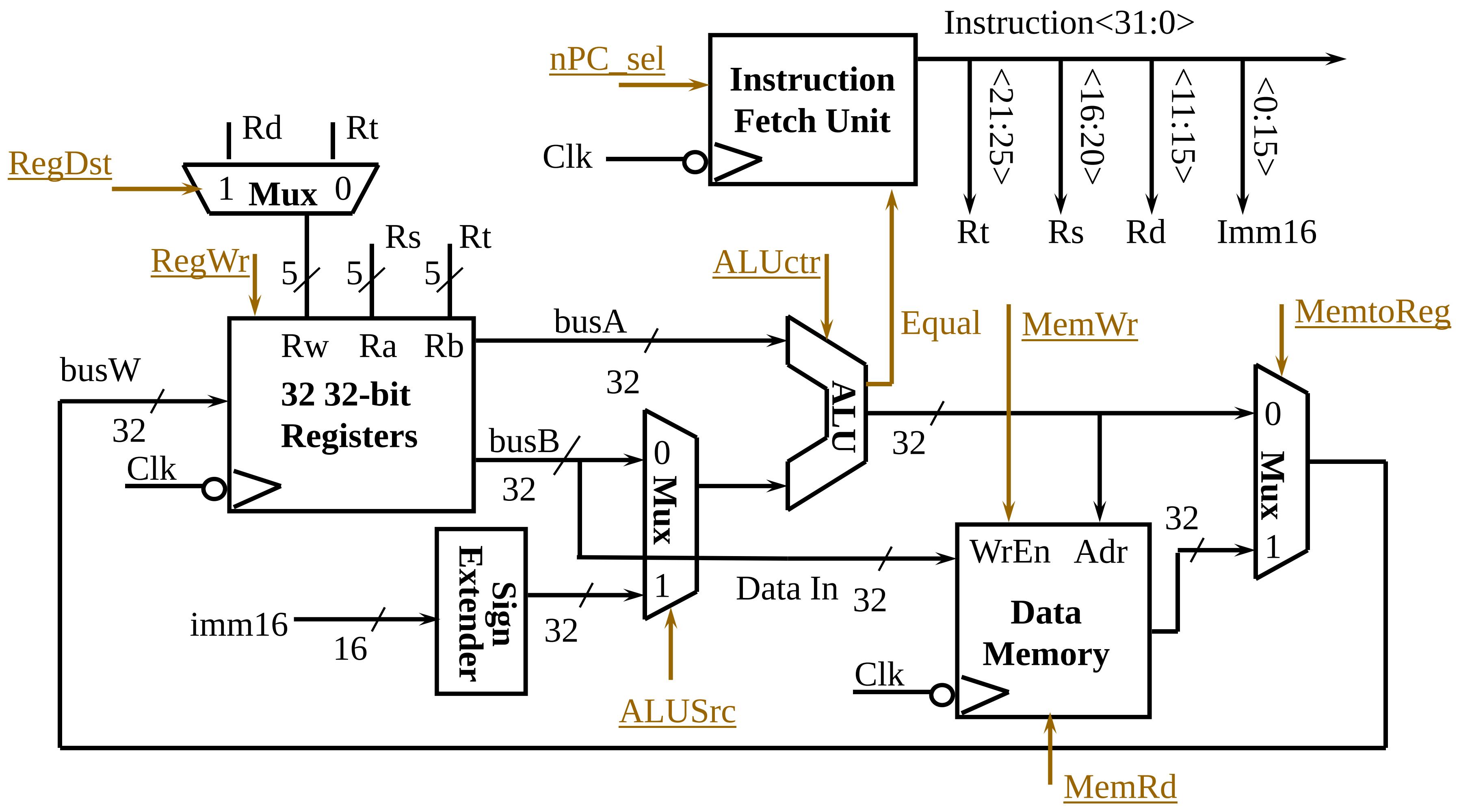
COMP303 - Computer Architecture Lecture 10

