Topic 9: Microprogramming and Exceptions

COS / ELE 375

Computer Architecture and Organization

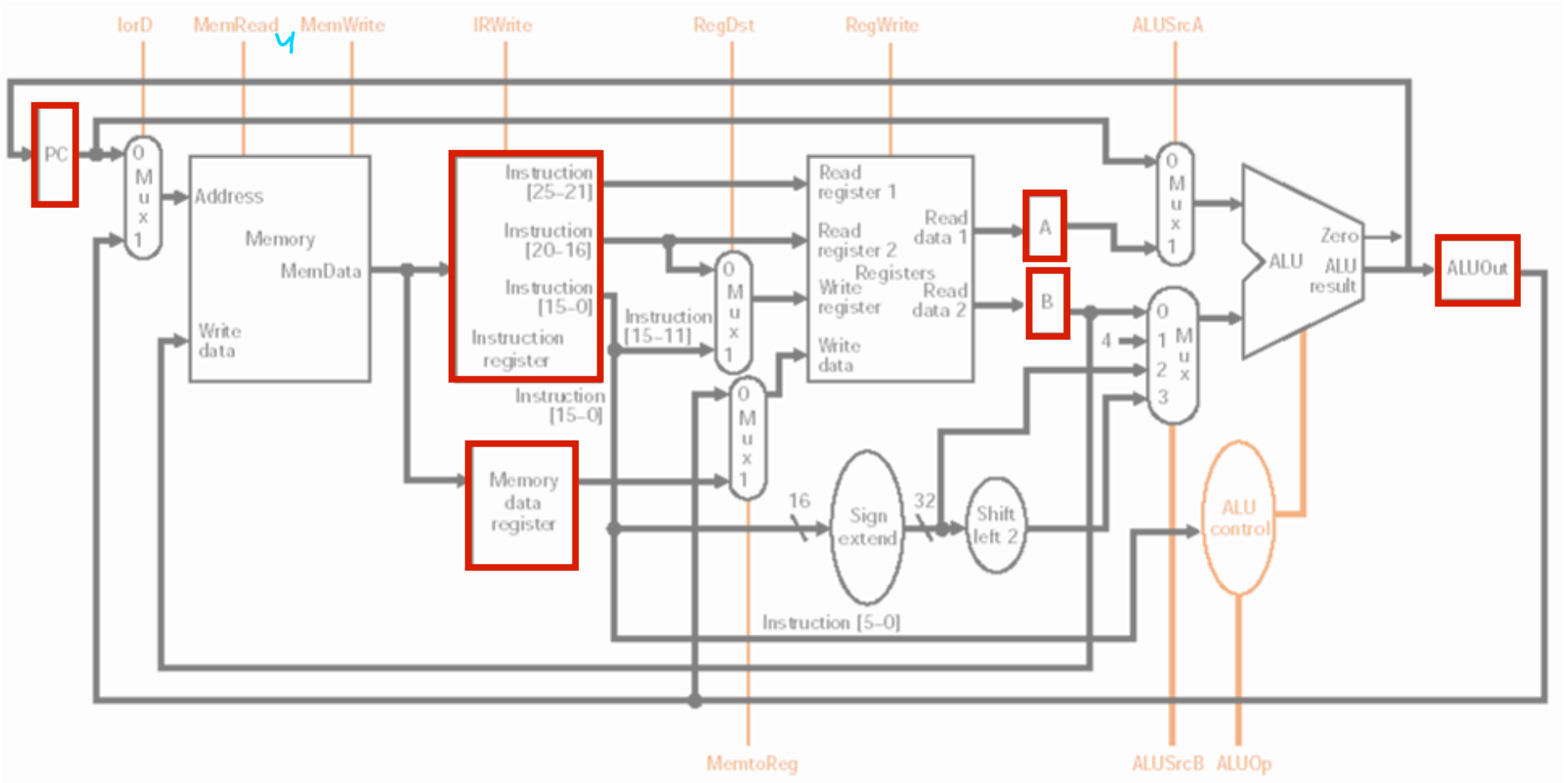
Princeton University
Fall 2015

Prof. David August

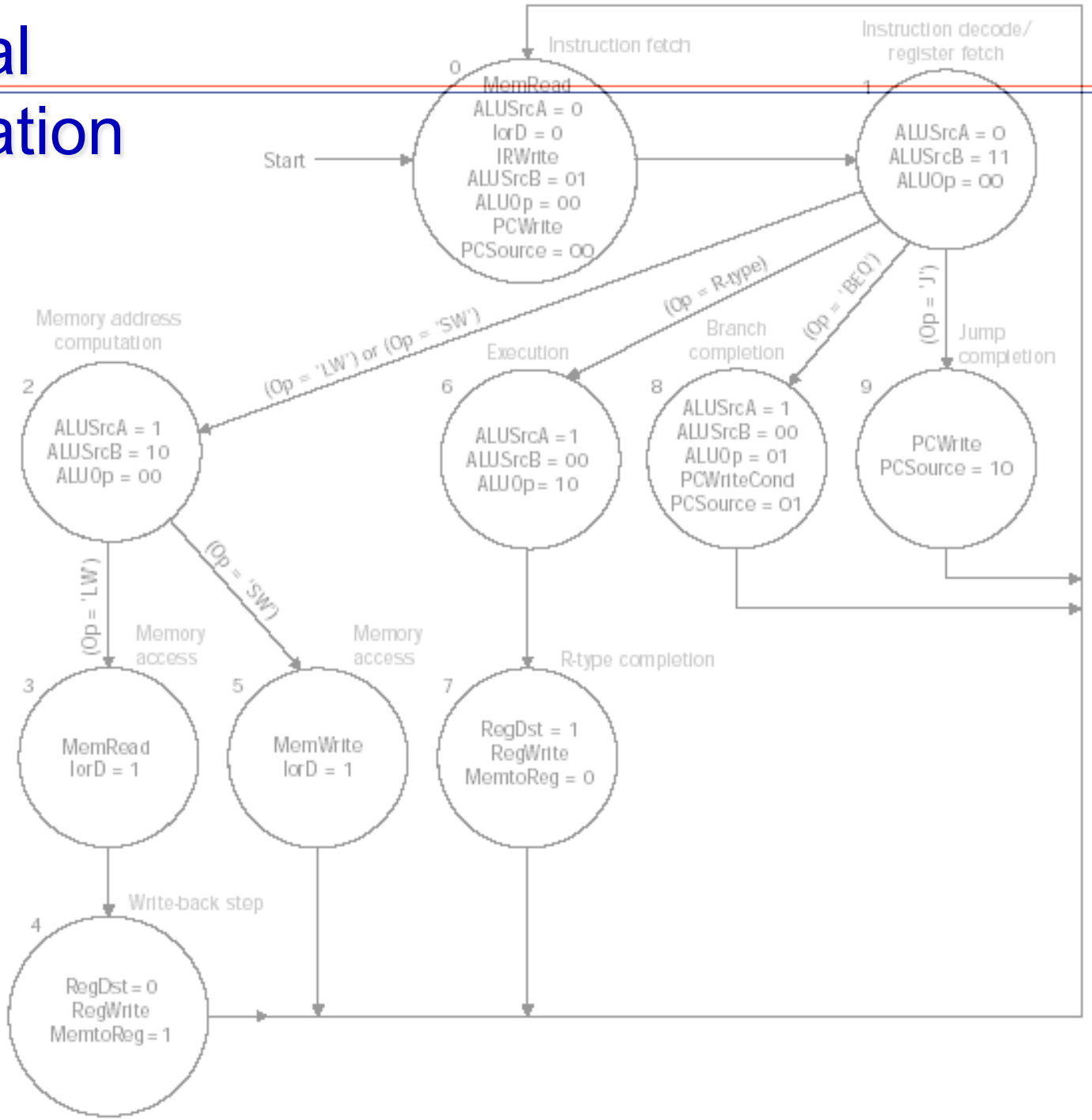
Program Notes

- HW #2 due October 26th at 5PM – No Late Days.
- Project #1 due Novemer 11th at 5PM
- Midterm October 28th In Class
  - Closed book/notes
  - One Two-sided "cheat sheet" allowed

