U. Wisconsin CS/ECE 552
Introduction to Computer Architecture

Prof. Karu Sankaralingam

Miscellaneous (5.5, 5.7, 5.6, & 6.8)

www.cs.wisc.edu/~karu/courses/cs552

Slides combined and enhanced by Karu Sankaralingam from work by Falsafi, Hill, Marculescu, Nagle, Patterson, Roth, Rutenbar, Schmidt, Shen, Sohi, Sorin, Thottethodi, Vijaykumar, & Wood

Outline

- Multicycle Design (5.5)
- Implementing Control & Microprogramming (5.7)
- Exceptions (5.6)
- Exceptions in a Pipeline (6.8)

Multicycle Approach (No Pipelining)

- 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
  - introduce additional "internal" registers

Multicycle Approach

- 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 solely by instruction
  - e.g., what should the ALU do for a "subtract" instruction?
- We'll use a finite state machine for control

What Instructions Need to Do

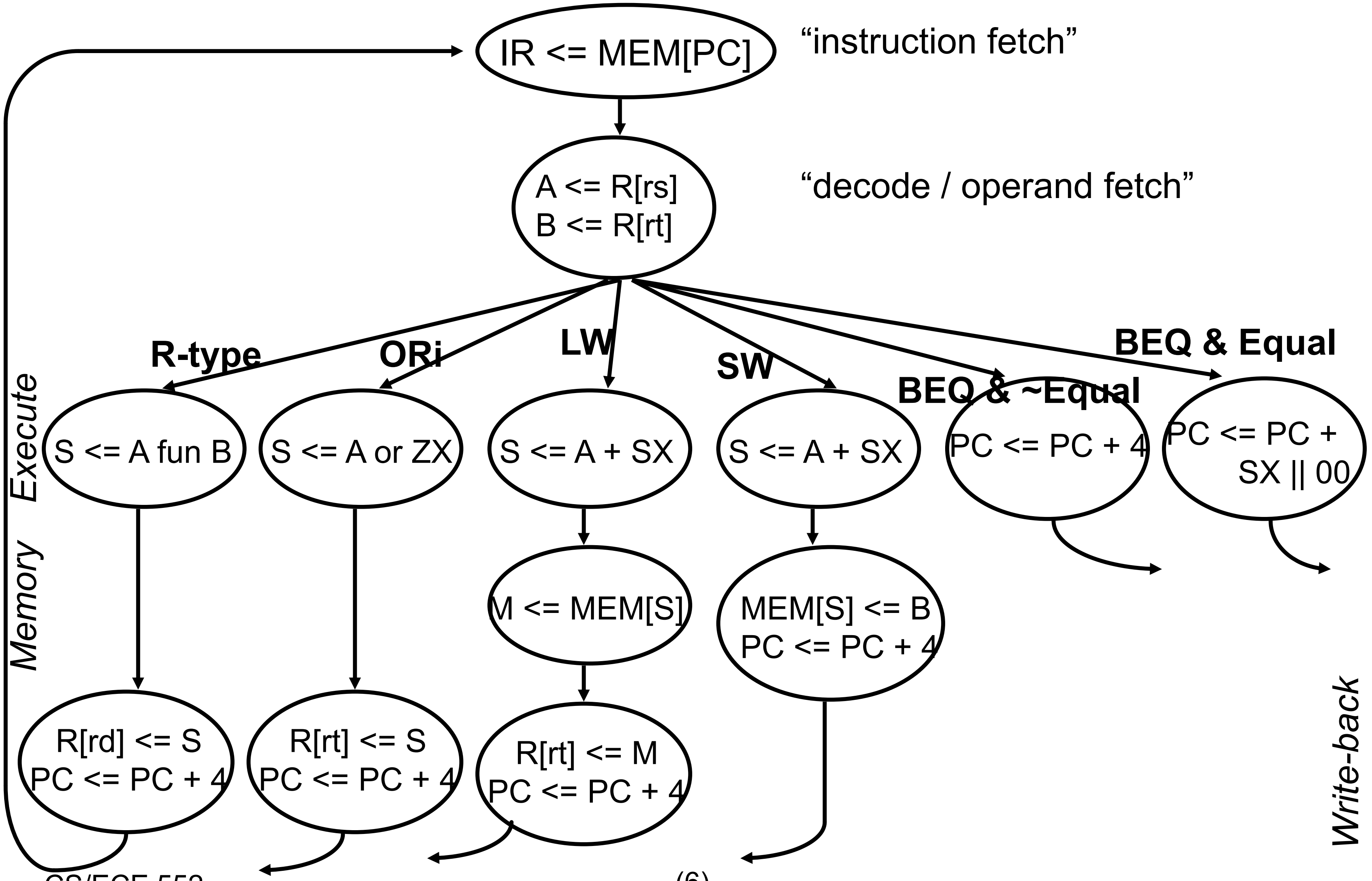
Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps

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)			

