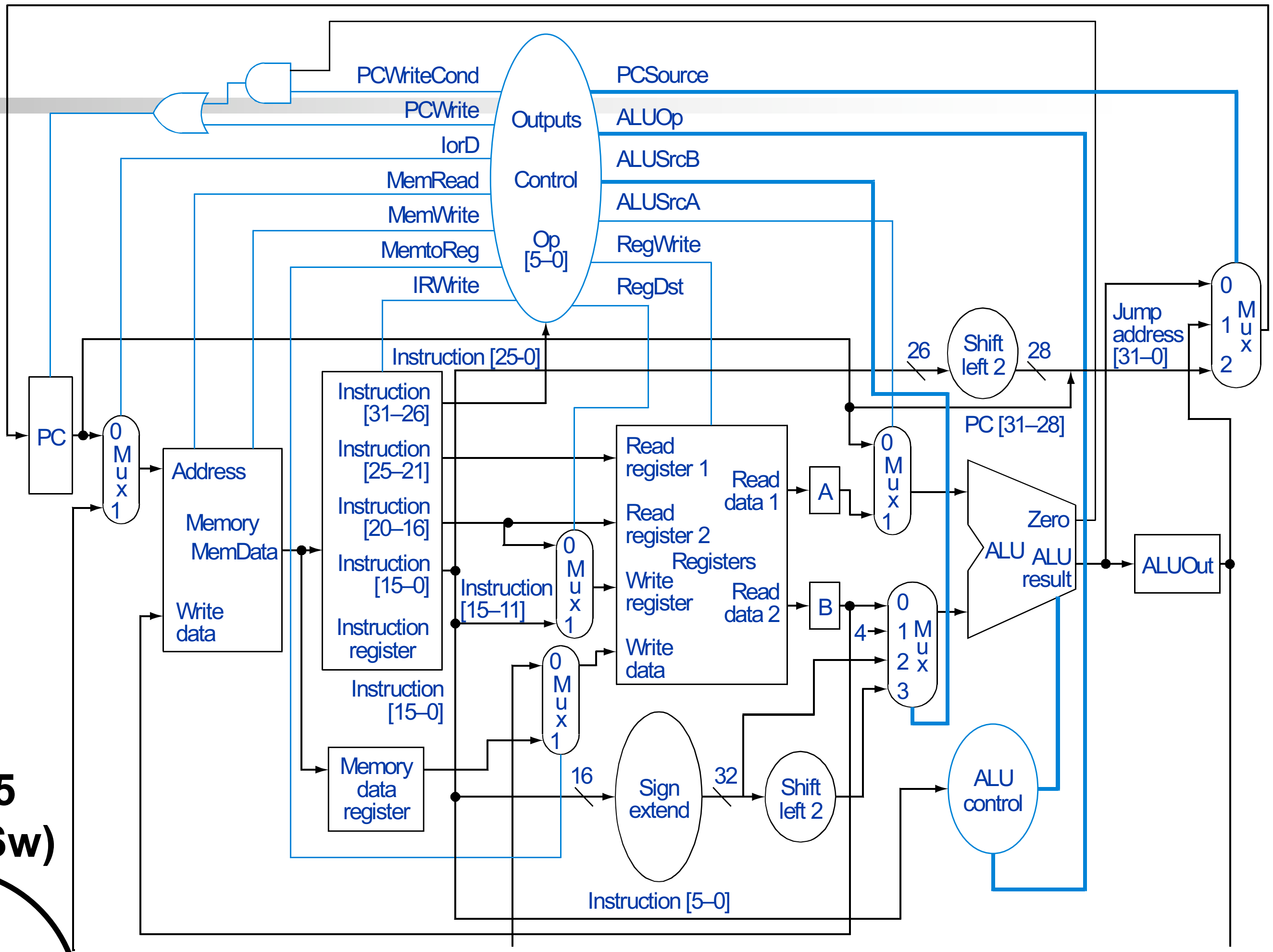
State 5 (op= Sw)

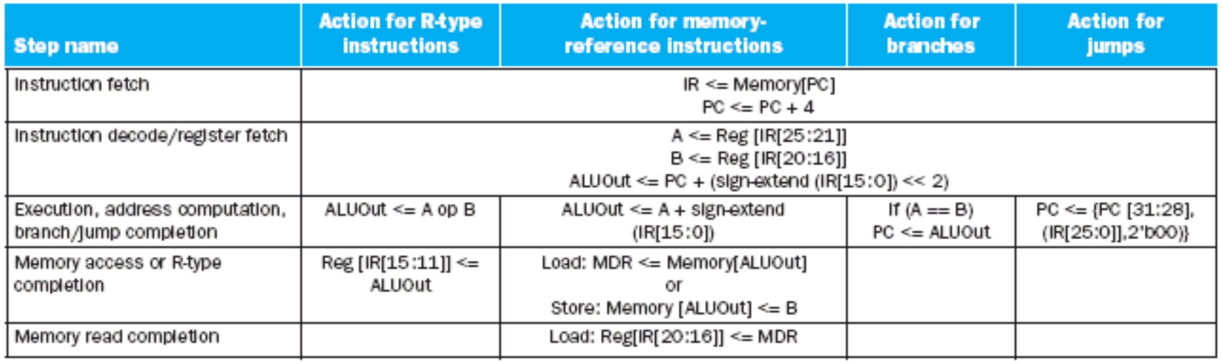
- IorD=1
- MemWrite=1

State 4 (Write LWdata to Reg[Rt])

- MemtoReg=1
- RegDst=0
- RegWr=1

To State 0

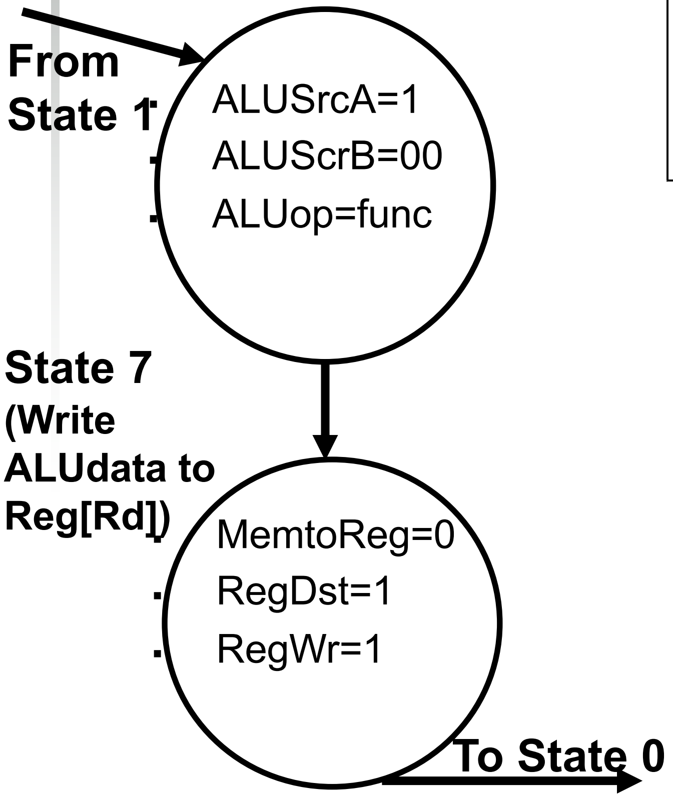




Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR <= Memory[PC] PC <= PC + 4			
Instruction decode/register fetch	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC + (sign-extend (IR[15:0]) << 2)			
Execution, address computation, branch/jump completion	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	If (A == B) PC <= ALUOut	PC <= {PC [31:28], (IR[25:0]],2'b00)}
Memory access or R-type completion	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Memory read completion		Load: Reg[IR[20:16]] <= MDR		

Multicycle Processor

State 6 (op=R-type)



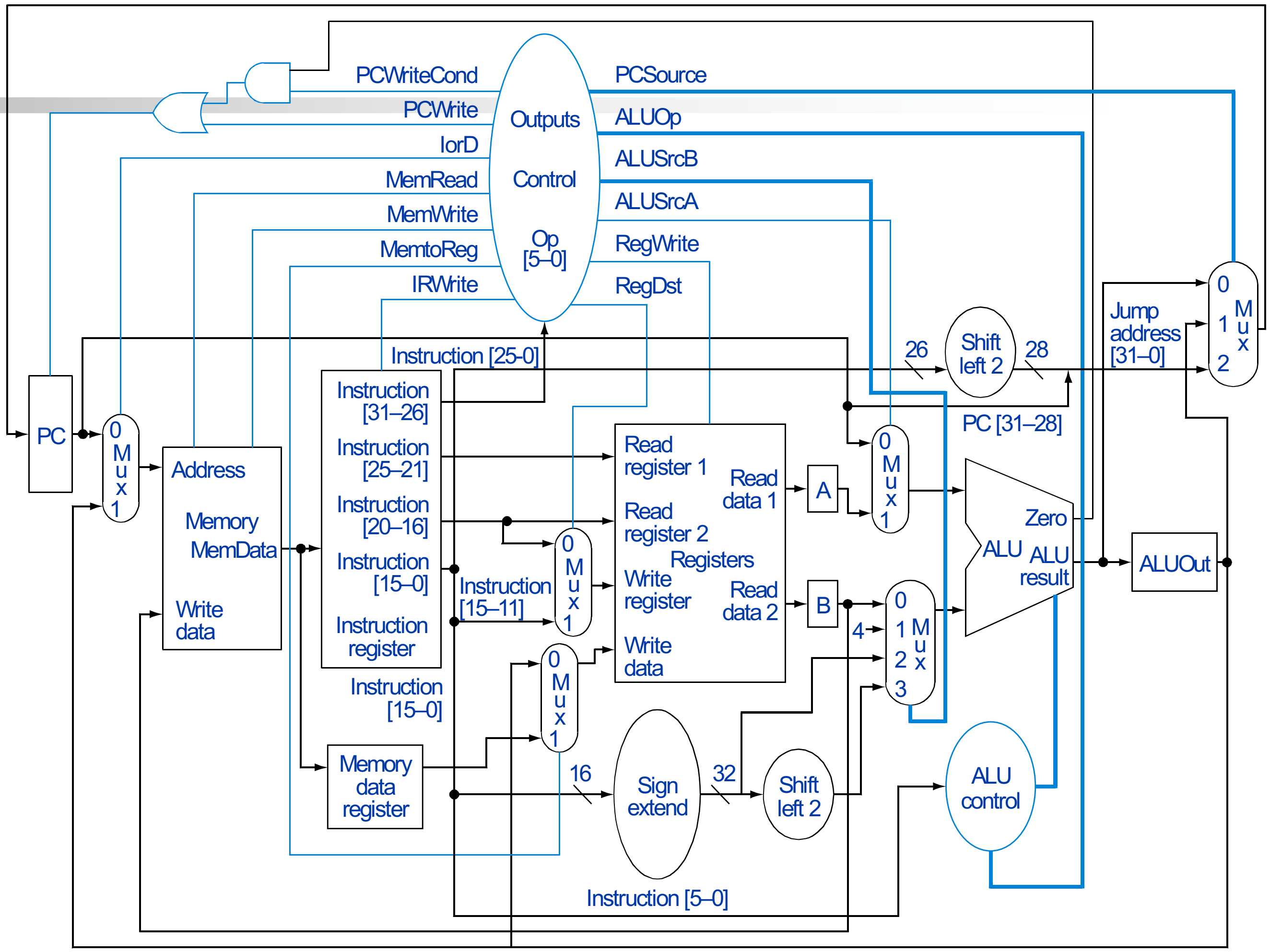
From State 1

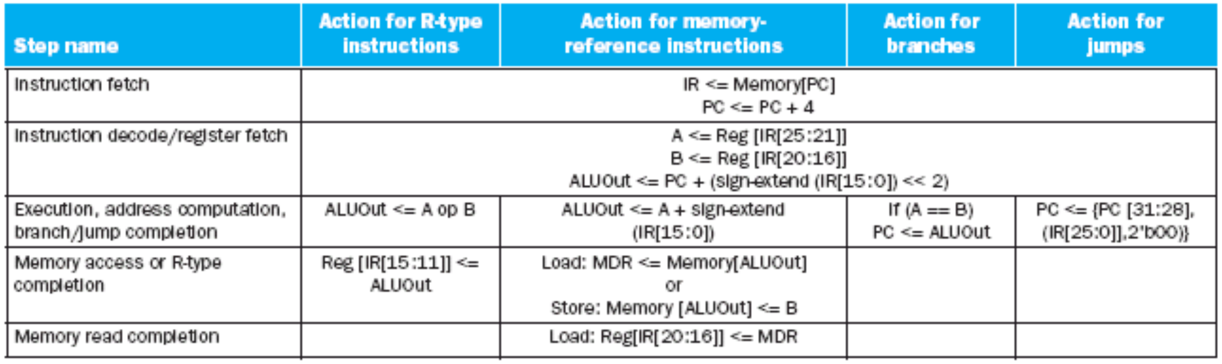
ALUSrcA=1
ALUScrB=00
ALUop=func

State 7 (Write ALUdata to Reg[Rd])