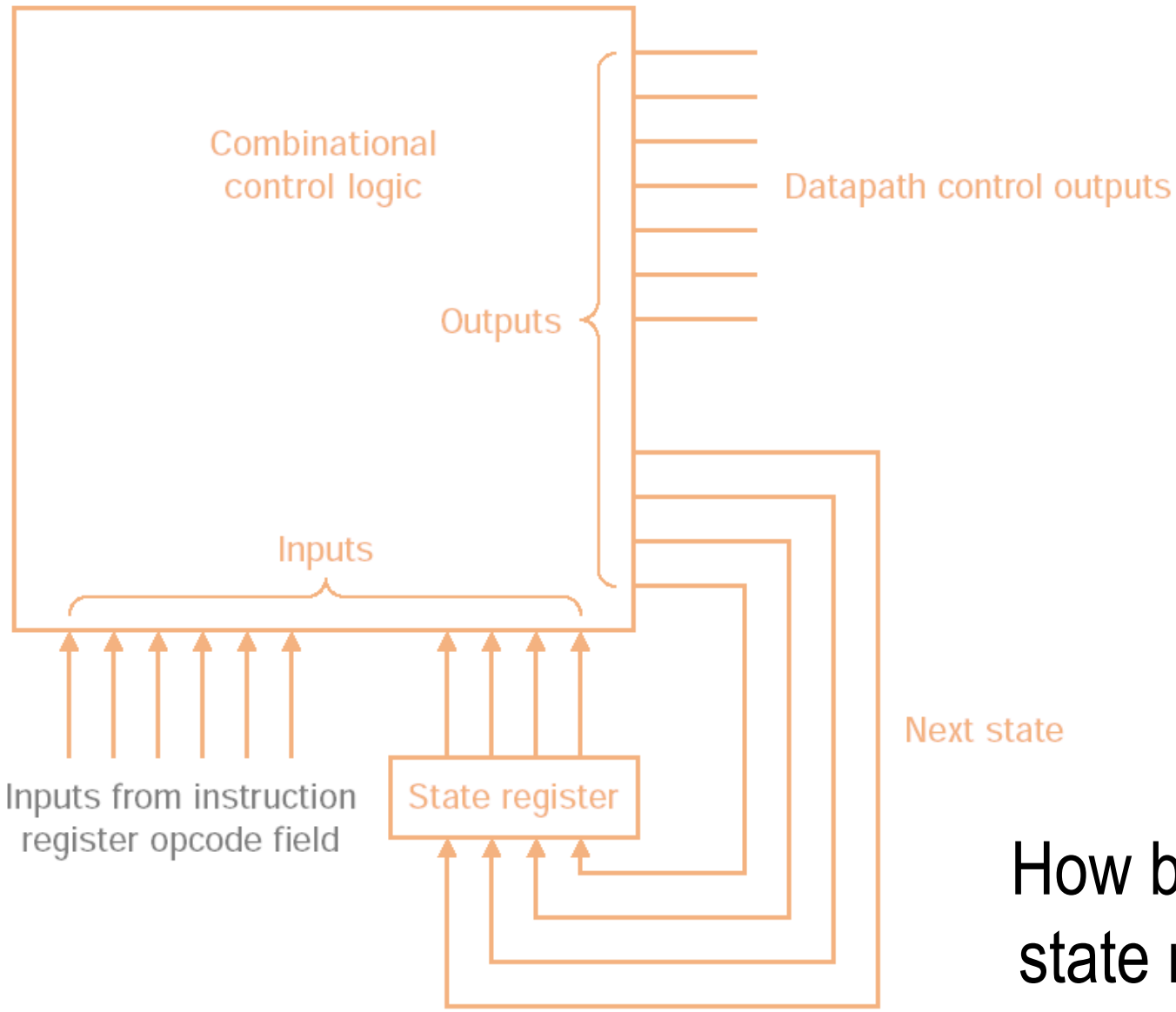
Review


IorD
MemRead
MemWrite
IRWrite
RegDst
RegWrite
ALUSrcA
PC
0
M
u
x
1
Address
Memory
MemData
Write
data
Instruction
[25–21]
Instruction
[20–16]
Instruction
[15–0]
Instruction
register
Instruction
[15–0]
Instruction
[15–11]
Memory
data
register
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
A
B
4
ALU
Zero
ALU
result
ALUOut
16
Sign
extend
32
Shift
left 2
ALU
control
Instruction [5–0]
MemtoReg
ALUSrcB
ALUOp