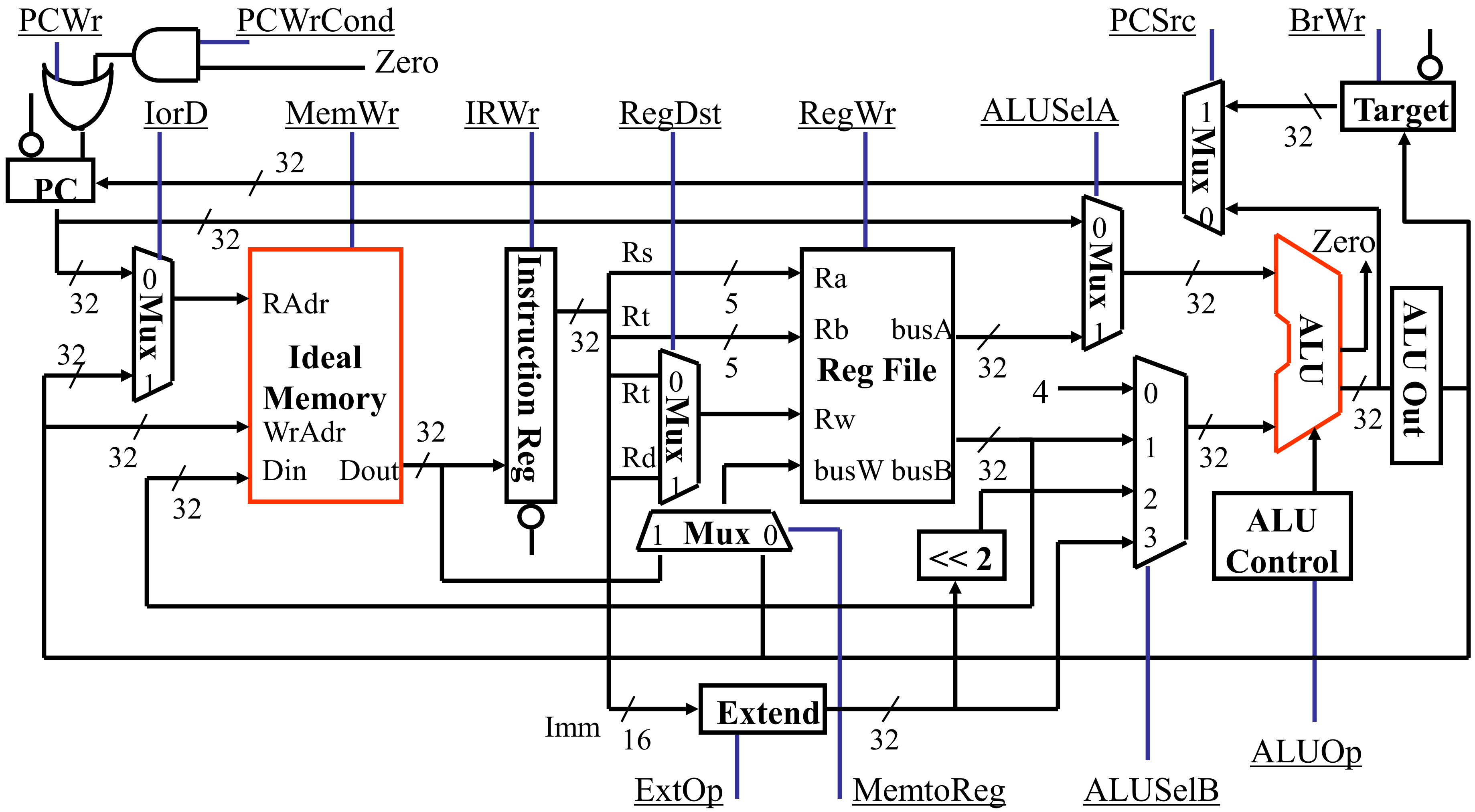
Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + sign-extend (IR[15-0])	if (A ==B) then PC = ALUOut	PC = PC [31-28] II (IR[25-0]<<2)
Memory access or R-type completion	Reg [IR[15-11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
Memory read completion		Load: Reg[IR[20-16]] = MDR		

FSM view of Control



Multicycle Datapath

- Miminizes Hardware: 1 memory, 1 adder



Outline

- Multicycle Design (5.5)
- Implementing Control & Microprogramming (5.7)
- Exceptions (5.6)
- Exceptions in a Pipeline (6.8)

Implementing the Control

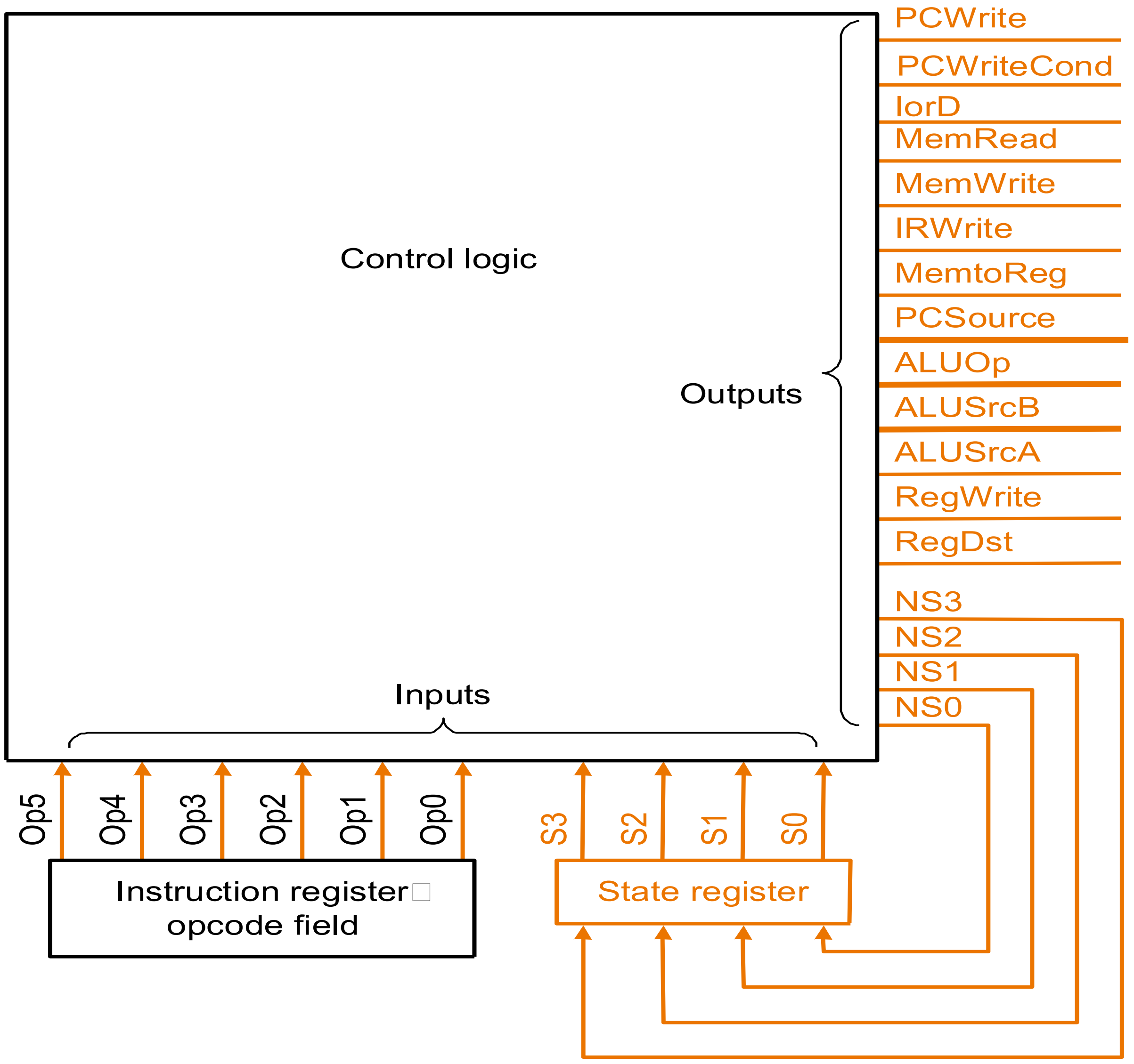
- Value of control signals is dependent upon:
  - what instruction is being executed
  - which step is being performed