MemtoReg=0
RegDst=1
RegWr=1

To State 0



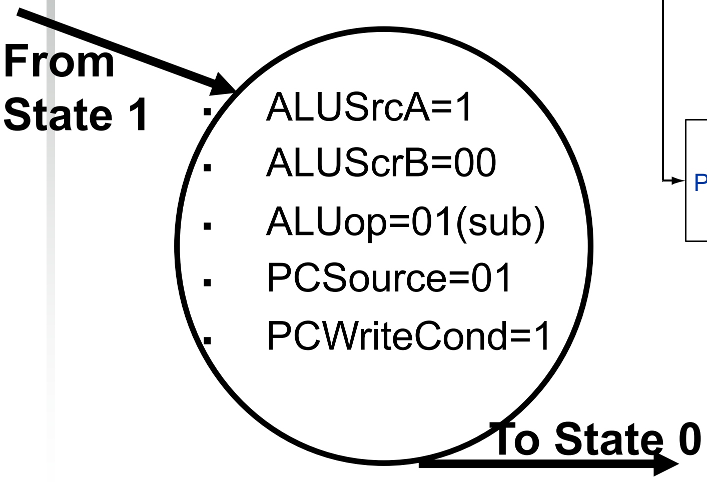


Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR <= Memory[PC] PC <= PC + 4			
Instruction decode/register fetch	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC + (sign-extend (IR[15:0]) << 2)			
Execution, address computation, branch/jump completion	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	if (A == B) PC <= ALUOut	PC <= {PC [31:28], (IR[25:0]],2'b00)}
Memory access or R-type completion	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Memory read completion		Load: Reg[IR[20:16]] <= MDR		

Stony Brook University

Multicycle Processor

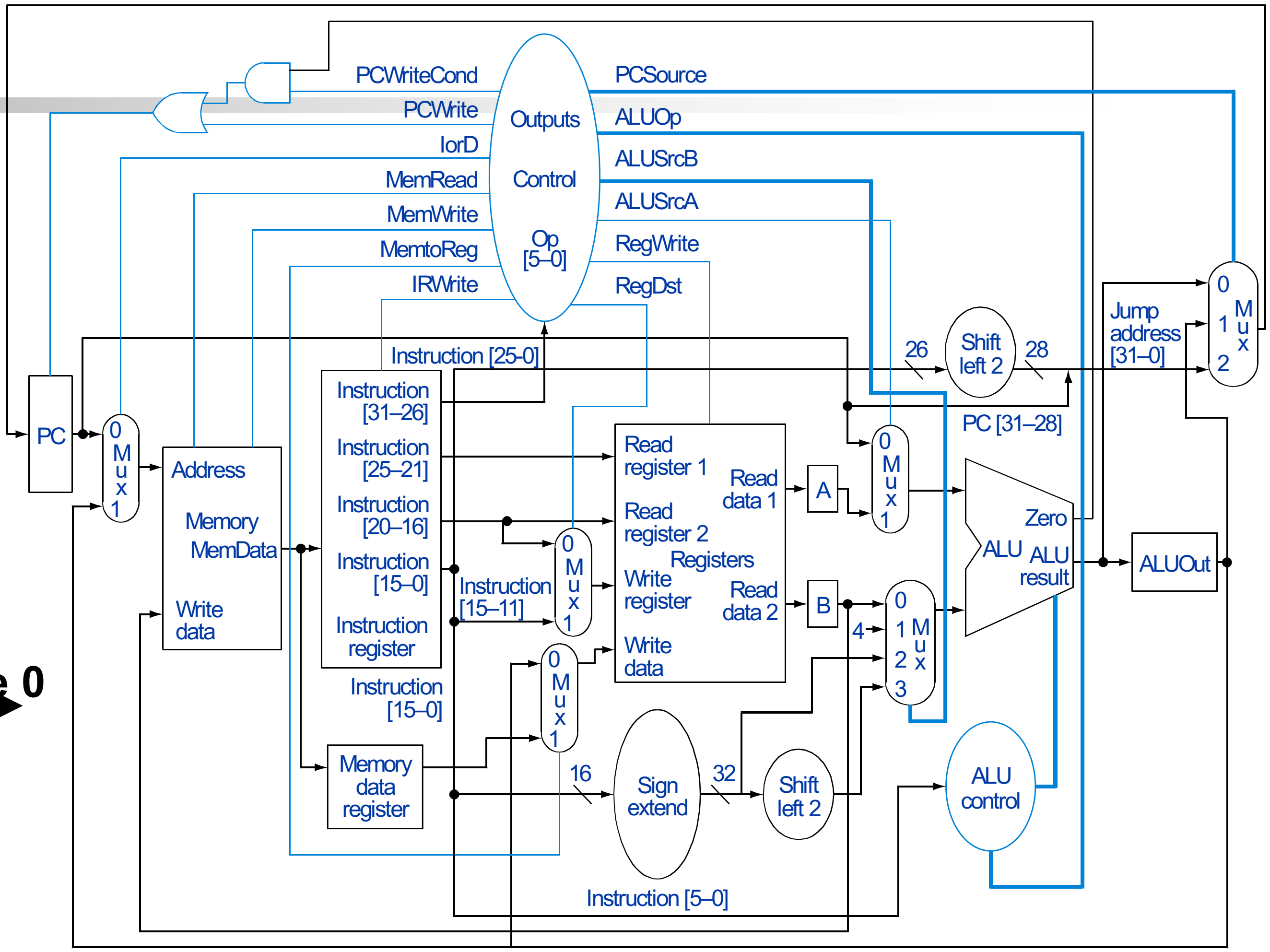
State 8 (op=BEQ)