Graphical Specification of FSM

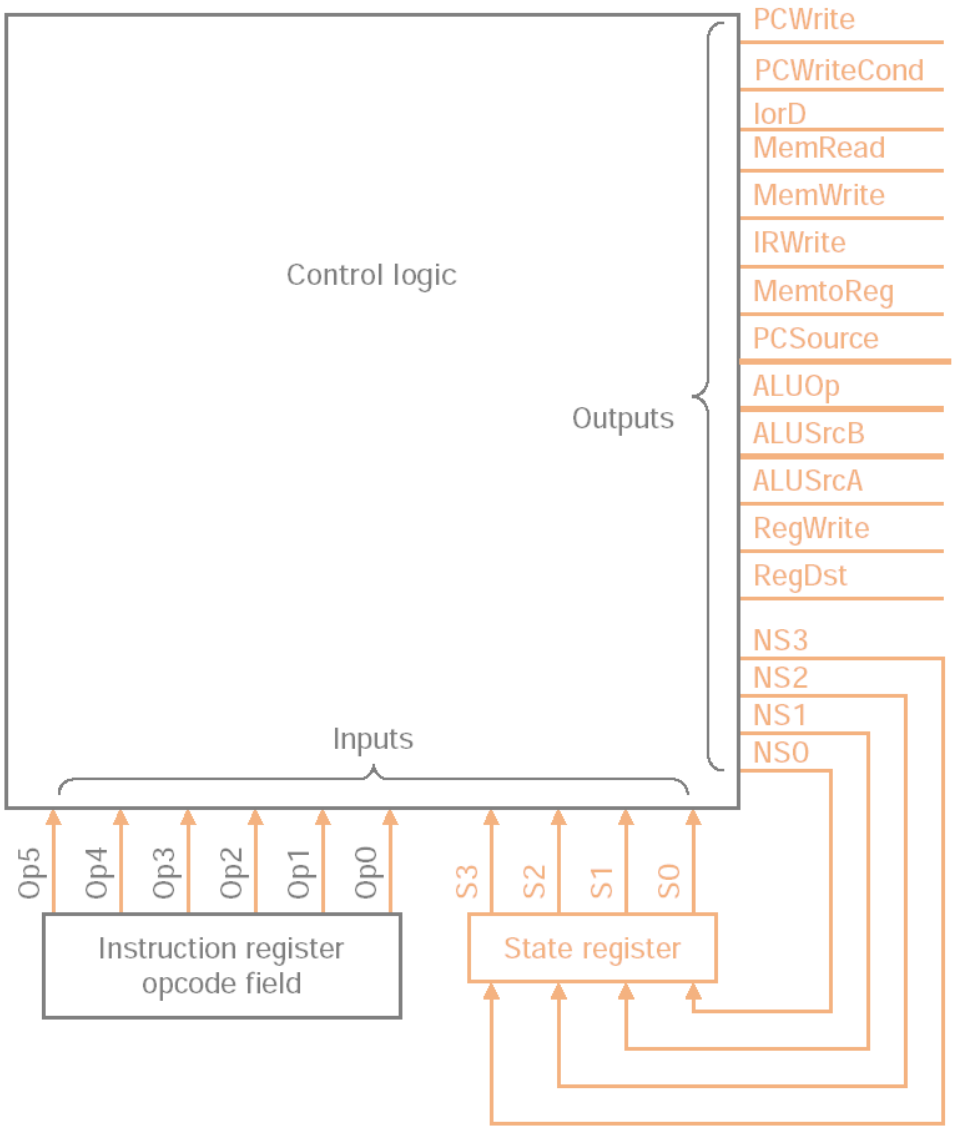


FSM Implementation

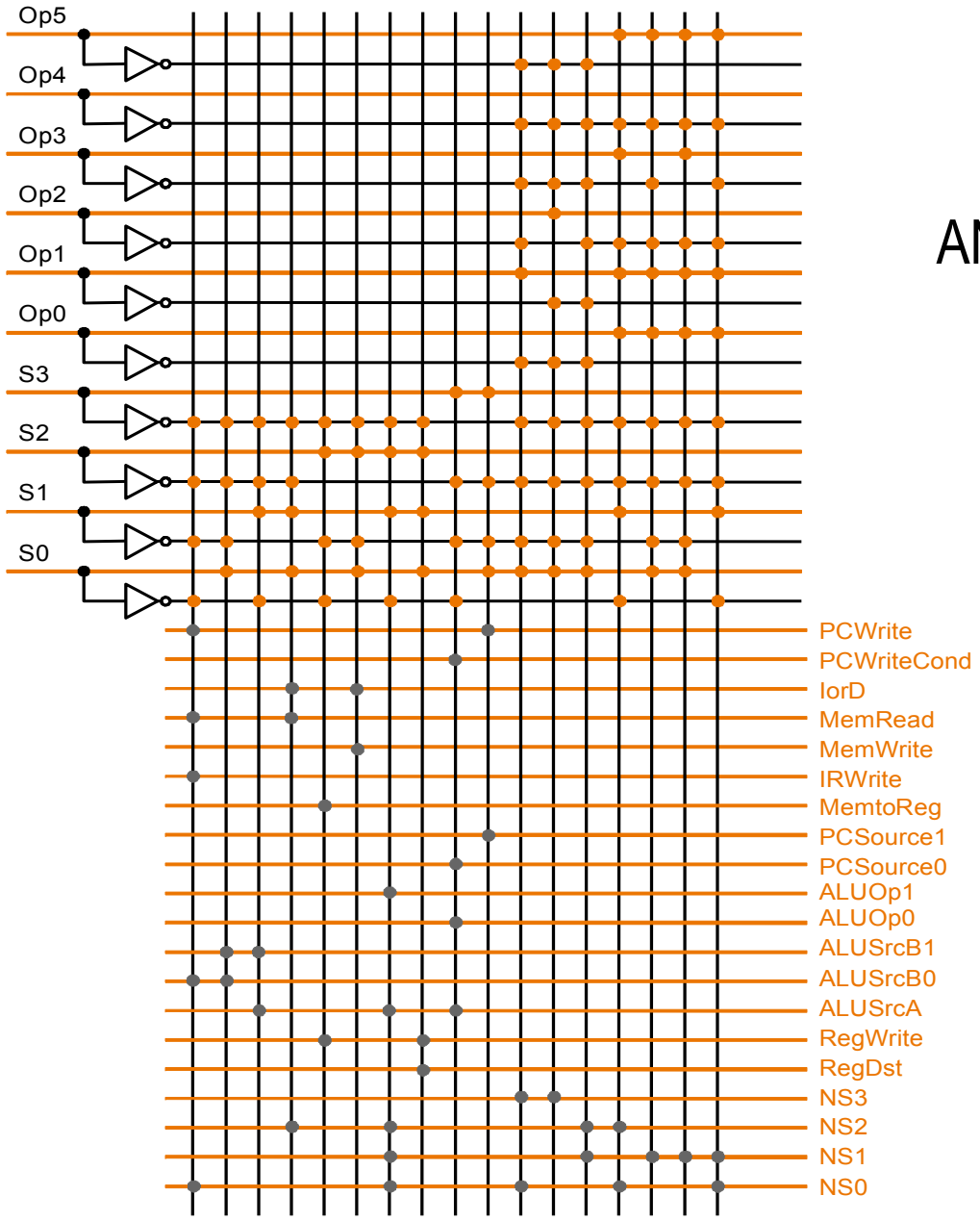


How big must the state register be?

For Our Machine…



PLA Implementation



AND-Plane

OR-Plane

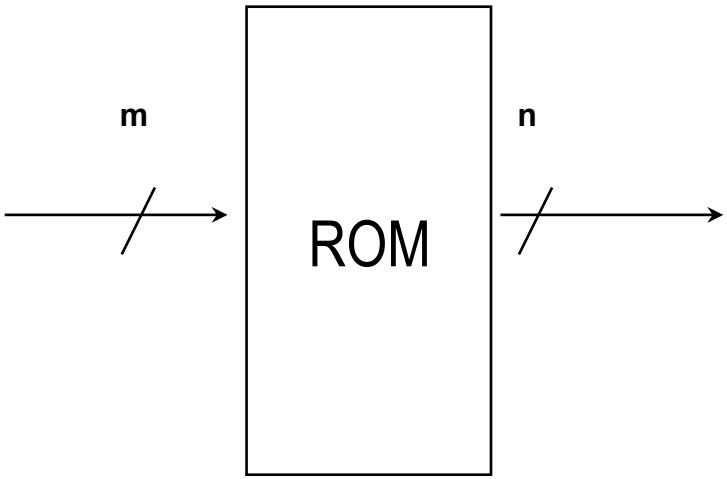
ROM Implementation

ROM = “Read Only Memory” - values fixed ahead of time

A ROM can be used to implement a truth table

- If address is m-bits, 2^m entries in the ROM
- Outputs are the n-bits of data at that address
- Consider m the “height” and n the “width”

n = control wire bits + next state bits
m = opcode bits + current state bits



Addr	Data
00	0101
01	1101
10	1001
11	0001

ROM Implementation for MIPS

- How many inputs are there?
  6 bits for opcode, 4 bits for state = 10 address lines
  (2^{10} = 1024 different addresses)

- How many outputs are there?
  16 datapath-control bits, 4 state bits = 20 bits

- ROM is 210 x 20 = 20K bits (a rather unusual size)

- Wasteful. A full truth table!
  Consider: opcode is often ignored

ROM vs PLA

Factor ROM Table

- 4 bits for the 16 control signals, 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 even smaller!

- can take into account “don't cares”
- can share product terms

Size is:

(#inputs)(#product-terms) + (#outputs)(#product-terms)

For this example = (10 x 17) + (20 x 17) = 460 PLA cells

Full MIPS

Full MIPS: More complex control

- Over 100 instructions
- 1 to 20 clock cycles

X86: Forget about it!!!

With thousands of states, mistakes likely to happen

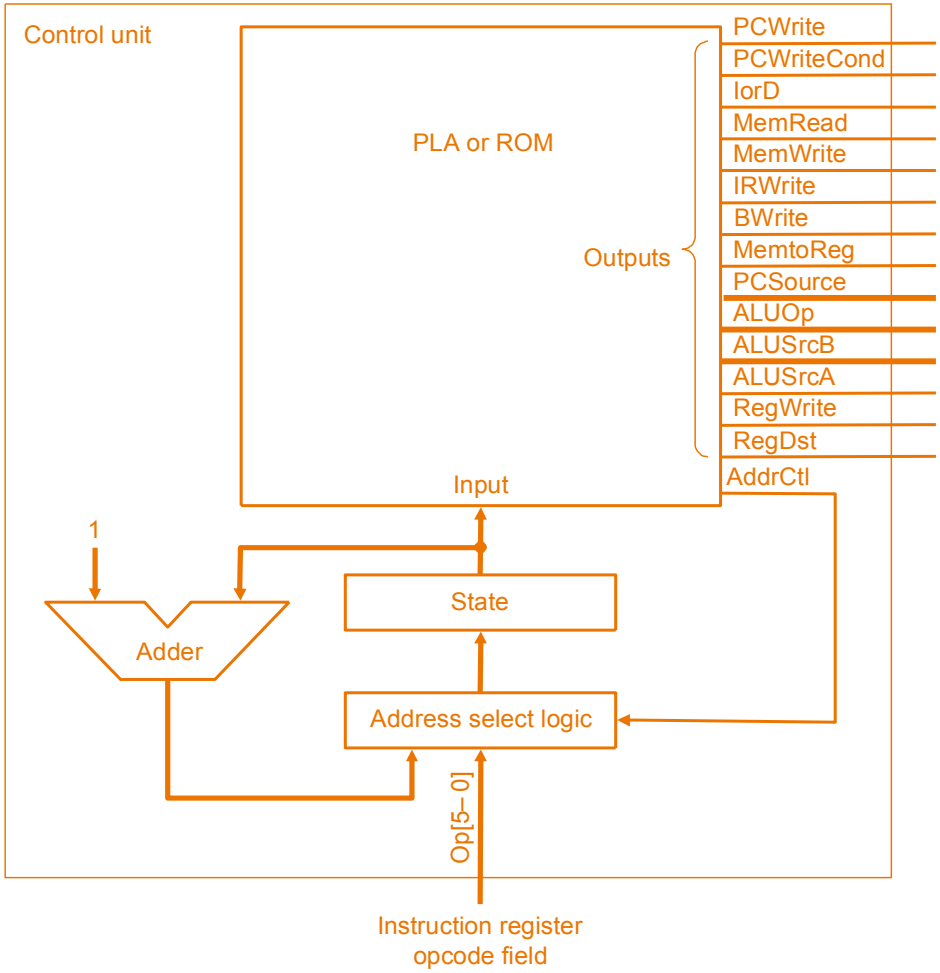
Solution: Treat state machine as a program!