- Use the information we've acculumated to specify a finite state machine
  - specify the finite state machine graphically, or
  - use microprogramming

- Implementation can be derived from specification

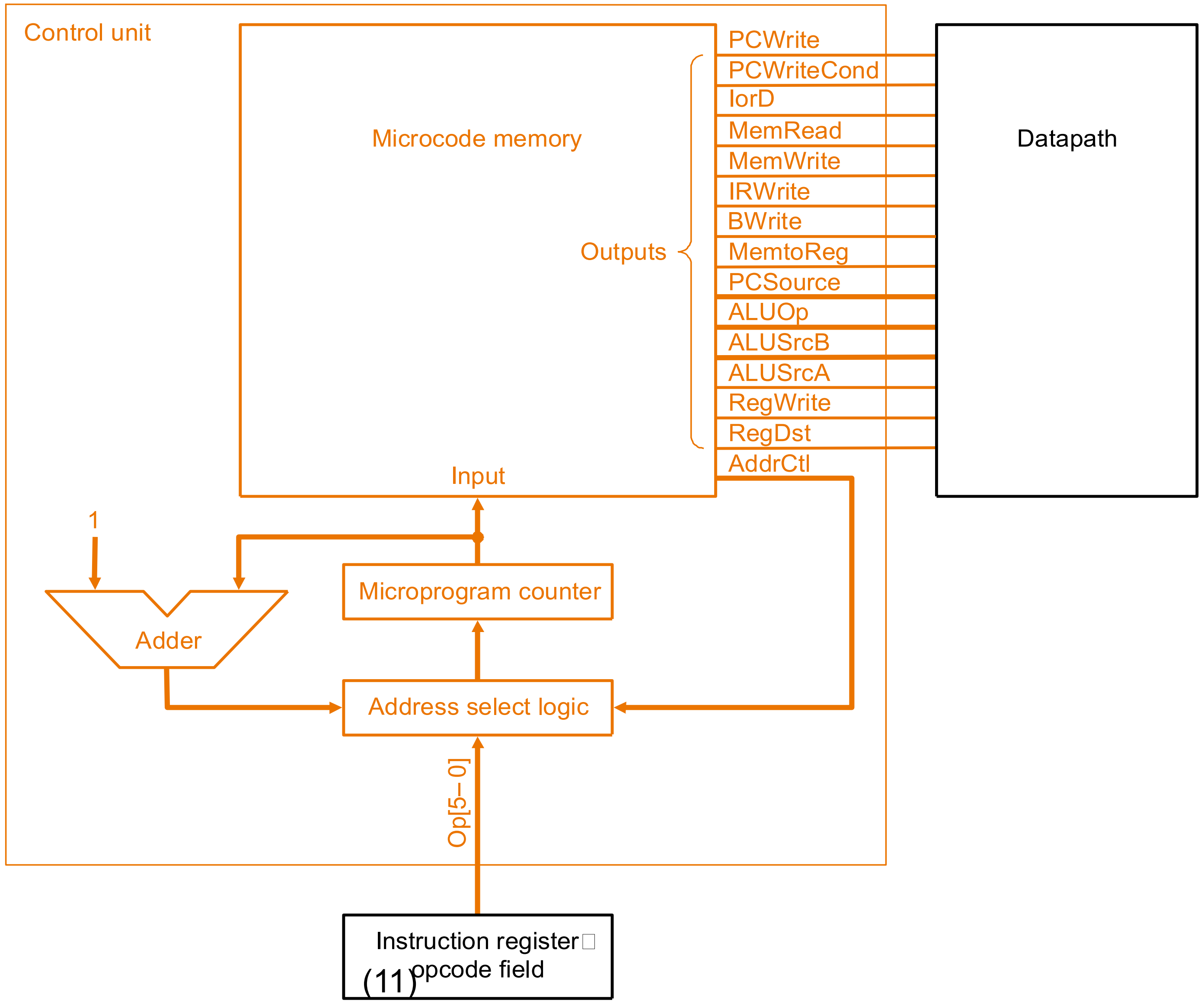
Finite State Machine for Multicycle Control

- Implementation
- State bits
  - D-flipflops
- Control Logic
  - Comb. Block
  - Use PLA or ROM