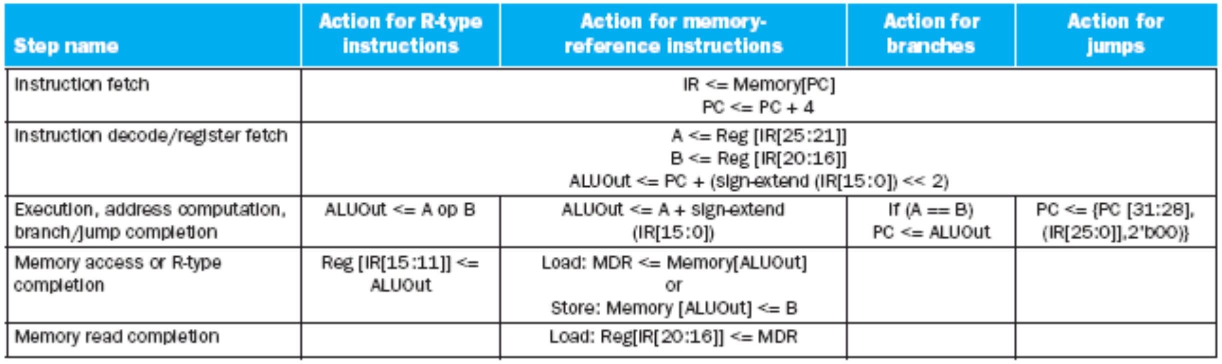


From State 1

- ALUSrcA=1
- ALUScrB=00
- ALUop=01(sub)
- PCSource=01
- PCWriteCond=1

To State 0

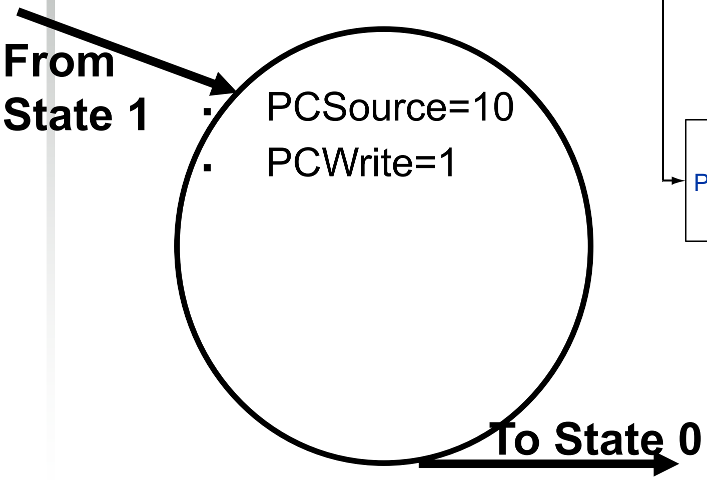


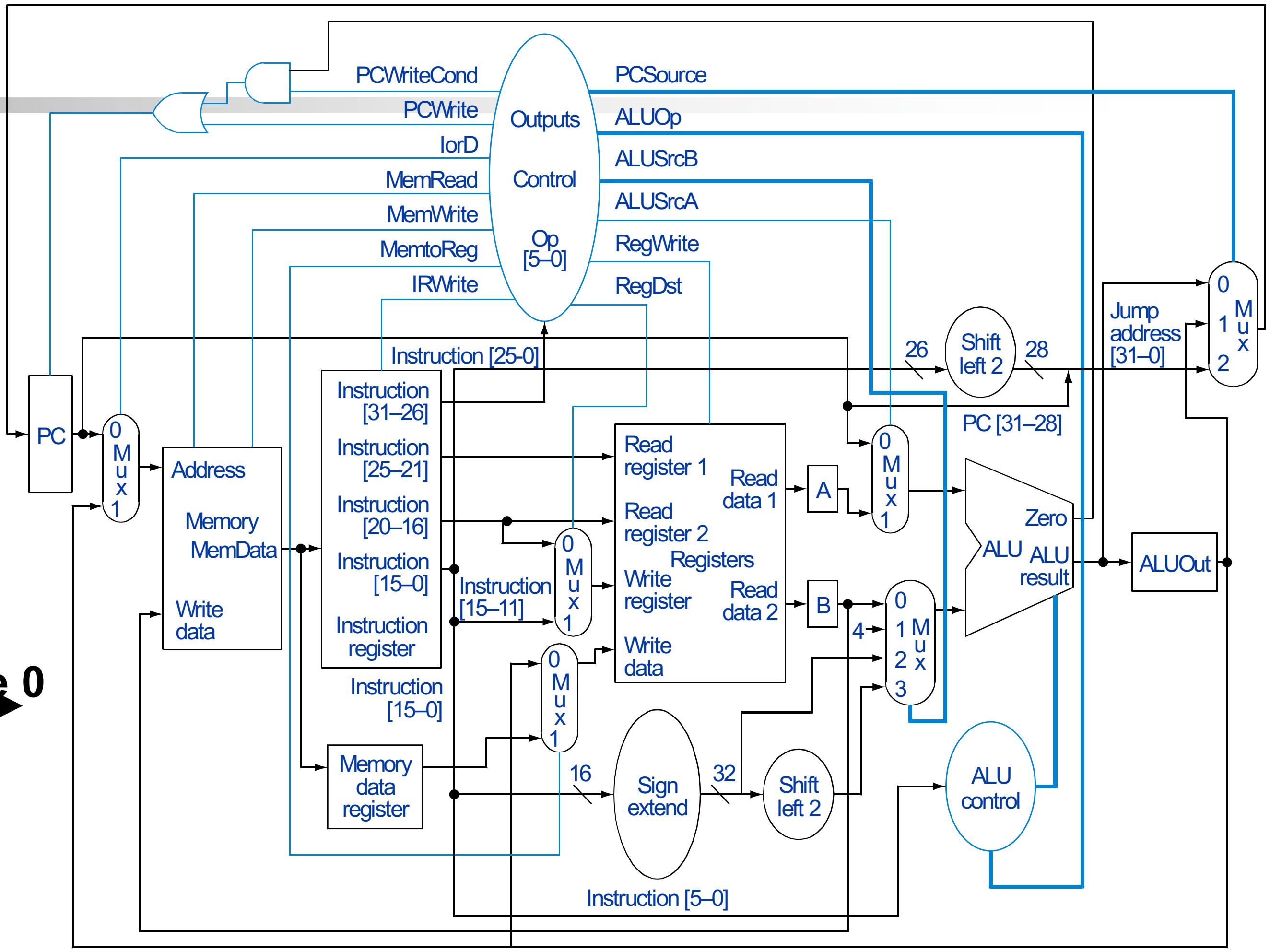


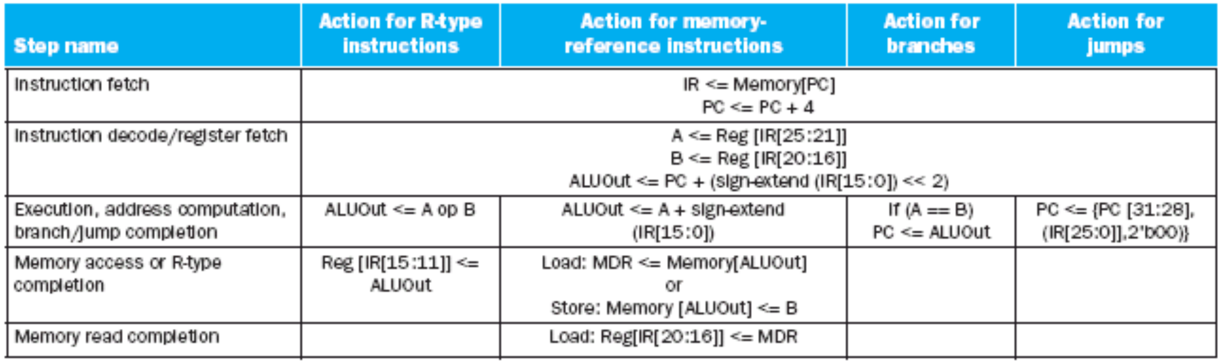
Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR <= Memory[PC] PC <= PC + 4			
Instruction decode/register fetch	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC + (sign-extend (IR[15:0]) << 2)			
Execution, address computation, branch/jump completion	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	if (A == B) PC <= ALUOut	PC <= {PC [31:28], (IR[25:0]],2'b00)}
Memory access or R-type completion	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Memory read completion		Load: Reg[IR[20:16]] <= MDR		

Multicycle Processor

State 9 (op=J)





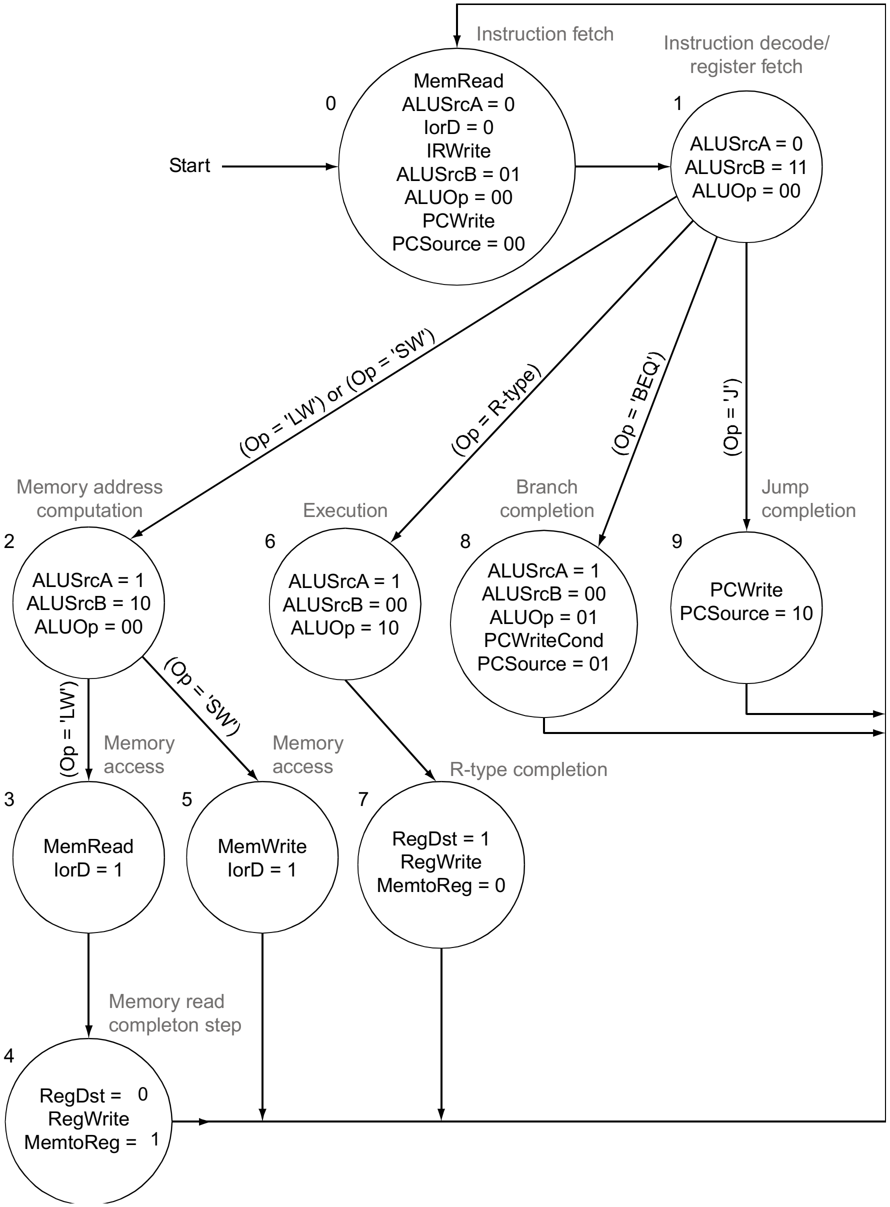


Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR <= Memory[PC] PC <= PC + 4			
Instruction decode/register fetch	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC + (sign-extend (IR[15:0]) << 2)			
Execution, address computation, branch/jump completion	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	If (A == B) PC <= ALUOut	PC <= {PC [31:28], (IR[25:0]],2'b00)}
Memory access or R-type completion	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Memory read completion		Load: Reg[IR[20:16]] <= MDR		

Graphical Specification of FSM

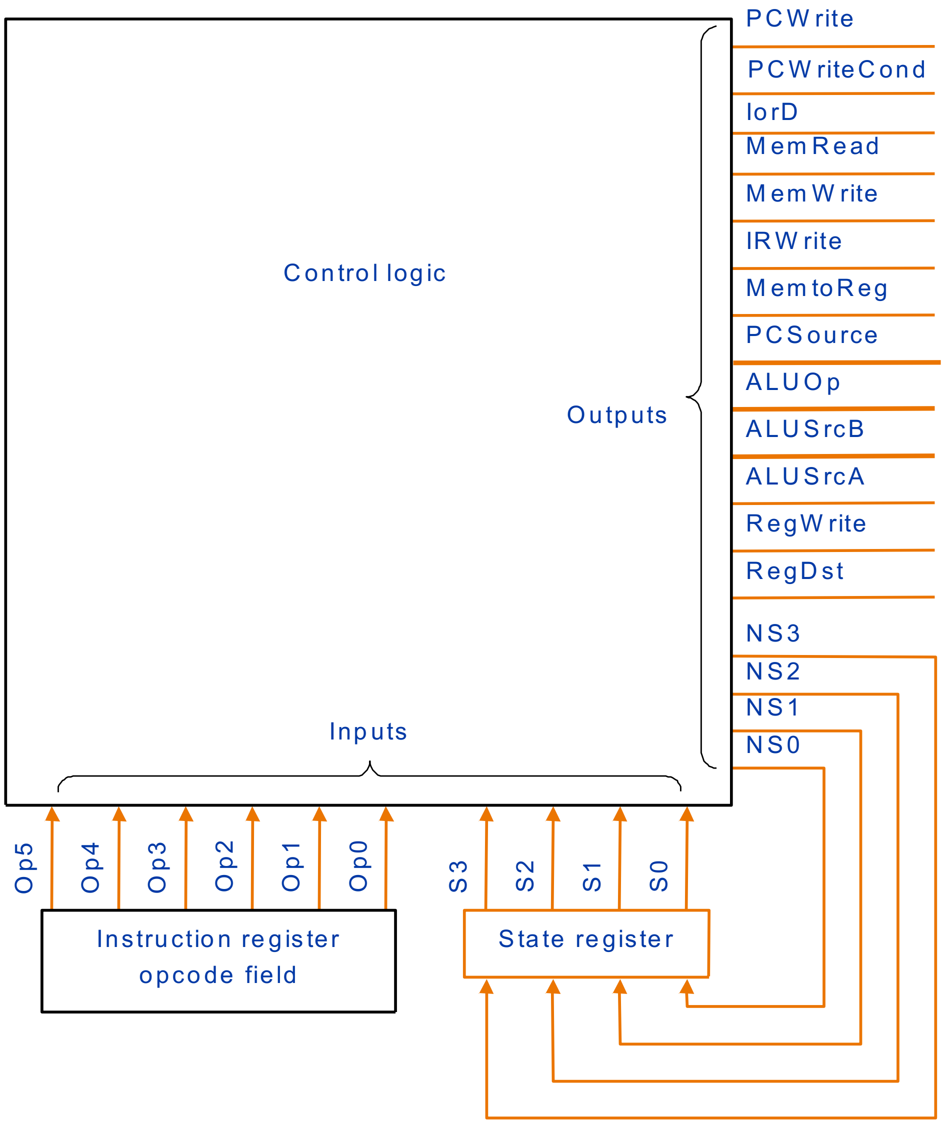
- Note:
  - don’t care if not mentioned
  - asserted if name only
  - otherwise exact value

- How many state bits will we need?