(We're experts on bugs in programs…)

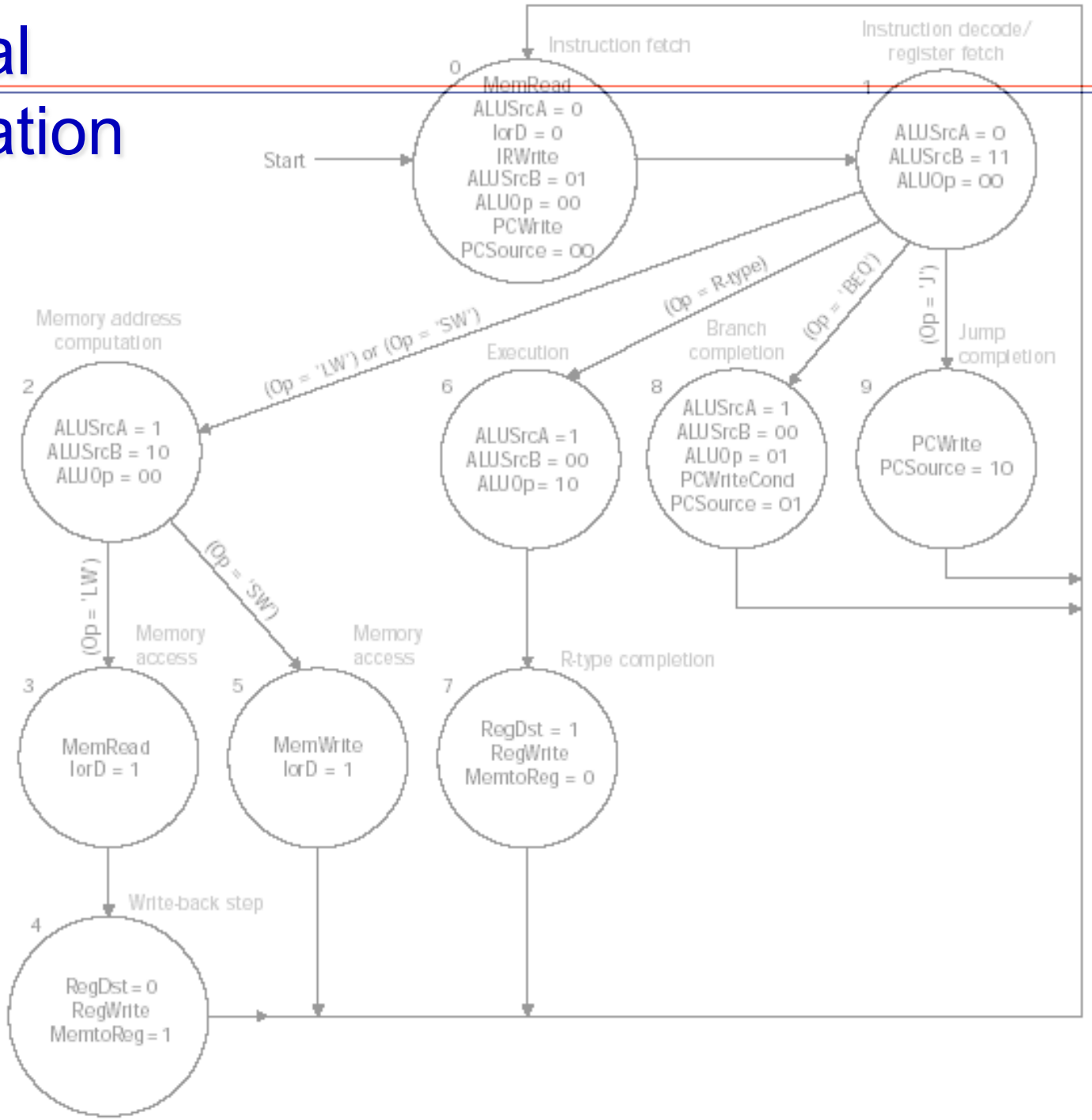
Microprogramming

Another Implementation Style

- The “next state” is often current state + 1
- The “next instruction” is often current instruction + 1