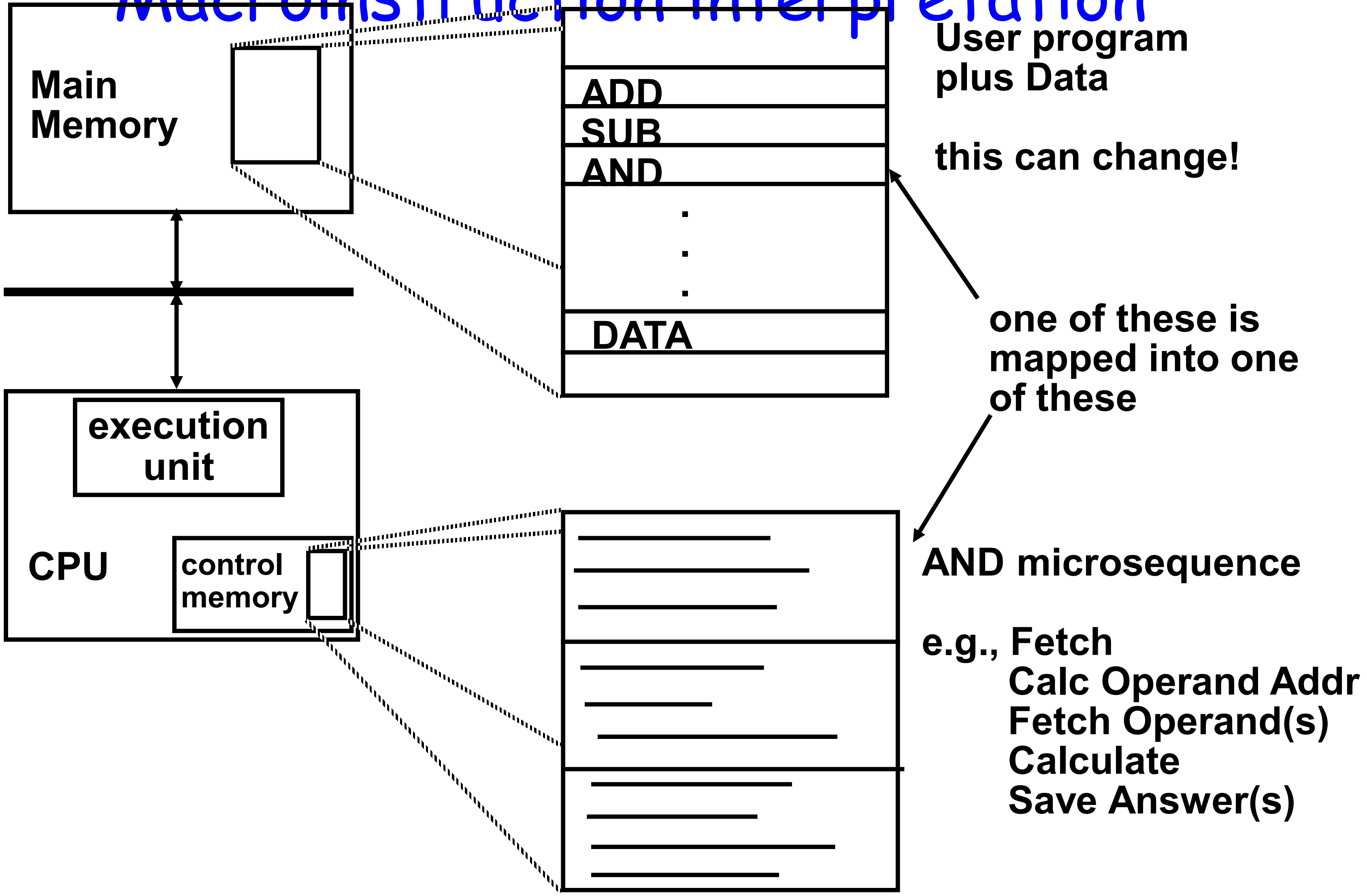


Alternative: Microprogramming

- Sequence of RTL steps
  - program using microinstructions



Macroinstruction interpretation



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

- A specification methodology (alternate to FSM)
  - appropriate if hundreds of opcodes, modes, cycles, etc.
  - signals specified symbolically using microinstructions
  - Microassembler?

Microprogramming Pros and Cons

- Ease of design
- Flexibility
  - Easy to adapt to changes in organization, timing, technology
  - Can make changes late in design cycle, or even in the field
- Can implement very powerful instruction sets (just more control memory)
- Generality
  - Can implement multiple instruction sets on same machine.
  - Can tailor instruction set to application.
- Compatibility
  - Many organizations, same instruction set
- Costly to implement
- Slow