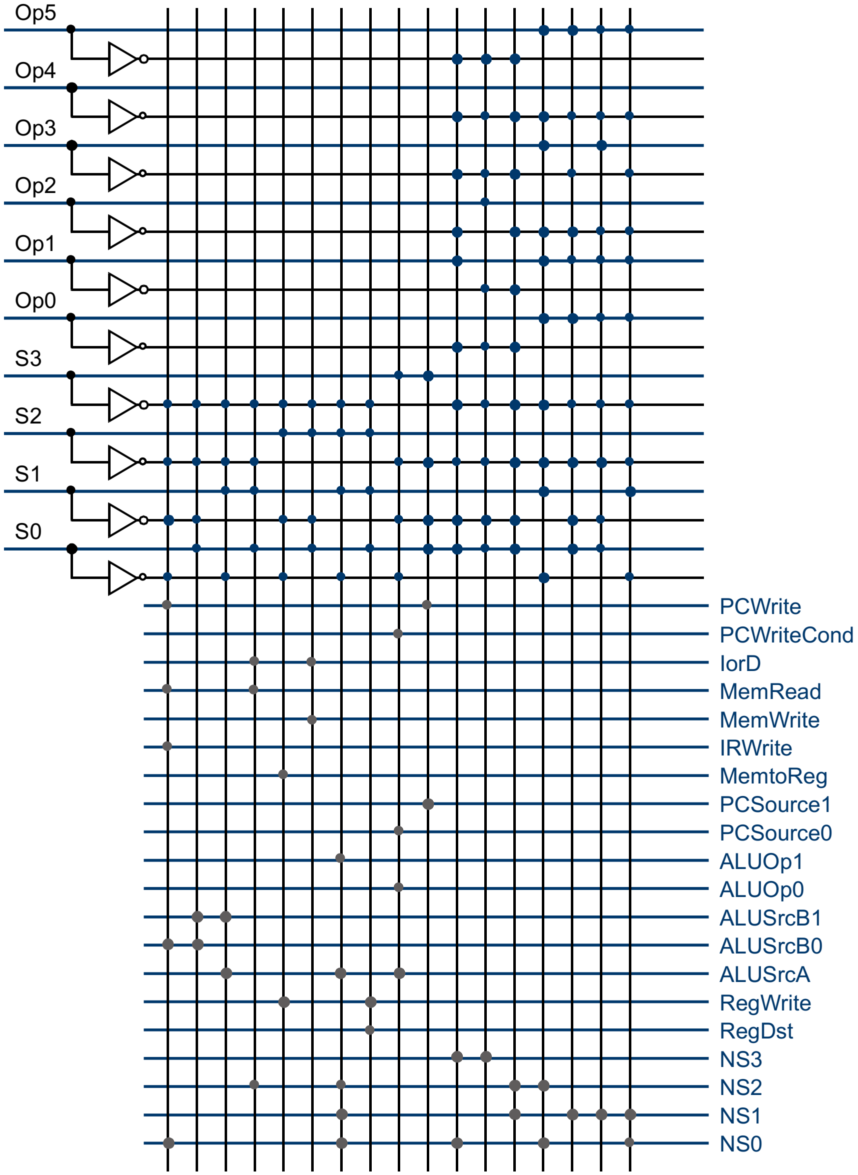


Finite State Machine for Control

- Implementation:

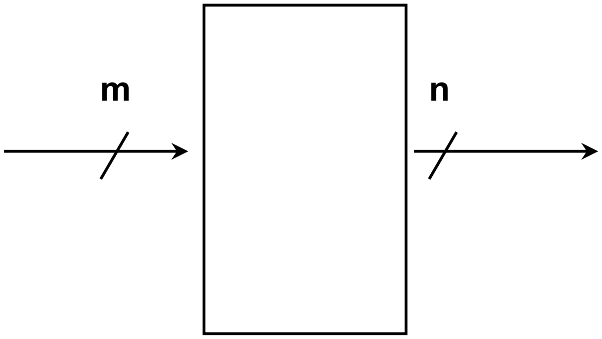


PLA Implementation



ROM Implementation

- ROM = "Read Only Memory"
  - values of memory locations are fixed ahead of time
- A ROM can be used to implement a truth table
  - if the address is m-bits, we can address 2^m entries in the ROM.
  - our outputs are the bits of data that the address points to.



0	0	0	0	0	1	1
0	0	1	1	1	0	0
0	1	0	1	1	0	0
0	1	1	1	0	0	0
1	0	0	0	0	0	0
1	0	1	0	0	0	1
1	1	0	0	1	1	0
1	1	1	0	1	1	1

2^m is the memory ”depth", and n is the "width"

ROM Implementation

- How many inputs are there?
  6 bits for opcode, 4 bits for state = 10 address lines
  (i.e., 2^{10} = 1024 different addresses)
- How many outputs are there?
  16 datapath-control outputs, 4 state bits = 20 outputs
- ROM is 2^{10} x 20 = 20K bits (and a rather unusual size)
- Rather wasteful, since for lots of the entries, the outputs are the same
  — i.e., opcode is often ignored

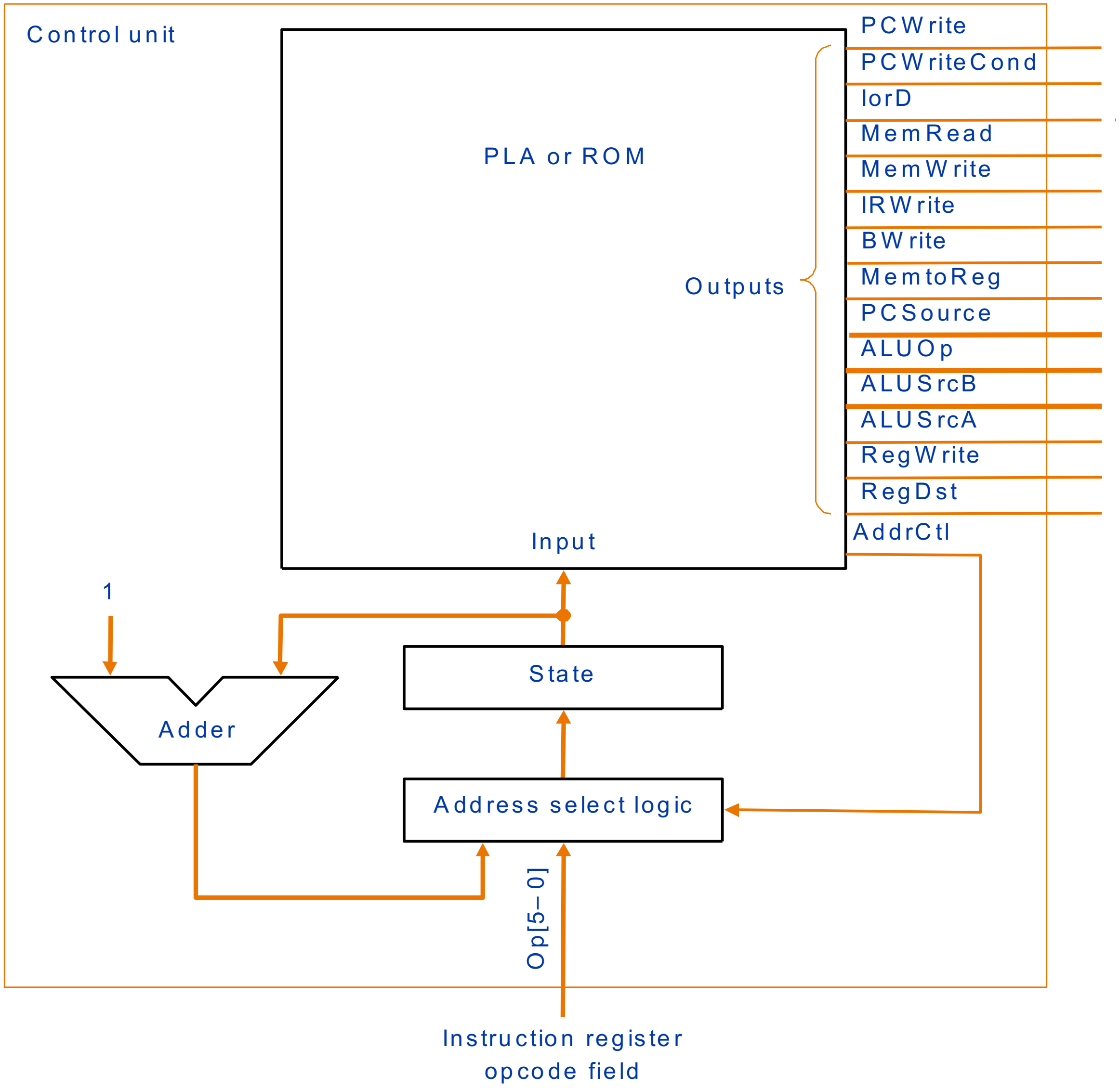
ROM vs PLA

- Break up the table into two parts
  - — 4 state bits tell you the 16 outputs, 2^4 x 16 bits of ROM
  - — 10 bits tell you the 4 next state bits, 2^{10} x 4 bits of ROM
  - — Total: 4.3K bits of ROM
- PLA is much smaller
  - — can share product terms
  - — only need entries that produce an active output
  - — can take into account don't cares
- Size is (#inputs ´ #product-terms) + (#outputs ´ #product-terms)

  For this example = (10x17)+(20x17) = 510 PLA cells
- PLA cells usually about the size of a ROM cell (slightly bigger)

Another Implementation Style

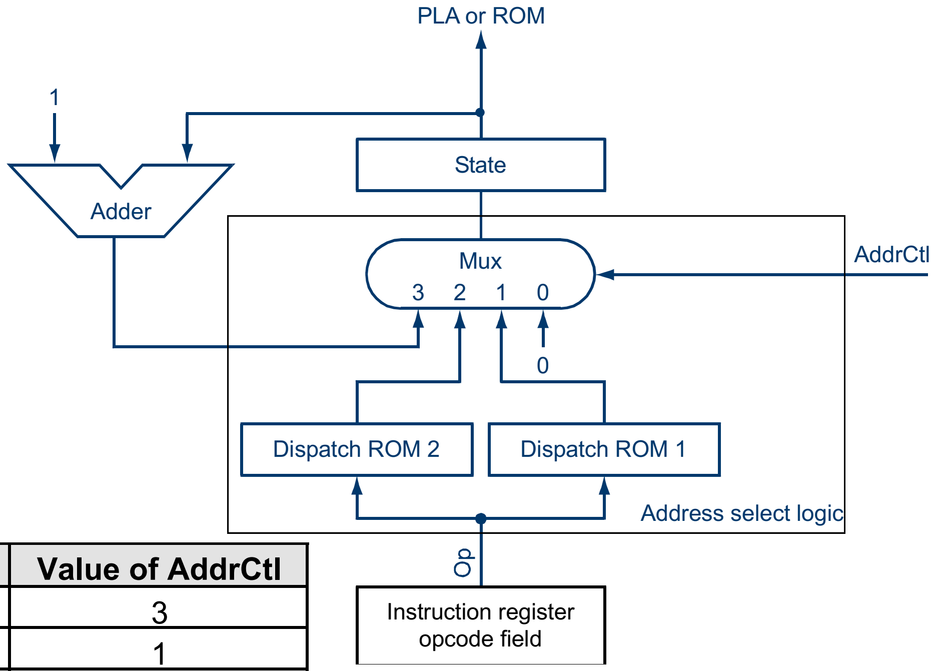
- Complex instructions: the "next state" is often current state + 1