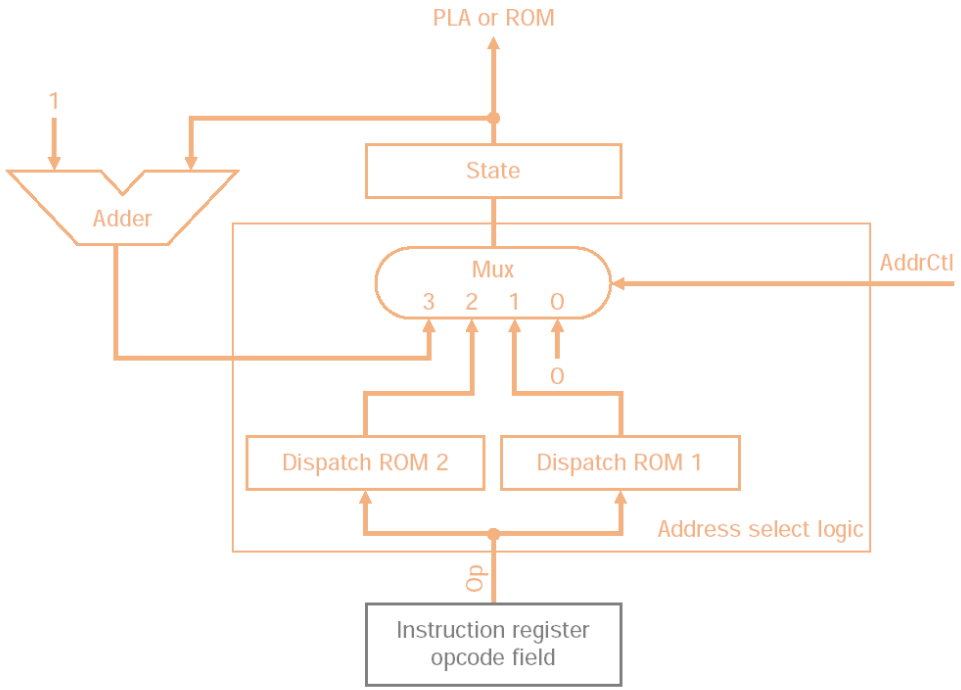
Graphical Specification of FSM



Address Select Logic

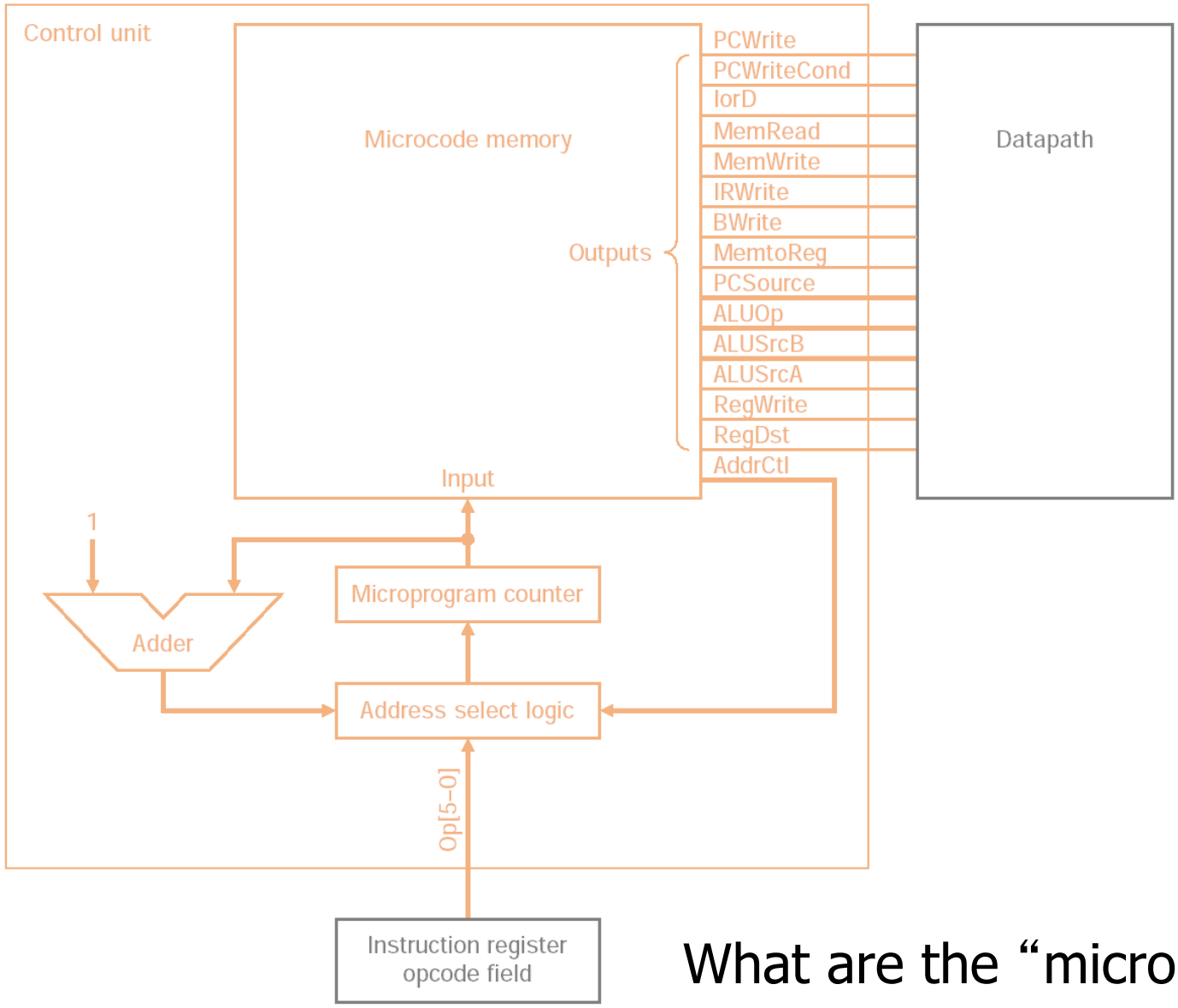
Dispatch ROM 1		
Op	**Opcode name**	**Value**
000000	R-format	0110
000010	`jmp`	1001
000100	`beq`	1000
100011	`lw`	0010
101011	`sw`	0010

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

Microprogramming



What are the “microinstructions” ?

Microprogramming

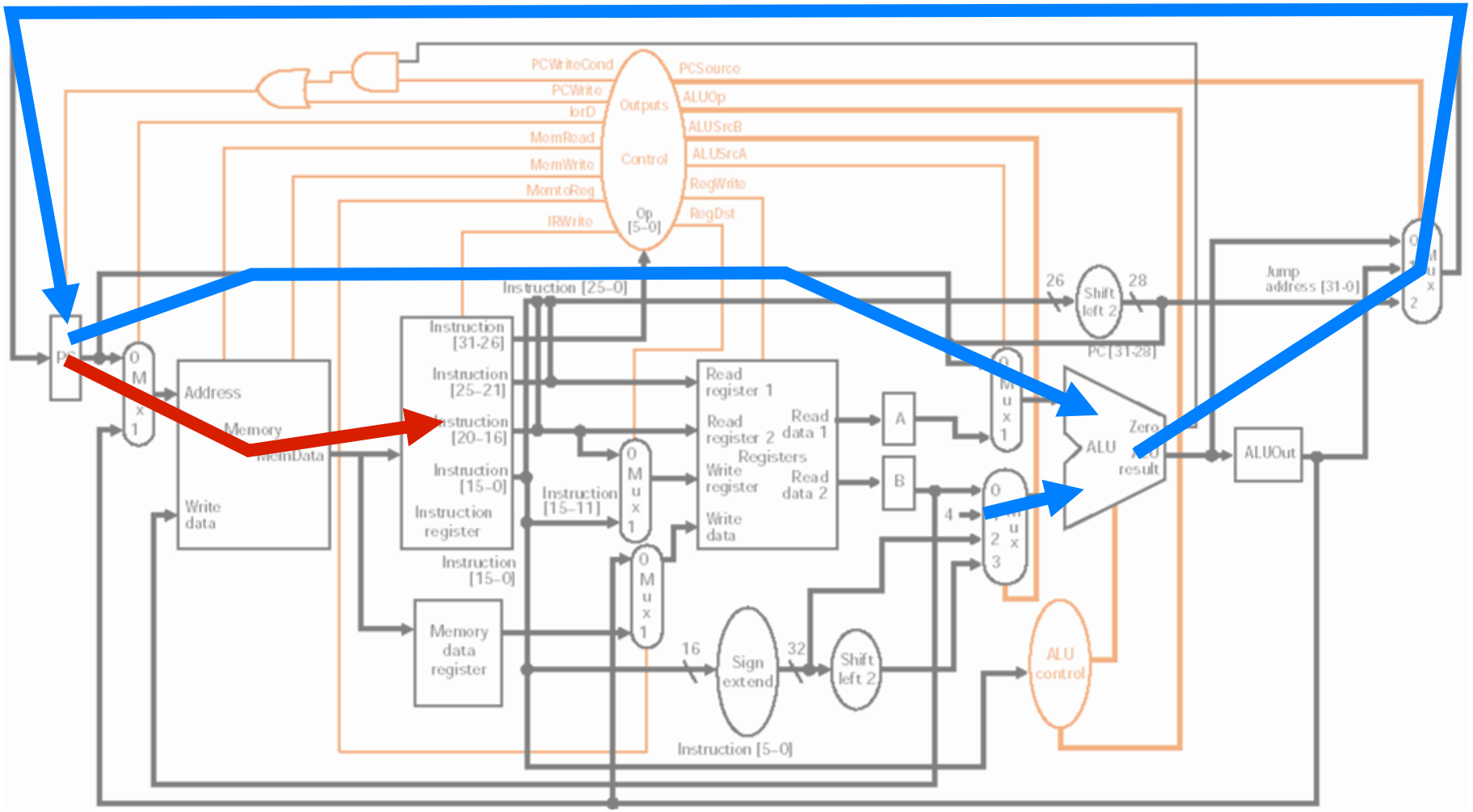
Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch
BEQ1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

- Each state is a microinstruction
- Signals specified symbolically - don't need to deal with 1/0's
- Labels for sequencing - don't need to set addresses manually

Will two implementations of the same architecture have the same microcode?

What would a microassembler do?

Step 1: Instruction Fetch



IR = Memory[PC];

PC = PC + 4;

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.

Maximally vs. Minimally Encoded Control Signals

- No encoding:
  - 1 bit for each datapath control signal
  - Faster, requires more memory (logic)
  - Used for Vax 780 — an astonishing 400K of memory!
- Lots of encoding:
  - Send control codes through more logic to get control signals
  - Uses less memory, slower
- Historical context:
  - 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 - too easy…

To Quote Intel…

The Pentium(R) Pro processor and Pentium(R) II processor may contain design defects or errors known as errata that may cause the product to deviate from published specifications. Many times, the effects of the errata can be avoided by implementing hardware or software work-arounds, which are documented in the Pentium Pro Processor Specification Update and the Pentium II Processor Specification Update. Pentium Pro and Pentium II processors include a feature called "reprogrammable microcode", which allows certain types of errata to be worked around via microcode updates. The microcode updates reside in the system BIOS and are loaded into the processor by the system BIOS during the Power-On Self Test, or POST.