Next time: memory

Control: Summary

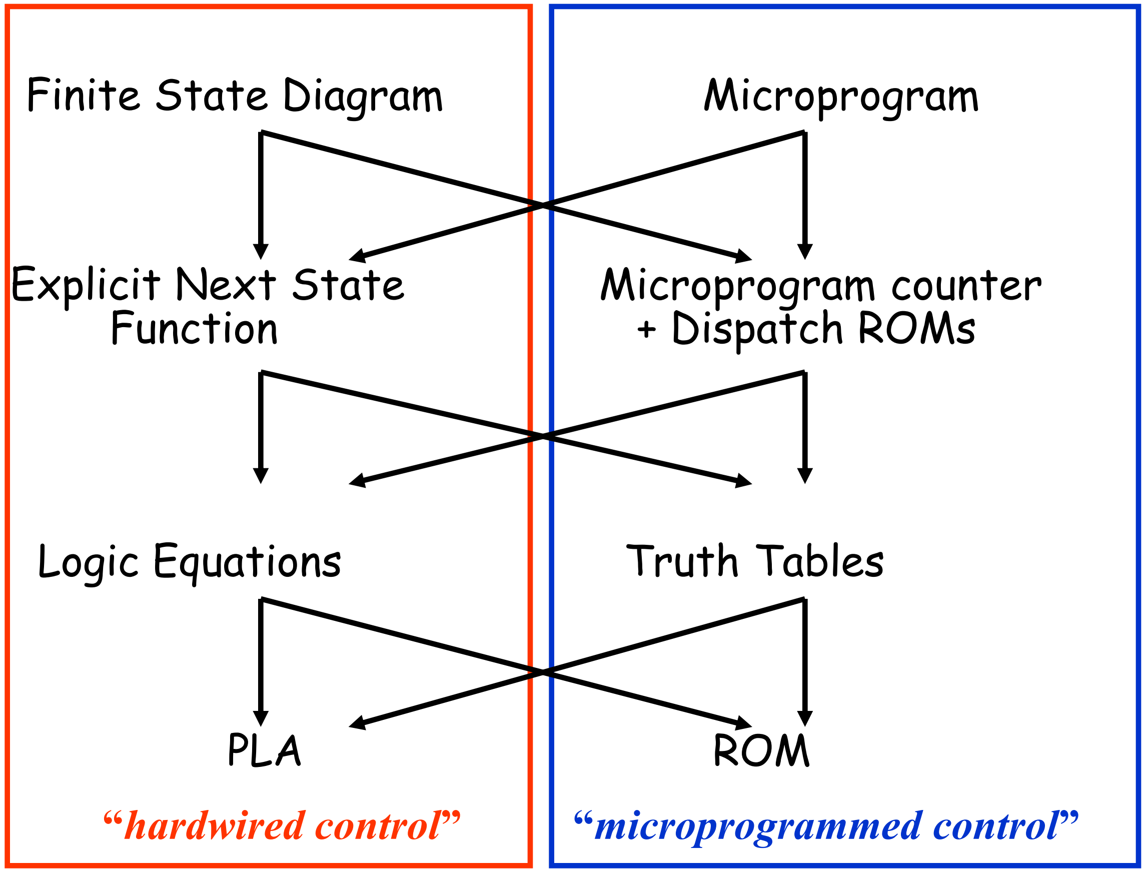
Control is the hard part

Initial Representation

Sequencing Control

Logic Representation

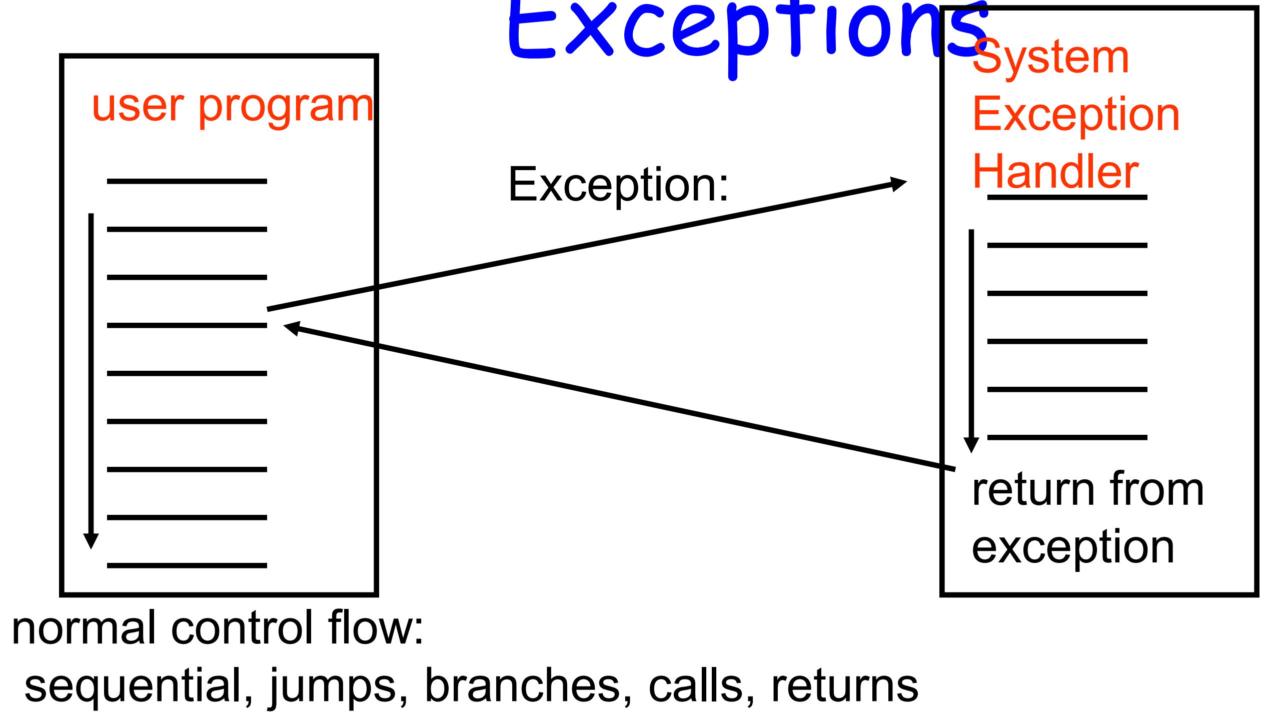
Implementation Technique


Finite State Diagram
Microprogram
Explicit Next State Function
Microprogram counter + Dispatch ROMs
Logic Equations
Truth Tables
PLA
ROM
"hardwired control"
"microprogrammed control"

Outline

- Multicycle Design (5.5)
- Implementing Control & Microprogramming (5.7)
- Exceptions (5.6)
- Exceptions in a Pipeline (6.8)

Exceptions



- Exception = unprogrammed control transfer
  - system takes action to handle the exception
    - must record the address of the offending instruction
  - returns control to user
  - must save & restore user state
- Allows constuction of a "user virtual machine"

Interrupt, Exception, Trap?