Details

Dispatch ROM 1		
Op	**Opcode name**	**Value**
000000	R-format	0110
000010	jmp	1001
000100	beq	1000
100011	lw	0010
101011	sw	0010

Dispatch ROM 2		
Op	**Opcode name**	**Value**
100011	lw	0011
101011	sw	0101



State number	Address-control action	Value of AddrCtl
0	Use incremented state	3
1	Use dispatch ROM 1	1
2	Use dispatch ROM 2	2
3	Use incremented state	3
4	Replace state number by 0	0
5	Replace state number by 0	0
6	Use incremented state	3
7	Replace state number by 0	0
8	Replace state number by 0	0
9	Replace state number by 0	0

ISA to Microarchitecture Mapping

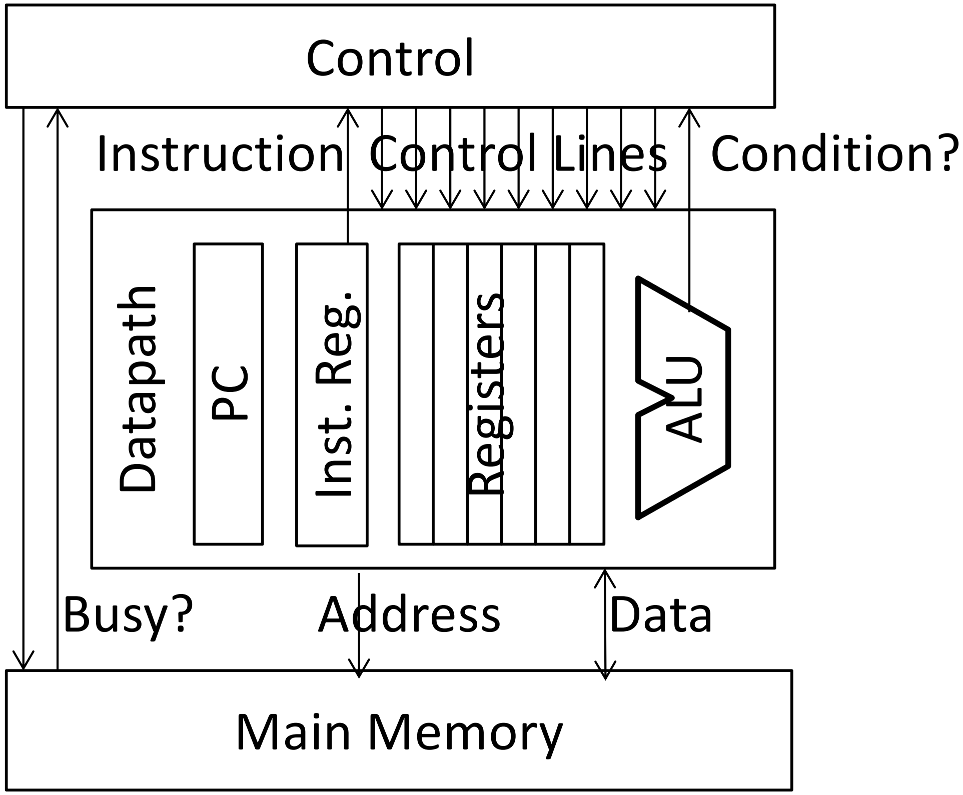
- ISA often designed with particular microarchitectural style in mind, e.g.,

  Accumulator ⇒ hardwired, unpipelined
  CISC ⇒ microcoded
  RISC ⇒ hardwired, pipelined
  VLIW ⇒ fixed-latency in-order parallel pipelines
  JVM ⇒ software interpretation

- But can be implemented with any microarchitectural style
  - Intel Ivy Bridge: hardwired pipelined CISC (x86) machine (with some microcode support)
  - Spike: Software-interpreted RISC-V machine
  - ARM Jazelle: A hardware JVM processor

Control versus Datapath

- As we already know, processor designs are split between *datapath*, where operations computed, and *control*, which sequences operations on datapath

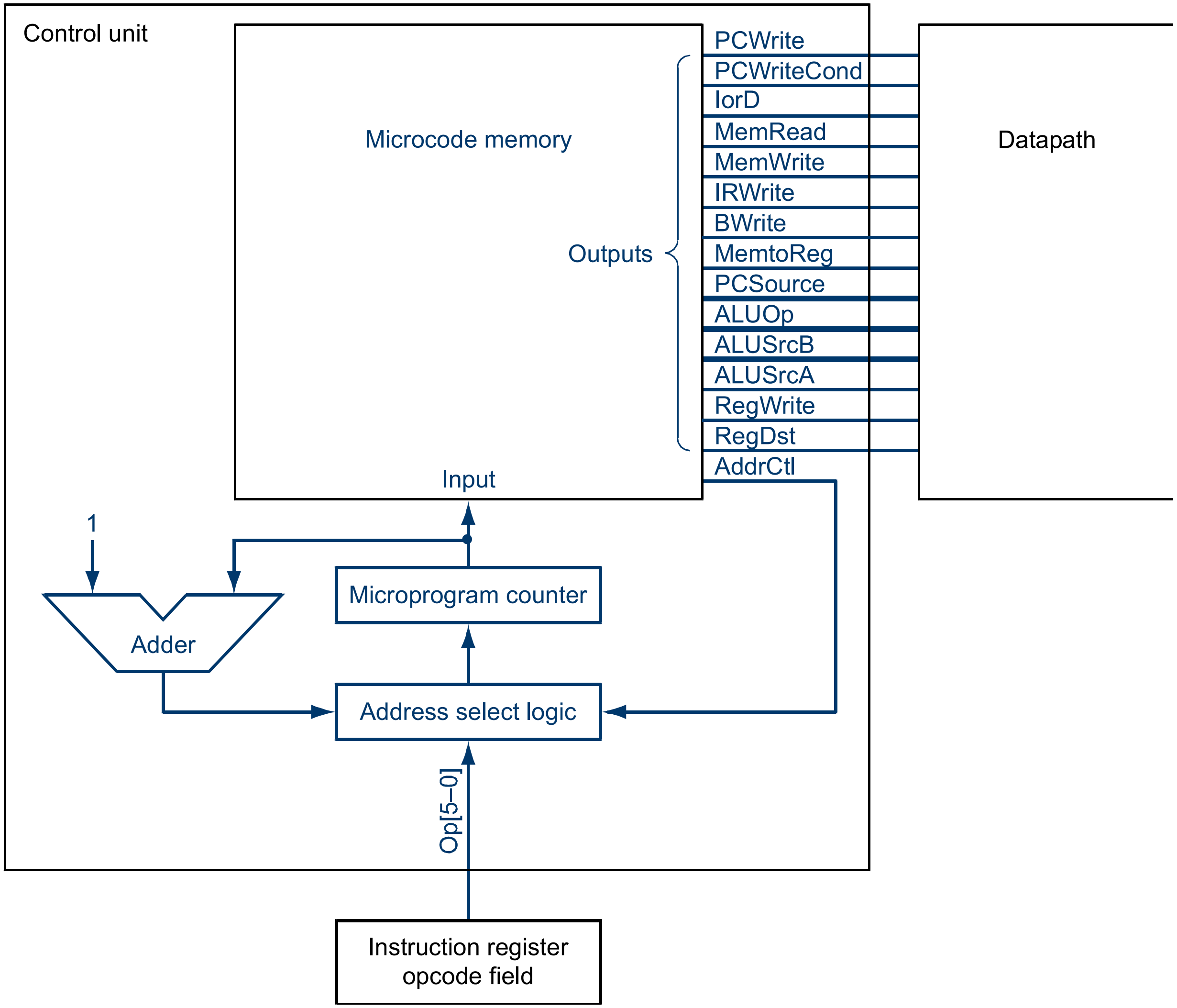


- Biggest challenge for early computer designers was getting control circuitry correct
- Maurice Wilkes invented the idea of microprogramming to design the control unit of a processor for EDSAC-II, 1958

Why Learn Microprogramming?

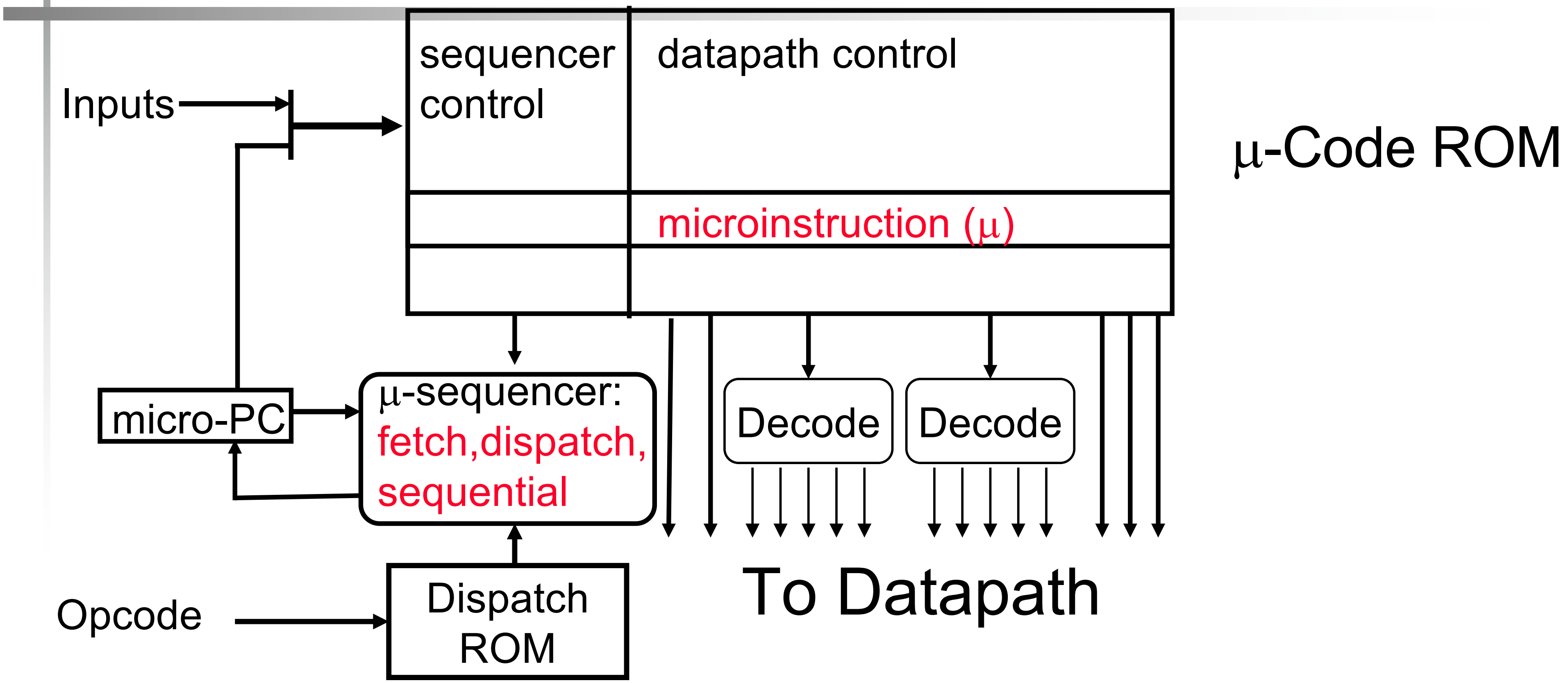
- To show how to build very small processors with complex ISAs
- To help you understand where CISC machines came from
- Because still used in common machines (x86, IBM360, PowerPC)
- As a gentle introduction into machine structures
- To help understand how technology drove the move to RISC

Microprogramming



- What are the “microinstructions” ?