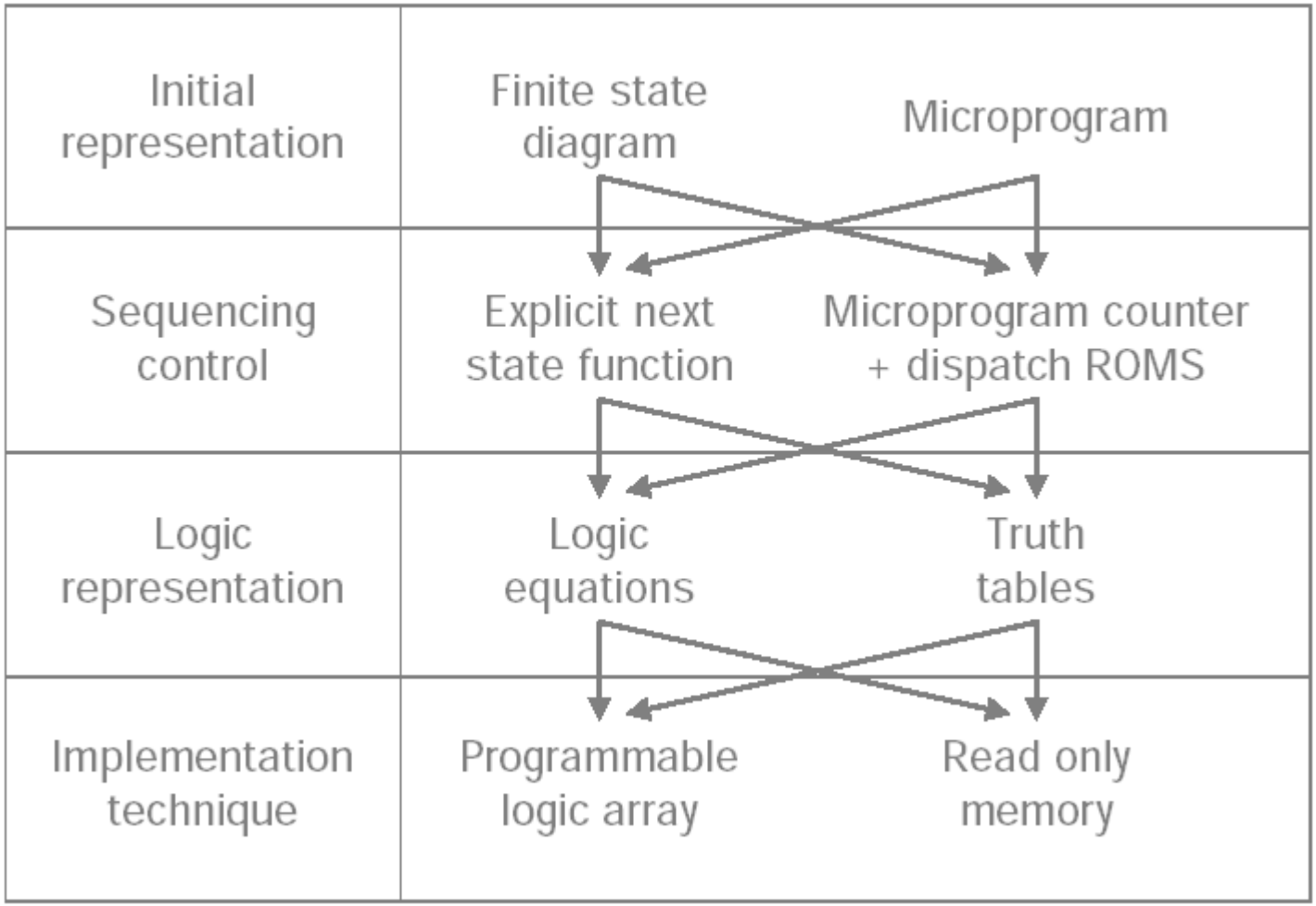
Microcode Trade-offs

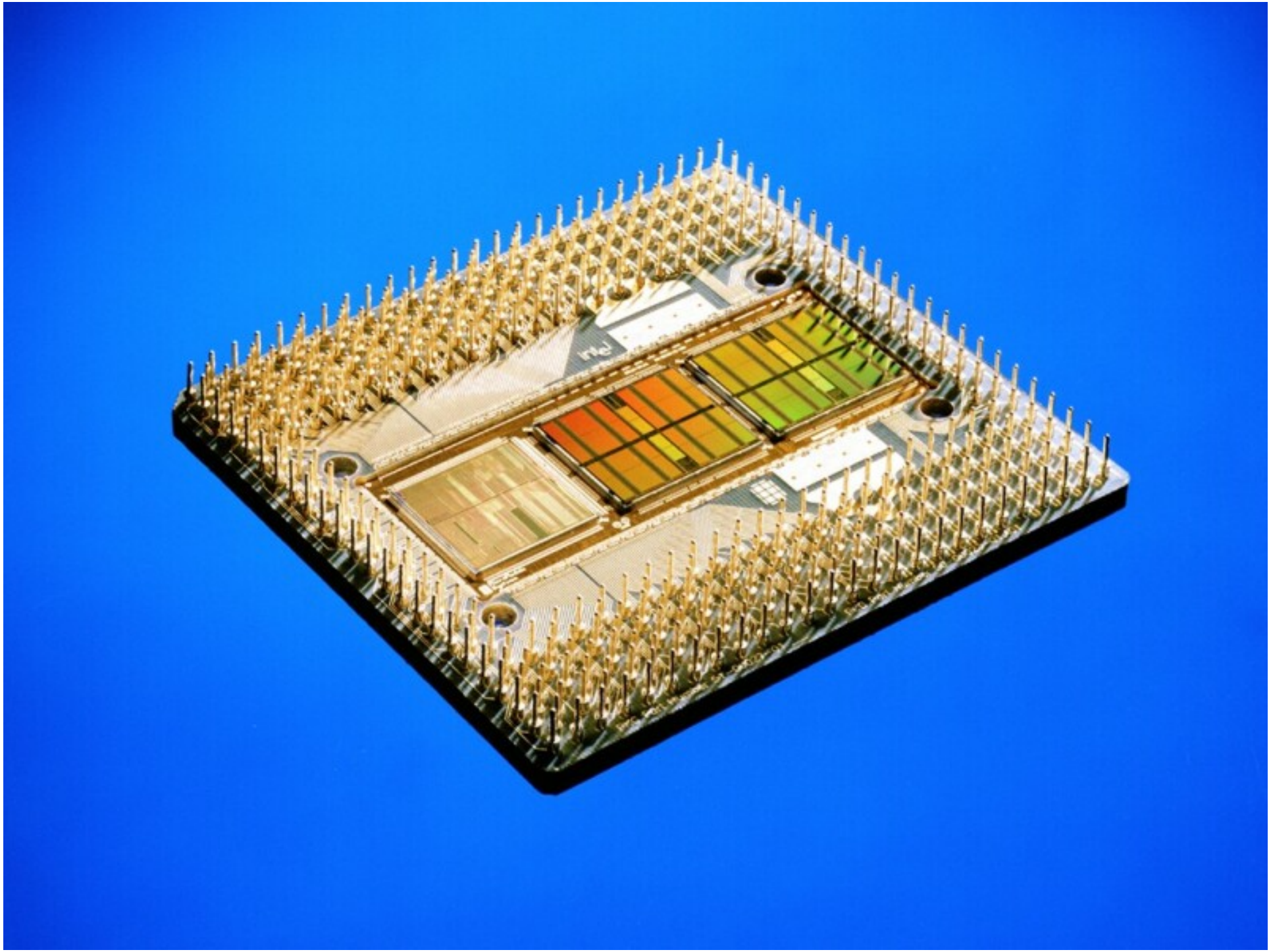
- Advantages:
  - Easy to design and write
  - Design architecture and microcode in parallel
  - Potentially easier to change since values are in memory
  - Can emulate other architectures

    (IBM was big on this: 7090→360)

- Disadvantages:
  - May be slower than optimized logic
  - May be less compact as well