- Interrupts
  - caused by external events
  - asynchronous to program execution
  - may be handled between instructions
  - simply suspend and resume user program
- Traps
  - caused by internal events
    - exceptional conditions (overflow)
    - errors (parity)
    - faults (non-resident page)
  - synchronous to program execution
  - condition must be remedied by the handler
  - instruction may be retried or simulated and program continued or program may be aborted
- MIPS convention:
  - External : Interrupts
  - Internal : Exception

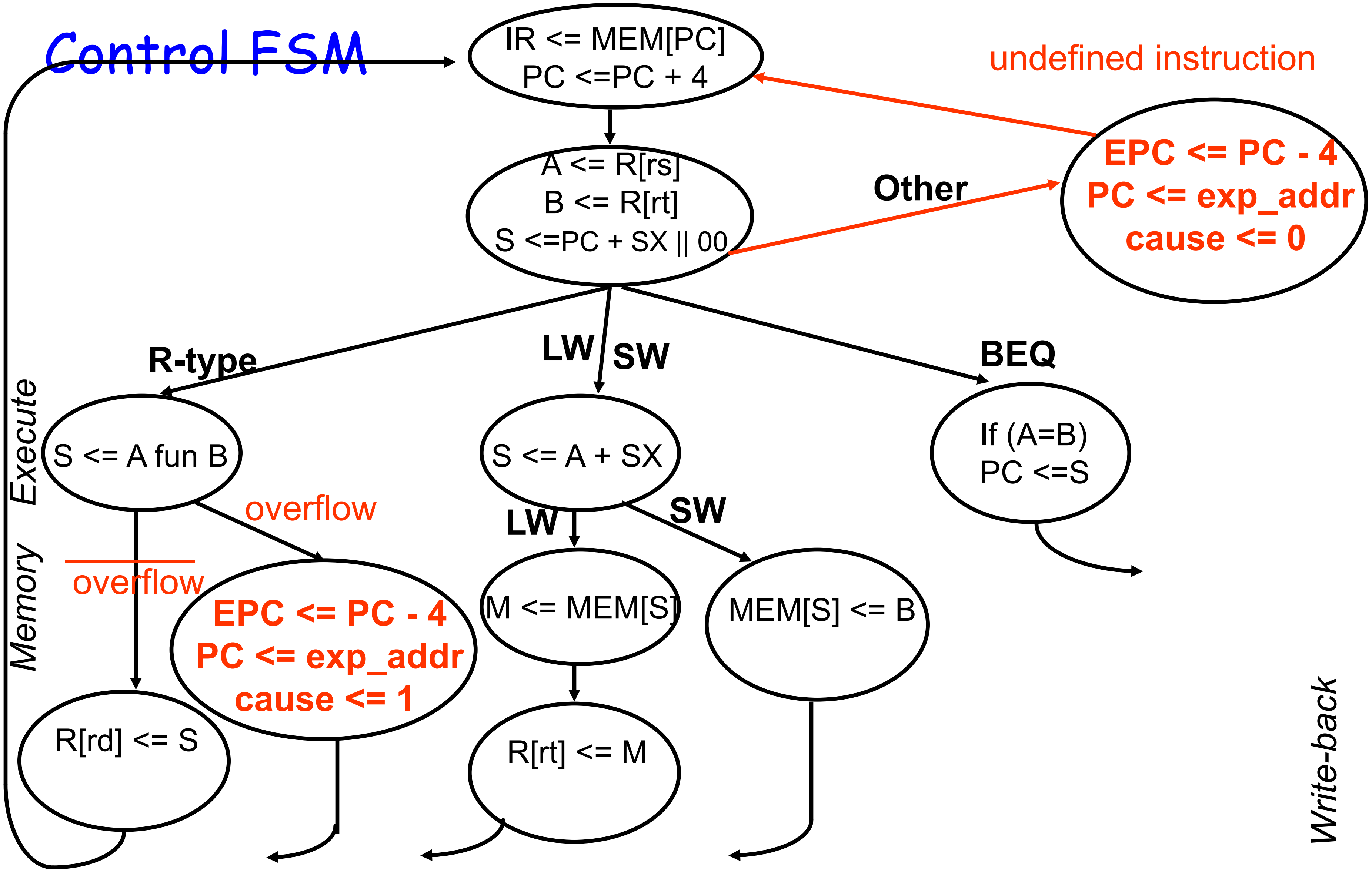
Exception Semantics

- MIPS architecture defines the instruction as having <u>no effect</u> if the instruction causes an exception.
- When get to virtual memory we will see that certain classes of exceptions must prevent the instruction from changing the machine state.
- This aspect of handling exceptions becomes complex and potentially limits performance => why it is hard
  - Precise interrupts vs Imprecise interrupts

MIPS Exceptions

- All exceptions jump to same handler code
  - "Cause" register
- We consider
  - Illegal instructions
  - Arithmetic overflows
- Handler behavior
  - Save PC of offending instruction (How? PC+4 has already been written to PC)
  - Use special register EPC(why not use $31 like jal?)
  - Set cause register appropriately (0=ILL; 1=OVF)
  - Jump to handler at fixed address

Control FSM


IR <= MEM[PC]
PC <=PC + 4
A <= R[rs]
B <= R[rt]
S <=PC + SX || 00
undefined instruction
Other
EPC <= PC - 4
PC <= exp_addr
cause <= 0
R-type
LW
SW
BEQ
Execute
S <= A fun B
S <= A + SX
If (A=B)
PC <=S
overflow
LW
SW
Memory
~~overflow~~
EPC <= PC - 4
PC <= exp_addr
cause <= 1
M <= MEM[S]
MEM[S] <= B
R[rd] <= S
R[rt] <= M
Write-back

Other issues

- Vectored exceptions
  - “cause” folded into handler address
  - Different causes jump to different handlers
- User vs kernel mode
- Software issues
  - Disabling exceptions in handler
- Returning from interrupt

Outline

- Multicycle Design (5.5)
- Implementing Control & Microprogramming (5.7)
- Exceptions (5.6)
- Exceptions in a Pipeline (6.8)