Microprogramming



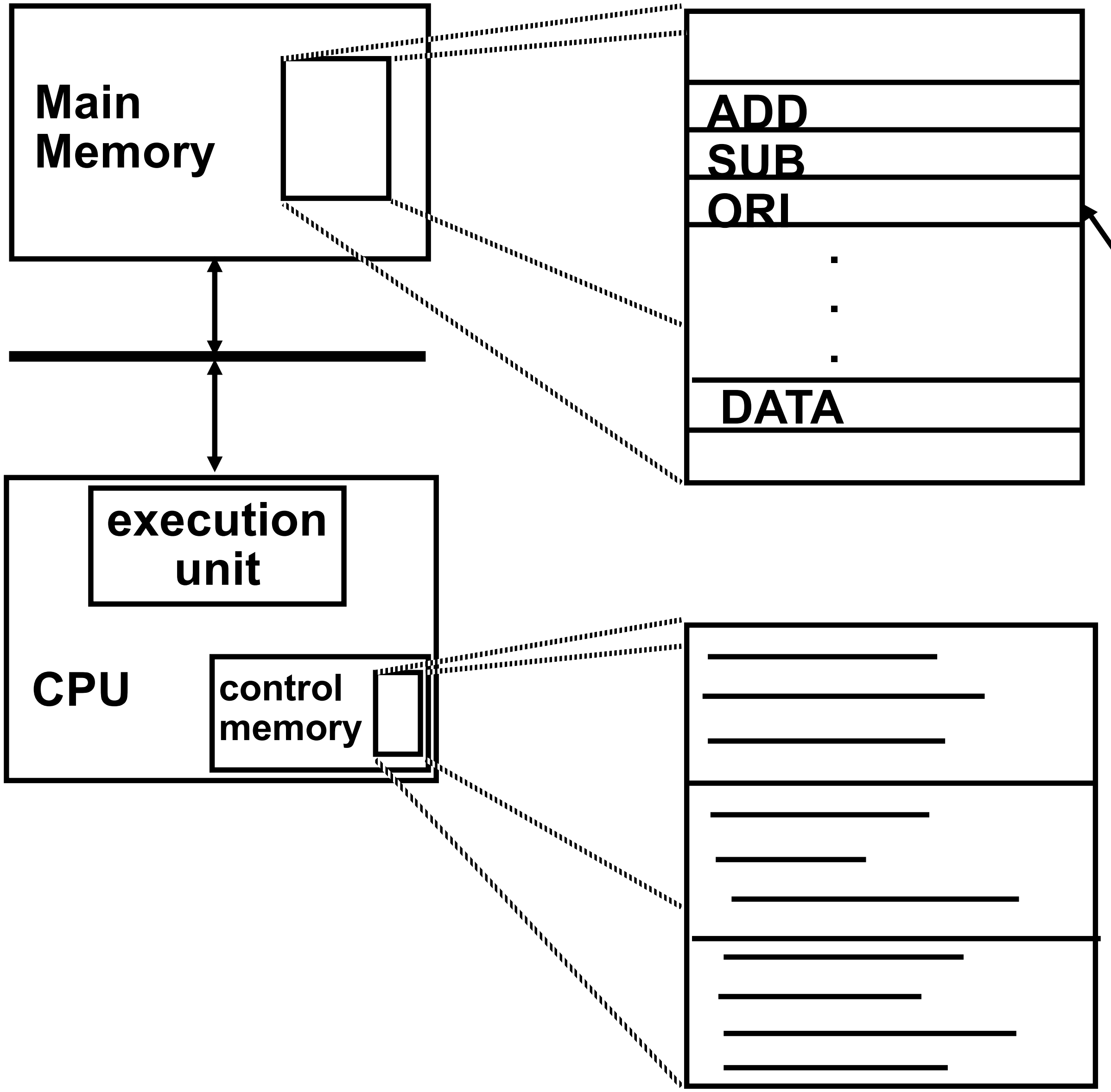
° **Microprogramming is a fundamental concept**

- **implement an instruction set by building a very simple processor and interpreting the instructions**
- **essential for very complex instructions and when few register transfers are possible**
- **overkill when ISA matches datapath 1-1**

Microprogramming

- Microprogramming is a convenient method for implementing *structured* control state diagrams:
  - Random logic replaced by microPC sequencer and ROM
  - Each line of ROM called a μinstruction:
    contains sequencer control + values for control points
  - limited state transitions:
    branch to zero, next sequential,
    branch to μinstruction address from displatch ROM
- Horizontal *μ*Code: one control bit in *μ*Instruction for every control line in datapath
- Vertical *μ*Code: groups of control-lines coded together in *μ*Instruction (e.g. possible ALU dest)
- Control design reduces to Microprogramming
  - Part of the design process is to develop a "language" that describes control and is easy for humans to understand

“Macroinstruction” Interpretation



User program
plus Data

this can change!

one of these is
mapped into one
of these

ORI microsequence

e.g., Fetch Instruction
Fetch Operand(s)
Calculate OR
Save result

Designing a Microinstruction Set

1) Start with list of control signals

2) Group signals together that make sense (vs. random): called "fields"

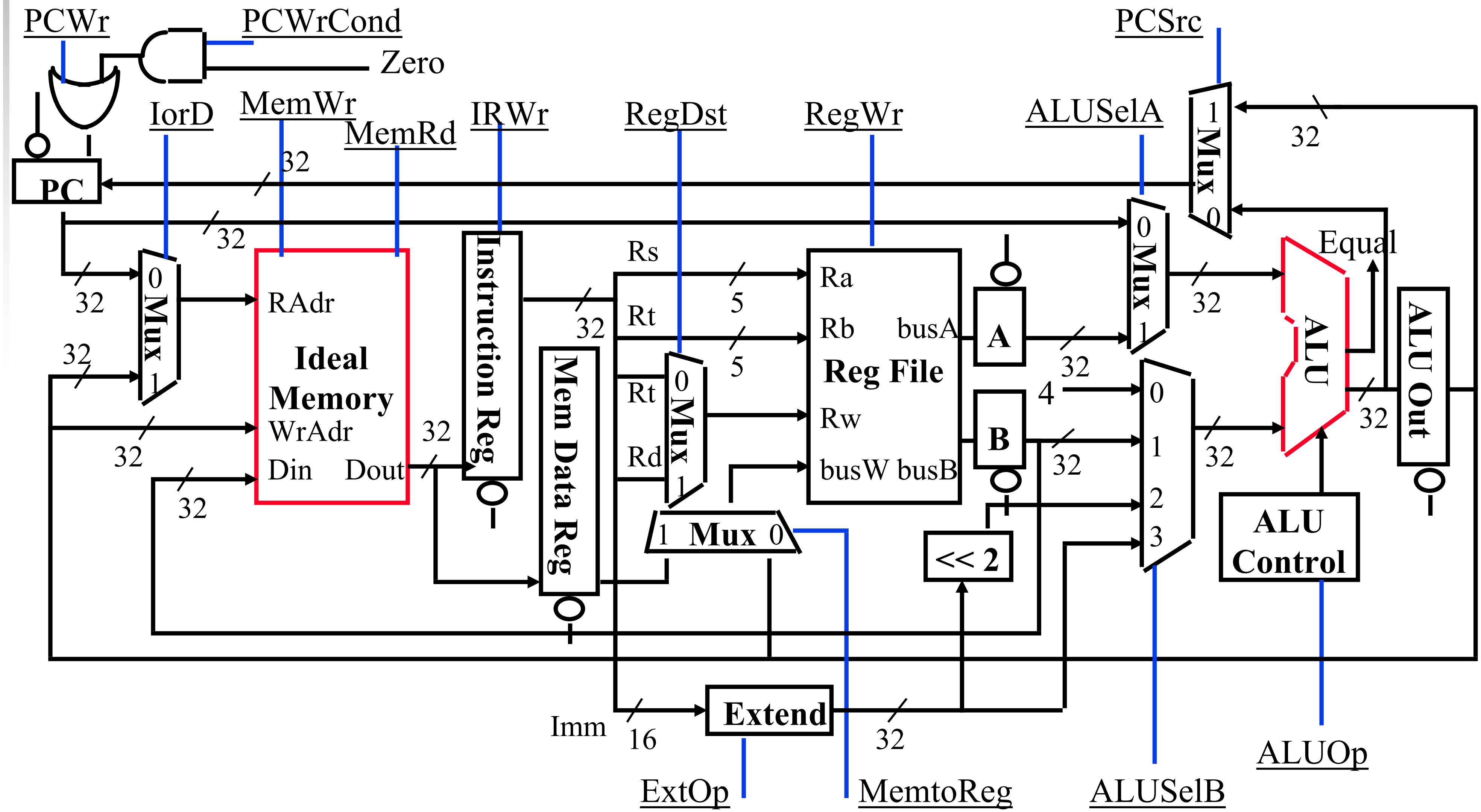
3) Place fields in some logical order (e.g., ALU operation & ALU operands first and microinstruction sequencing last)

4) To minimize the width, encode operations that will never be used at the same time

5) Create a symbolic legend for the microinstruction format, showing name of field values and how they set the control signals

- Use computers to design computers

Multicycle datapath (with ORI but w/o Jump)