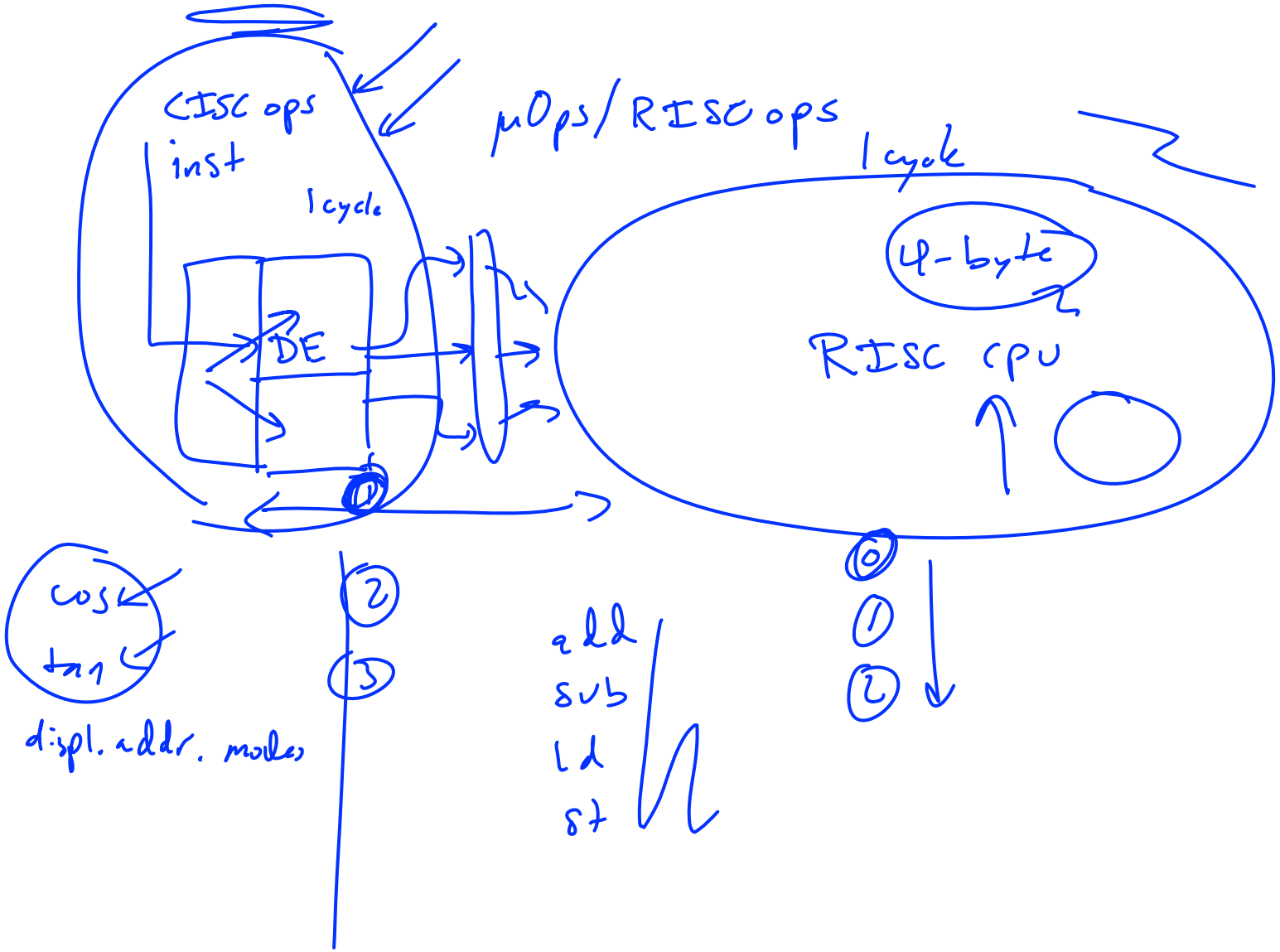
Control Specification Implementation Summary





Pentium Pro and Later

Microcode → Micro-Ops




NO EXCEPTIONS

Exceptions

In our simple MIPS machine:

- What happens if instruction encoding is not valid?
- What about arithmetic overflow?

Exception

An *unscheduled* event that disrupts program execution.

Interrupt

An *external unscheduled* event that disrupts program execution.

What Happens?

When an exception occurs:

- Save the current PC in the EPC (Exception PC)
- Store the cause in the Cause register
- Jump to the OS, some pre-specified address (possibly vectored)

OS can examine Cause and EPC

EPC and Cause?

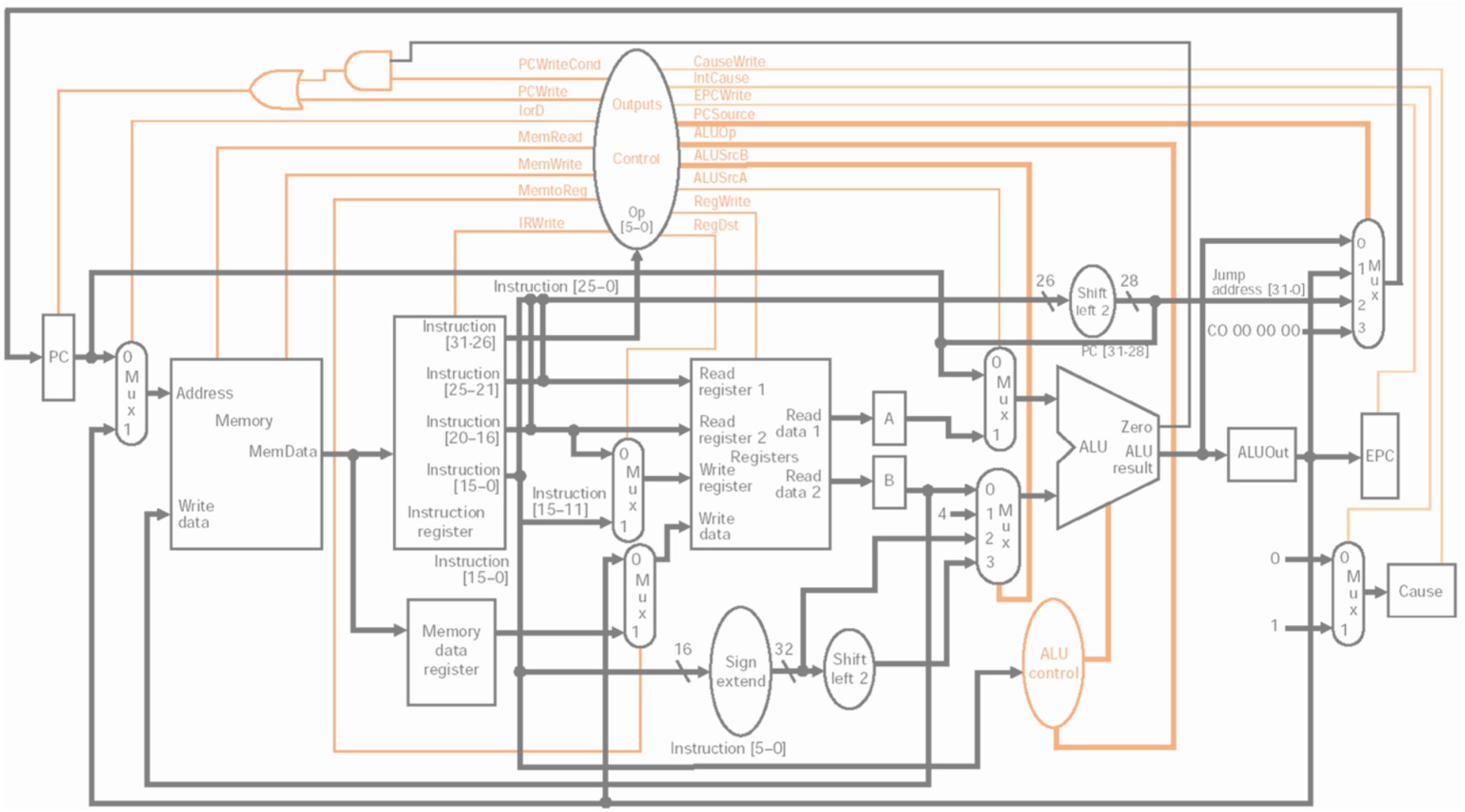
What Happens?