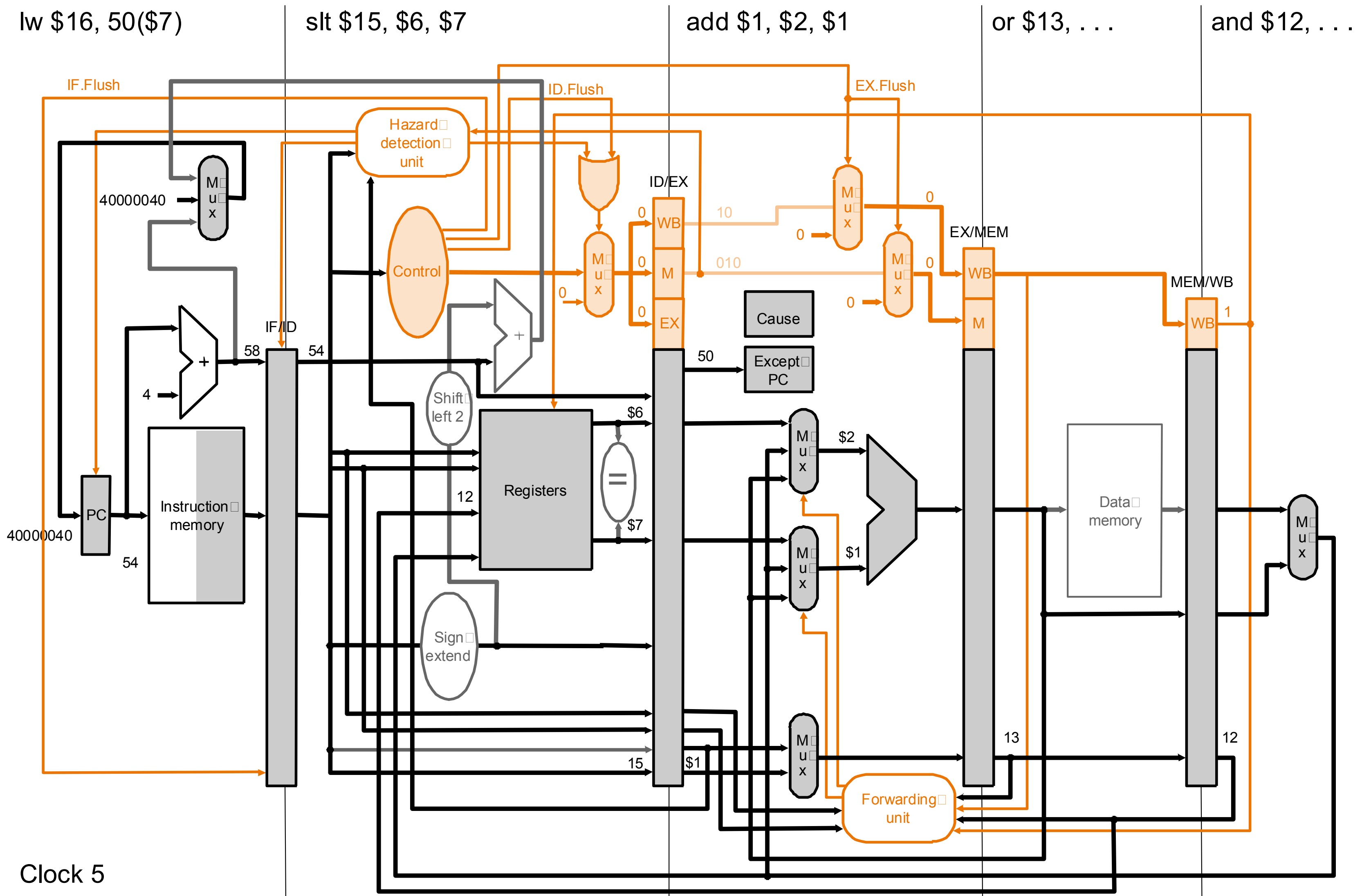
Exceptions

- Semantics
  - No instruction after the exception causing instruction may execute
  - Every instruction preceding the exception causing instruction must complete execution
  - Set cause register
  - Jump to exception handler address
- Multiple instructions (exceptions) in a cycle!

Datapath modifications

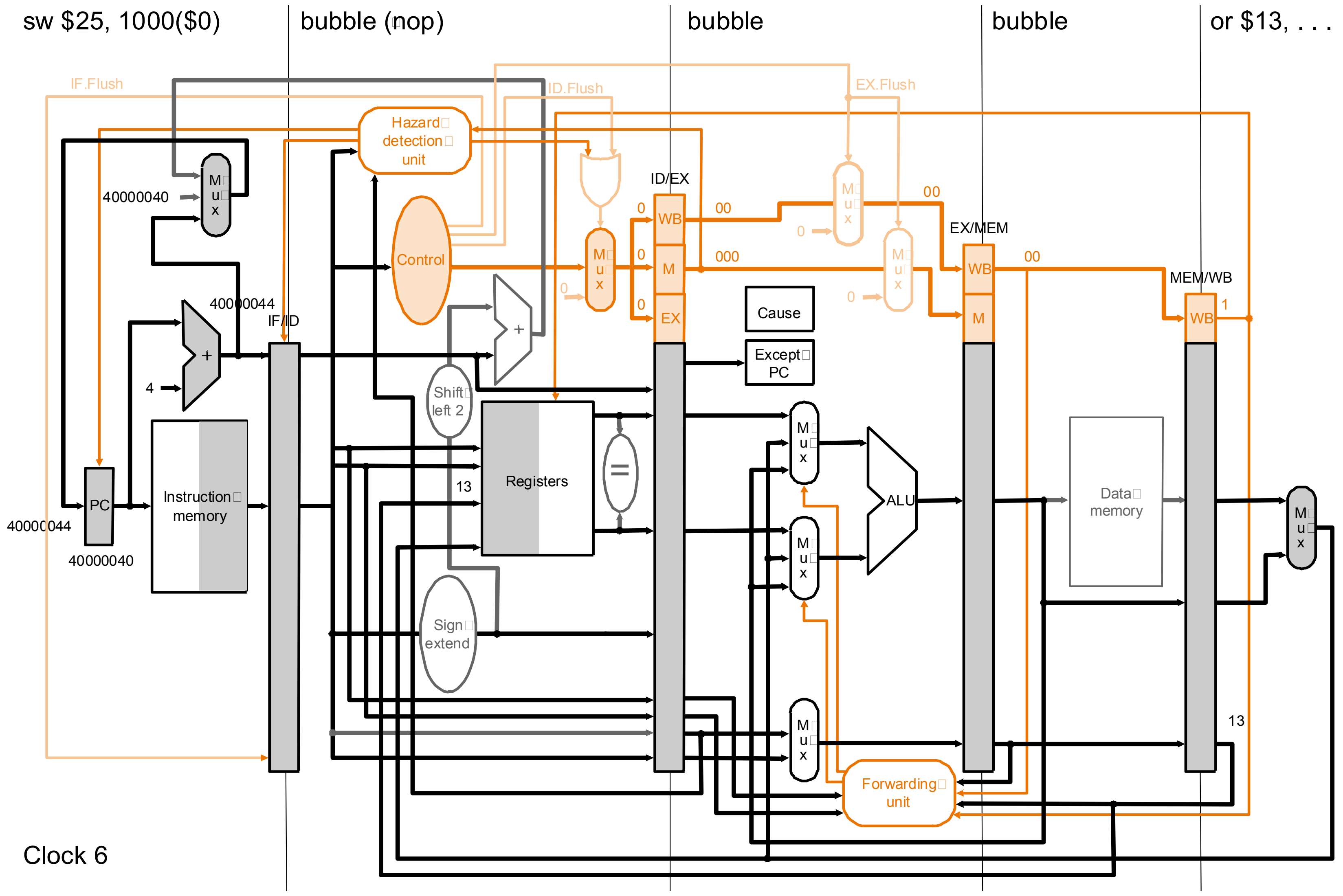
- Pipeline complications
- What stage is exception detected?
  - Overflow?
    - In EX stage, Also squash (convert to nop) EX stage
  - Illegal Instruction?
    - In ID stage, squash (convert to nop) ID stage
    - Similar to RAW hazard
  - What about external interrupts?
- Overflow in instruction i, illegal instruction in instruction i+1
  - Simultaneous exceptions
  - Hardware sorting