Step 1⇒ Start with List of control signals

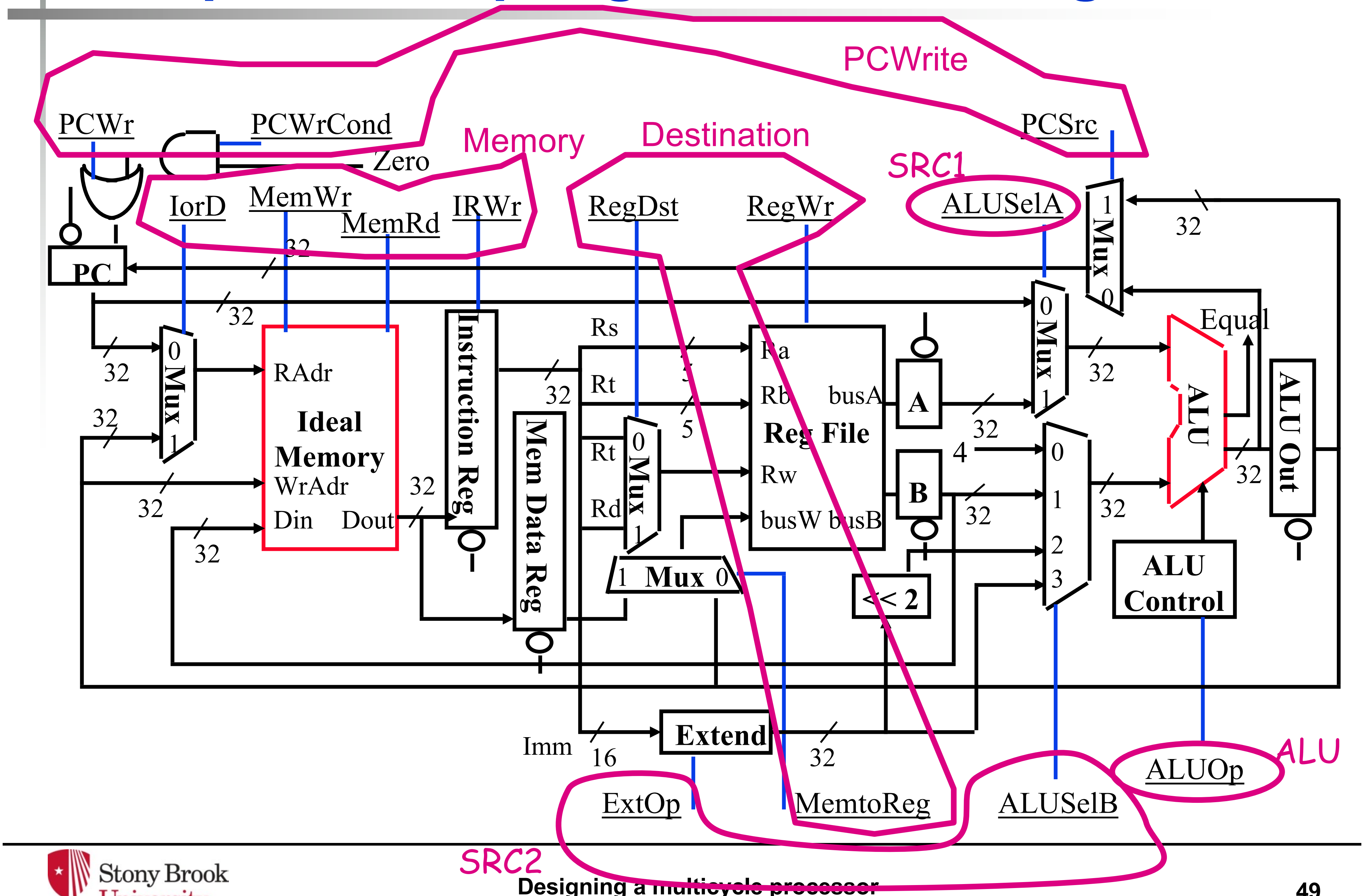
Single Bit Control

Signal name	Effect when deasserted	Effect when asserted
ALUSelA	1st ALU operand = PC	1st ALU operand = Reg[rs]
RegWrite	None	Reg. is written
MemtoReg	Reg. write data input = ALU	Reg. write data input = memory
RegDst	Reg. dest. no. = rt	Reg. dest. no. = rd
MemRead	None	Memory at address is read, MDR <= Mem[addr]
MemWrite	None	Memory at address is written
IorD	Memory address = PC	Memory address = S
IRWrite	None	IR <= Memory
PCWrite	None	PC <= PCSource
PCWriteCond	None	IF ALUzero then PC <= PCSource
PCSource	PCSource = ALU	PCSource = ALUout
ExtOp	Zero Extended	Sign Extended

Multiple Bit Control

Signal name	Value	Effect
ALUOp	00	ALU adds
	01	ALU subtracts
	10	ALU does function code
	11	ALU does logical OR
ALUSelB	00	2nd ALU input = 4
	01	2nd ALU input = Reg[rt]
	10	2nd ALU input = extended,shift left 2
	11	2nd ALU input = extended

Step 2⇒Group together related signals



3&4) Microinstruction Format: unencoded vs. encoded fields

Field Name	***Width***		***Control Signals Set***
	wide		**narrow**
ALU Control	**4**	**2**	**ALUOp**
SRC1	**2**	**1**	**ALUSelA**
SRC2	**5**	**3**	**ALUSelB, ExtOp**
ALU Destination	**3**	**2**	**RegWrite, MemtoReg, RegDst**
Memory	**3**	**2**	**MemRead, MemWrite, IorD**
Memory->Register	**1**	**1**	**IRWrite**
PCWrite Control	**3**	**2**	**PCWrite, PCWriteCond, PCSource**
Sequencing	**3**	**2**	**AddrCtl**
Total width	**24**	**15**	**bits**

Step 5⇒Group into Fields, Order and Assign Names

ALU	SRC1	SRC2	Dest	Mem	Memreg	Pcwrite	Seq

Field Name	*Values for Field*	*Function of Field with Specific Value*
ALU	Add	ALU adds
	Subt.	ALU subtracts
	Func	ALU does function code
	Or	ALU does logical OR
SRC1	PC	1st ALU input <= PC
	rs	1st ALU input <= Reg[rs]
SRC2	4	2nd ALU input <= 4
	Extend	2nd ALU input <= sign ext. IR[15-0]
	Extend0	2nd ALU input <= zero ext. IR[15-0]
	Extshft	2nd ALU input <= sign ex., sl IR[15-0]
	rt	2nd ALU input <= Reg[rt]
Dest(ination)	rd ALU	Reg[rd] <= ALUout
	rt ALU	Reg[rt] <= ALUout
	rt Mem	Reg[rt] <= Mem
Mem(ory)	Read PC	Read memory using PC; IR <= Mem [PC]
	Read ALU	Read memory using ALUout for addr
	Write ALU	Write memory using ALUout for addr
PCwrite	ALU	PC <= ALU
	ALUout-cond	IF Zero then PC <= ALUout
Seq(uencing)	Seq	Go to next sequential µinstruction
	Fetch	Go to the first microinstruction
	Dispatch 1	Dispatch using ROM1
	Dispatch 2	Dispatch using ROM2

Microinstruction Format

Field name	Value	Signals active	Comment
ALU control	Add	ALUOp = 00	Cause the ALU to add.
	Subt	ALUOp = 01	Cause the ALU to subtract; this implements the compare for branches.
	Func code	ALUOp = 10	Use the instruction's function code to determine ALU control.
SRC1	PC	ALUSrcA = 0	Use the PC as the first ALU input.
	A	ALUSrcA = 1	Register A is the first ALU input.
SRC2	B	ALUSrcB = 00	Register B is the second ALU input.
	4	ALUSrcB = 01	Use 4 as the second ALU input.
	Extend	ALUSrcB = 10	Use output of the sign extension unit as the second ALU input.
	Extshft	ALUSrcB = 11	Use the output of the shift-by-two unit as the second ALU input.
Register control	Read		Read two registers using the rs and rt fields of the IR as the register numbers and putting the data into registers A and B.

	Write ALU	RegWrite,
		RegDst = 1,
		MemtoReg = 0
			Write a register using the rd field of the IR as the register number and the contents of the ALUOut as the data.

	Write MDR	RegWrite,
		RegDst = 0,
		MemtoReg = 1
			Write a register using the rt field of the IR as the register number and the contents of the MDR as the data.

Memory	Read PC	MemRead, IorD = 0	Read memory using the PC as address; write result into IR (and the MDR).
	Read ALU	MemRead, IorD = 1	Read memory using the ALUOut as address; write result into MDR.
	Write ALU	MemWrite, IorD = 1	Write memory using the ALUOut as address, contents of B as the data.
PC write control	ALU	PCSource = 00 PCWrite	Write the output of the ALU into the PC.
	ALUOut-cond	PCSource = 01, PCWriteCond	If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.
	jump address	PCSource = 10, PCWrite	Write the PC with the jump address from the instruction.
Sequencing	Seq	AddrCtl = 11	Choose the next microinstruction sequentially.
	Fetch	AddrCtl = 00	Go to the first microinstruction to begin a new instruction.
	Dispatch 1	AddrCtl = 01	Dispatch using the ROM 1.
	Dispatch 2	AddrCtl = 10	Dispatch using the ROM 2.

Microprogramming

- A specification methodology
  - appropriate if hundreds of opcodes, modes, cycles, etc.
  - signals specified symbolically using microinstructions

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch
BEQ1	Subt	A	B			ALUOut-cond	Fetch
JUMP1						Jump address	Fetch

- *Will two implementations of the same architecture have the same microcode?*
- *What would a microassembler do?*

Maximally vs. Minimally Encoded

- No encoding:
  - 1 bit for each datapath operation
  - faster, requires more memory (logic)
  - used for Vax 780 — an astonishing 400K of memory!
- Lots of encoding:
  - send the microinstructions through logic to get control signals
  - uses less memory, slower
- Historical context of CISC:
  - Too much logic to put on a single chip with everything else
  - Use a ROM (or even RAM) to hold the microcode
  - It’s easy to add new instructions

Microcode: Trade-offs

- Distinction between specification and implementation is sometimes blurred
- Specification Advantages:
  - Easy to design and write
  - Design architecture and microcode in parallel
- Implementation (off-chip ROM) Advantages
  - Easy to change since values are in memory
  - Can emulate other architectures
  - Can make use of internal registers
- Implementation Disadvantages, SLOWER now that:
  - Control is implemented on same chip as processor
  - ROM is no longer faster than RAM
  - No need to go back and make changes

Technology Influence

- When microcode appeared in 50s, different technologies for:
  - Logic: Vacuum Tubes
  - Main Memory: Magnetic cores
  - Read-Only Memory: Diode matrix, punched metal cards, …
- Logic very expensive compared to ROM or RAM
- ROM cheaper than RAM
- ROM much faster than RAM

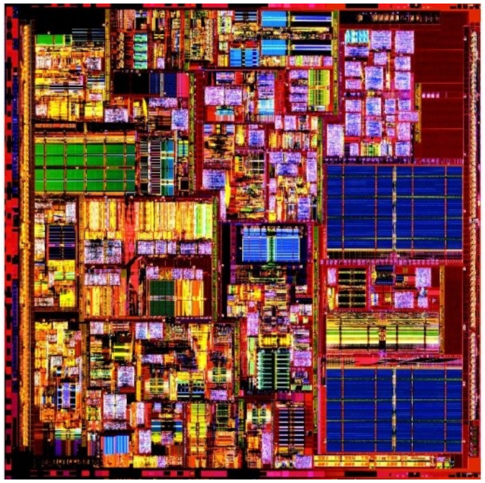
Historical Perspective

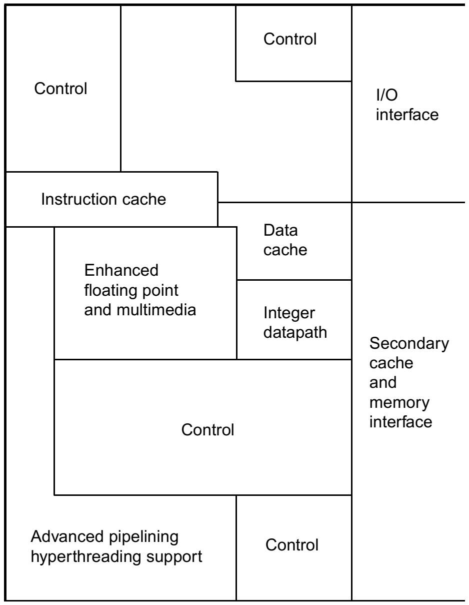
- In the ‘60s and ‘70s microprogramming was very important for implementing machines
- This led to more sophisticated ISAs and the VAX
- In the ‘80s RISC processors based on pipelining became popular
- Pipelining the microinstructions is also possible!
- Implementations of IA-32 architecture processors since 486 use:
  - “hardwired control” for simpler instructions
    (few cycles, FSM control implemented using PLA or random logic)
  - “microcoded control” for more complex instructions
    (large numbers of cycles, central control store)

- The IA-64 architecture uses a RISC-style ISA and can be implemented without a large central control store

Pentium 4

- Somewhere in all that “control we must handle complex instructions




Control
Control
I/O interface
Instruction cache
Data cache
Enhanced floating point and multimedia
Integer datapath
Secondary cache and memory interface
Control
Advanced pipelining hyperthreading support
Control

- Processor executes simple microinstructions, 70 bits wide (hardwired)
- 120 control lines for integer datapath (400 for floating point)
- If an instruction requires more than 4 microinstructions to implement, control from microcode ROM (8000 microinstructions)
- Its complicated!