In our case

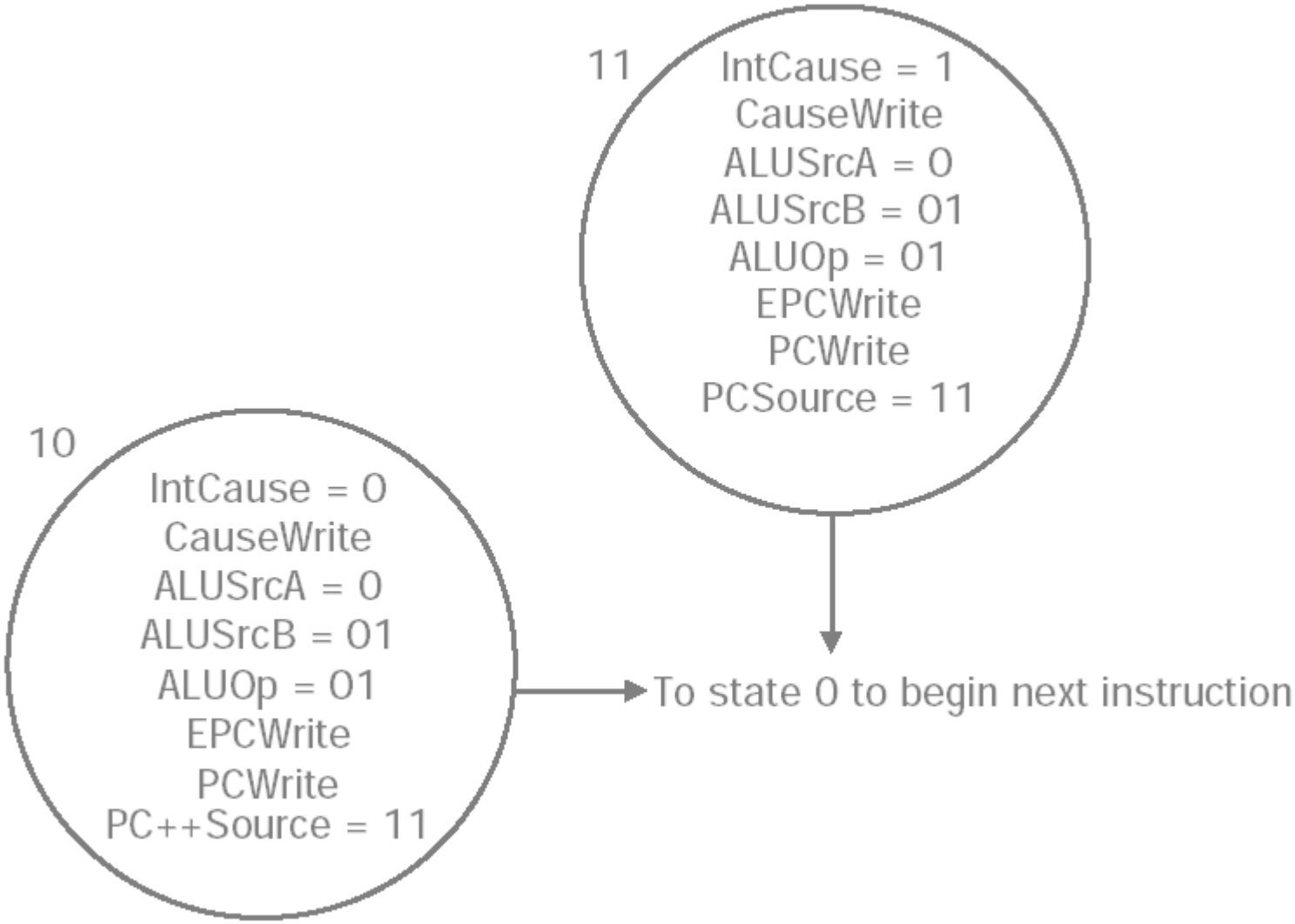
When an exception occurs:

- Save the current PC in the EPC
- Cause = 0 for Undefined Instruction, 1 for Overflow
- Jump to the OS at $C0000000_{16}$ (not vectored)

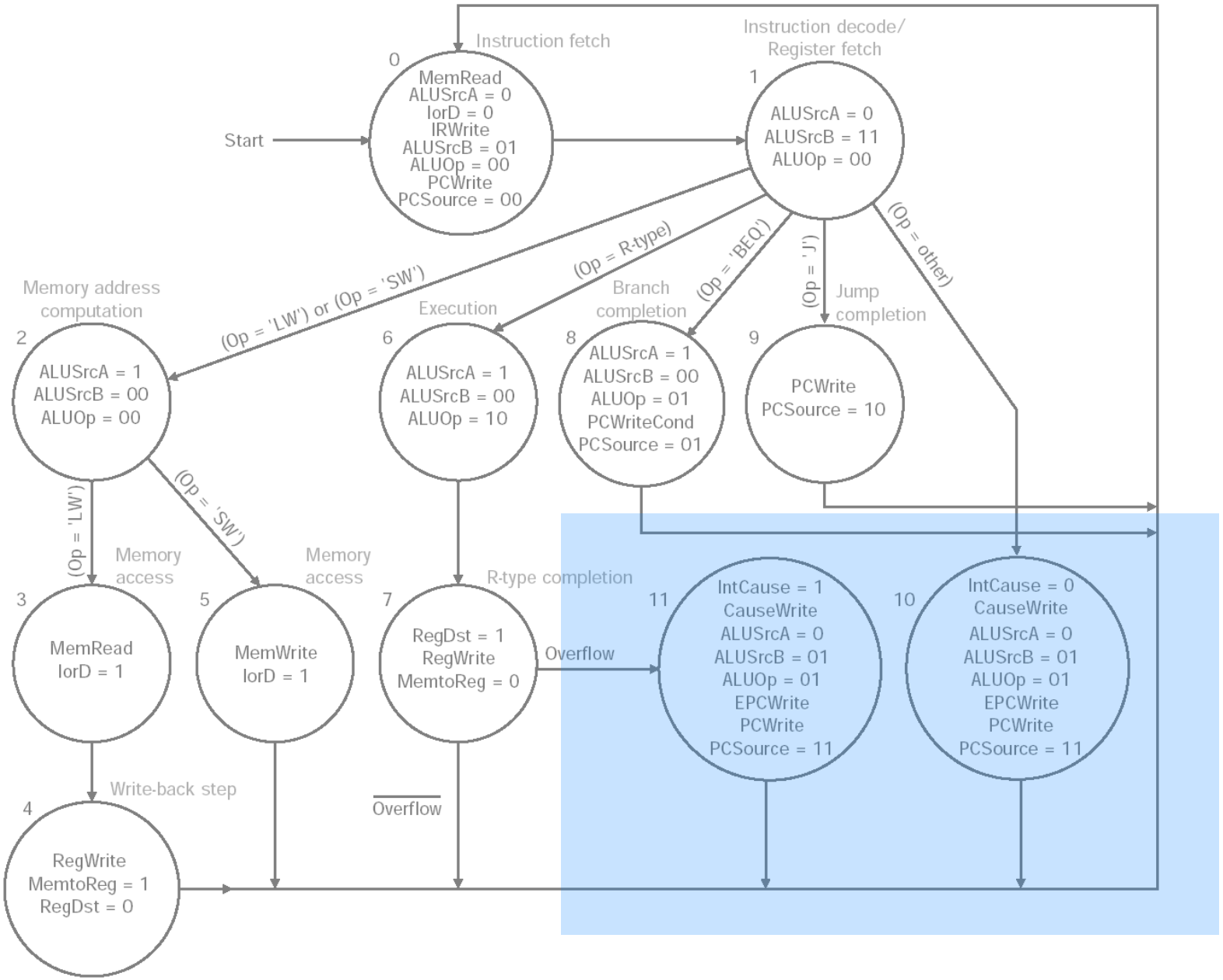
Exceptions


PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Outputs
Control
Op
[5–0]
CauseWrite
IntCause
EPCWrite
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
Instruction [25–0]
26
Shift
left 2
28
PC [31-28]
Jump
address [31-0]
CO 00 00 00
PC
Address
Memory
MemData
Write
data
Instruction
[31-26]
Instruction
[25–21]
Instruction
[20–16]
Instruction
[15–0]
Instruction
register
Instruction
[15–0]
Instruction
[15–11]
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
A
B
4
ALU
Zero
ALU
result
ALUOut
EPC
Cause
Memory
data
register
16
Sign
extend
32
Shift
left 2
ALU
control
Instruction [5–0]
0
1
M
u
x

Exceptions


11
IntCause = 1
CauseWrite
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 01
EPCWrite
PCWrite
PCSource = 11
10
IntCause = 0
CauseWrite
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 01
EPCWrite
PCWrite
PC++Source = 11
To state 0 to begin next instruction

Exceptions


Instruction fetch
0
MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00
Start
Instruction decode/
Register fetch
1
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
(Op = other)
Memory address
computation
2
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 00
Execution
6
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
Branch
completion
8
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01
Jump
completion
9
PCWrite
PCSource = 10
(Op = 'LW')
(Op = 'SW')
Memory
access
3
MemRead
IorD = 1
Memory
access
5
MemWrite
IorD = 1
R-type completion
7
RegDst = 1
RegWrite
MemtoReg = 0
Overflow
11
IntCause = 1
CauseWrite
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 01
EPCWrite
PCWrite
PCSource = 11
10
IntCause = 0
CauseWrite
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 01
EPCWrite
PCWrite
PCSource = 11
Write-back step
4
RegWrite
MemtoReg = 1
RegDst = 0
Overflow

Summary, Next Time, and Now

Summary

- Control can be hardwired, microcoded, or both
- Exceptions complicate things

Next Time

- Pipelining next time