Walkthrough (1 of 2)



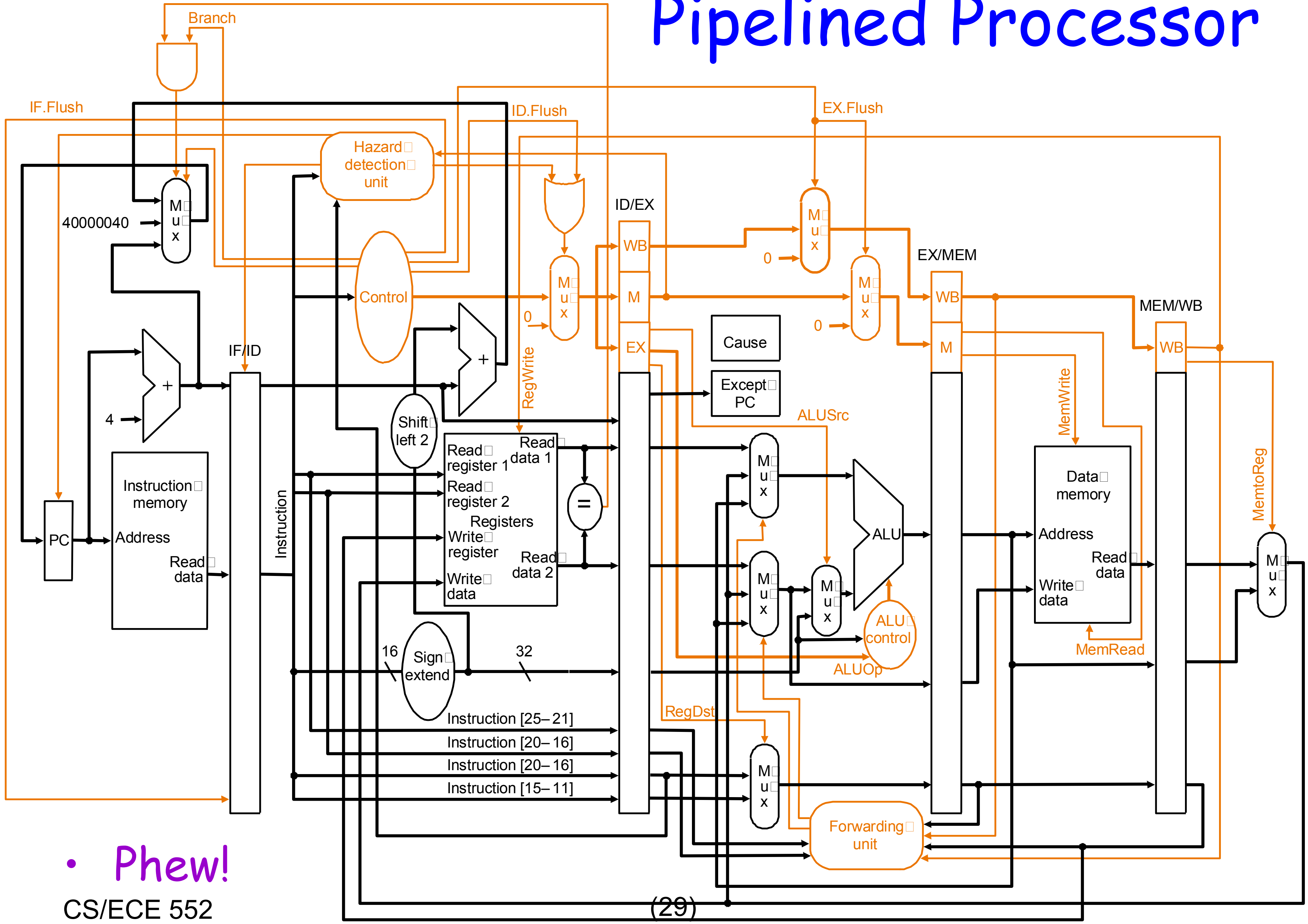
• All three instructions converted to nop

Walkthrough (2 of 2)



- Fetch next instruction from handler PC (MIPS)

Pipelined Processor



- Phew!