Microprogramming is far from extinct

- Played a crucial role in micros of the Eighties
    - DEC uVAX, Motorola 68K series, Intel 286/386
- Plays an assisting role in most modern micros
    - e.g., AMD Bulldozer, Intel Ivy Bridge, Intel Atom, IBM PowerPC, …
    - Most instructions executed directly, i.e., with hard-wired control
    - Infrequently-used and/or complicated instructions invoke microcode

- Patchable microcode common for post-fabrication bug fixes, e.g., Intel processors load µcode patches at bootup

Reconsidering Microcode Machine

RISC!

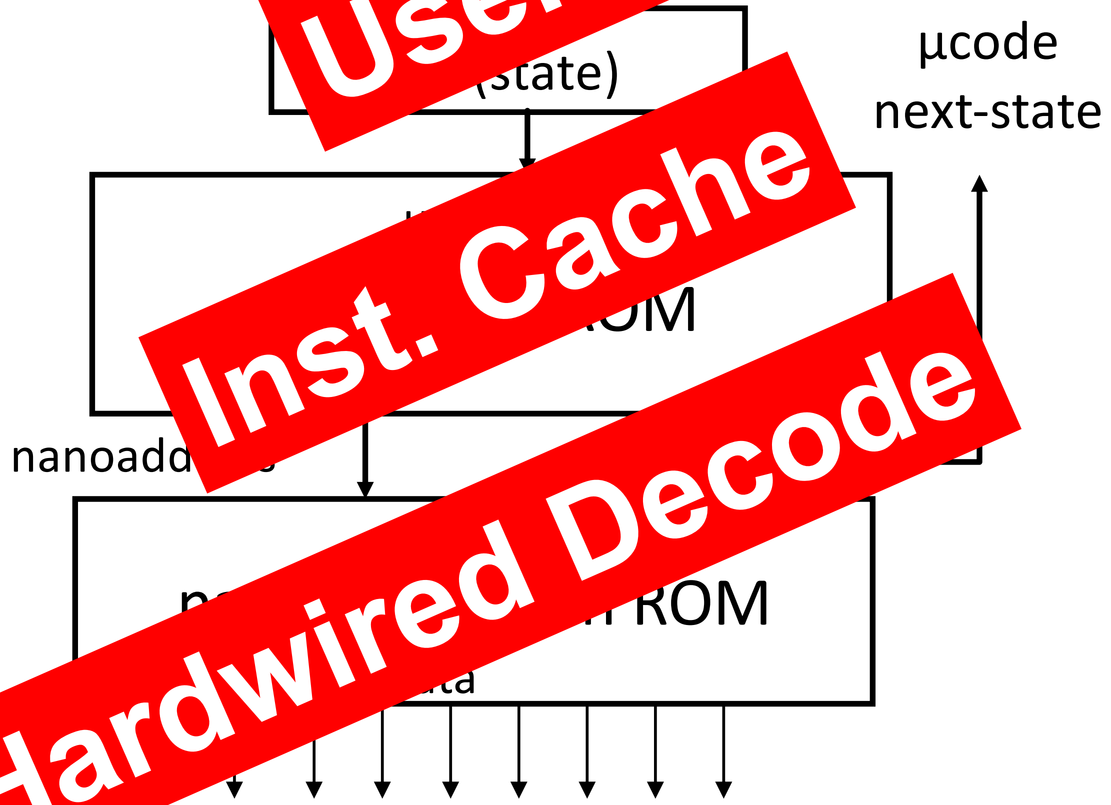
Exploits recurring control signal patterns in µcode, e.g.,

ALU0 A ⍰ Reg[rs1]

...

ALUI0 A ⍰ Reg[rs1]

...



User PC

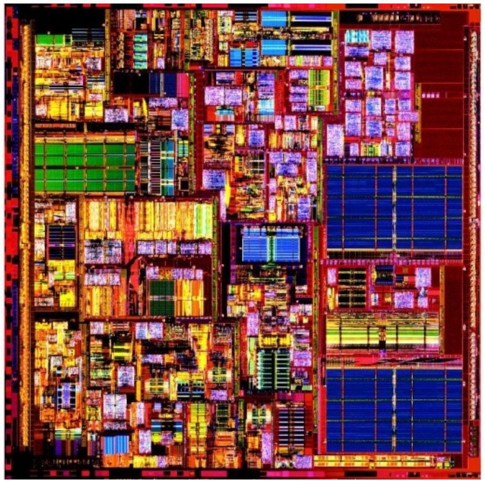
Inst. Cache

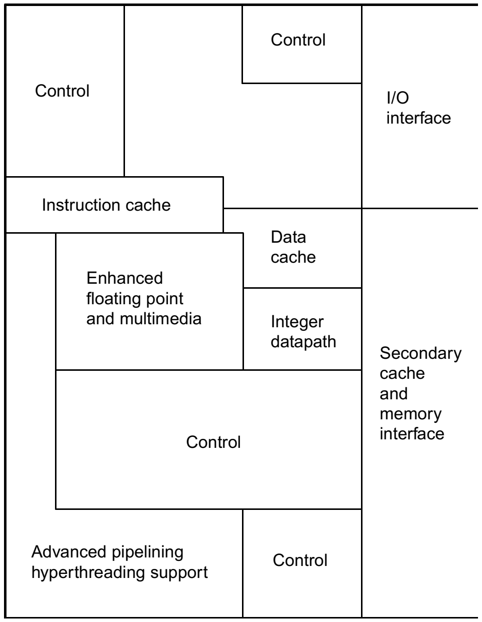
Hardwired Decode

- Motorola 68000 had 17-bit µcode containing either 10-bit µjump or 9-bit nanoinstruction pointer
  - Nanoinstructions were 68 bits wide, decoded to give 196 control signals

Pentium 4

- Pipelining is important (last IA-32 without it was 80386 in 1985)



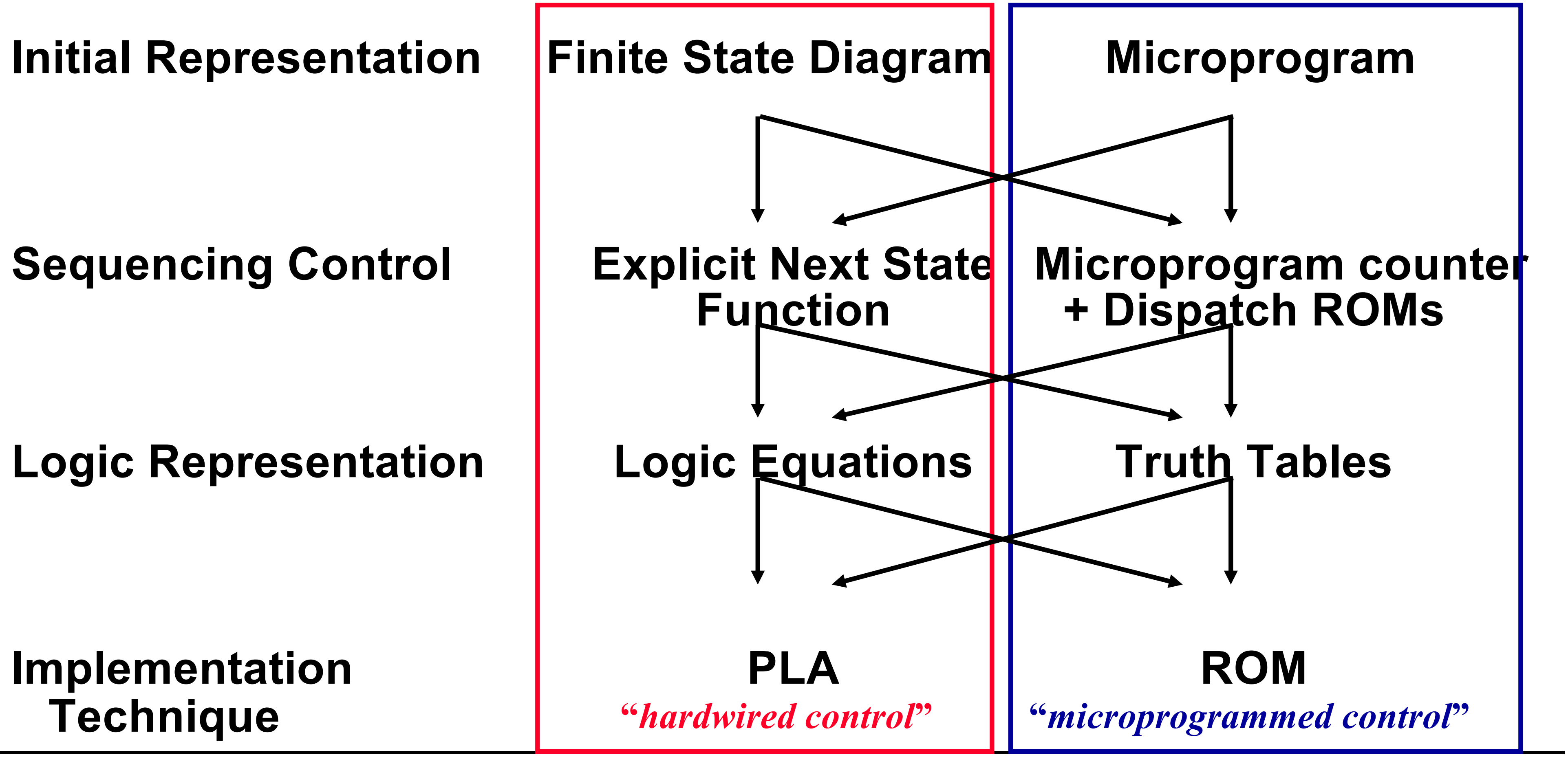

Control
Control
I/O interface
Instruction cache
Data cache
Enhanced floating point and multimedia
Integer datapath
Secondary cache and memory interface
Control
Advanced pipelining hyperthreading support
Control

- Pipelining is used for the simple instructions favored by compilers

  "Simply put, a high performance implementation needs to ensure that the simple instructions execute quickly, and that the burden of the complexities of the instruction set penalize the complex, less frequently used, instructions"

Overview of Control

° Control may be designed using one of several initial representations. The choice of sequence control, and how logic is represented, can then be determined independently; the control can then be implemented with one of several methods using a structured logic technique.



Initial Representation	Finite State Diagram	Microprogram
Sequencing Control	Explicit Next State Function	Microprogram counter + Dispatch ROMs
Logic Representation	Logic Equations	Truth Tables
Implementation Technique	PLA *"hardwired control"*	ROM *"microprogrammed control"*

Summary

- If we understand the instructions…

  We can build a simple processor!

- If instructions take different amounts of time, multi-cycle is better
- Datapath implemented using:
  - Combinational logic for arithmetic
  - State holding elements to remember bits
- Control implemented using:
  - Combinational logic for single-cycle implementation
  - Finite state machine for multi-cycle implementation

Acknowledgements

- These slides contain material developed and copyright by:
  - Morgan Kauffmann (Elsevier, Inc.)
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)