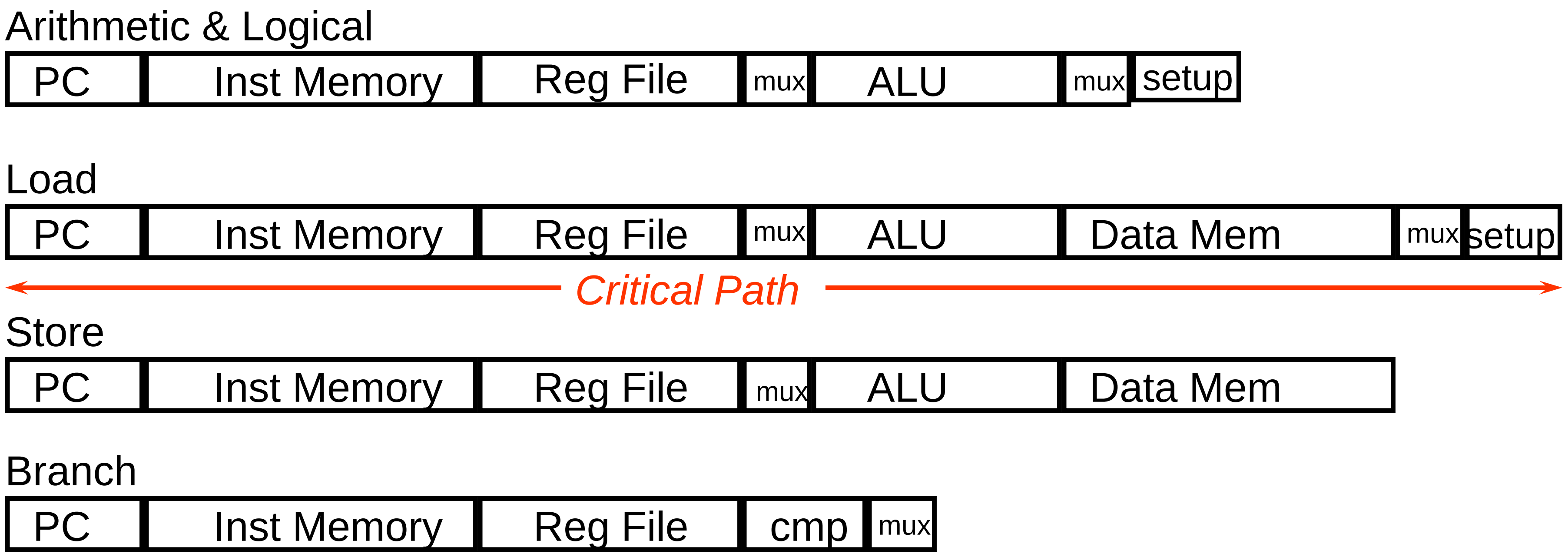
Multi-Cycle Design & Exceptions

Single Cycle Datapath

- We designed a processor that requires one cycle per instruction



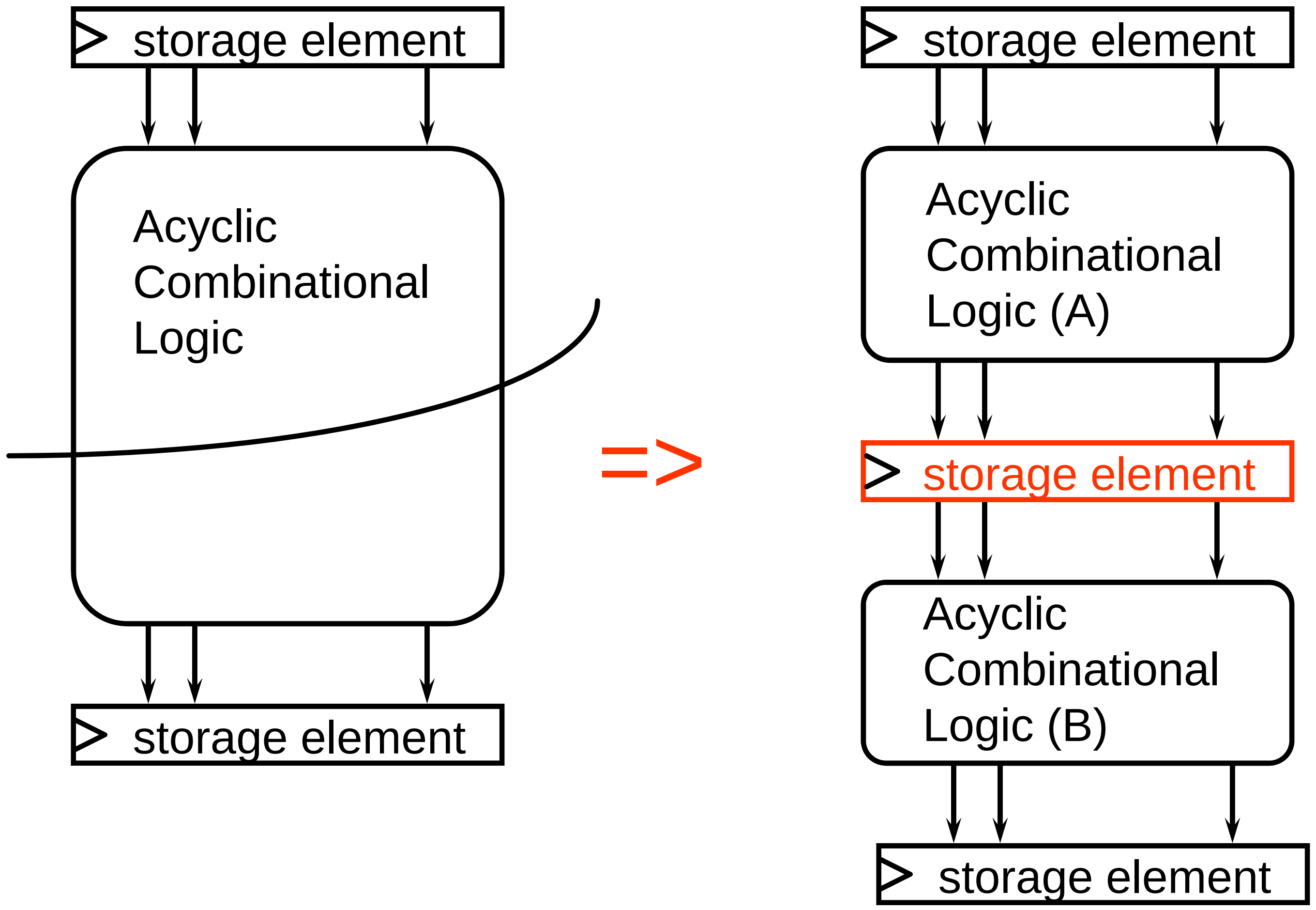
What's wrong with our CPI=1 processor?



- Long cycle time
- All instructions take as much time as the slowest
- Real memory is not so nice as our idealized memory
  - cannot always get the job done in one (short) cycle

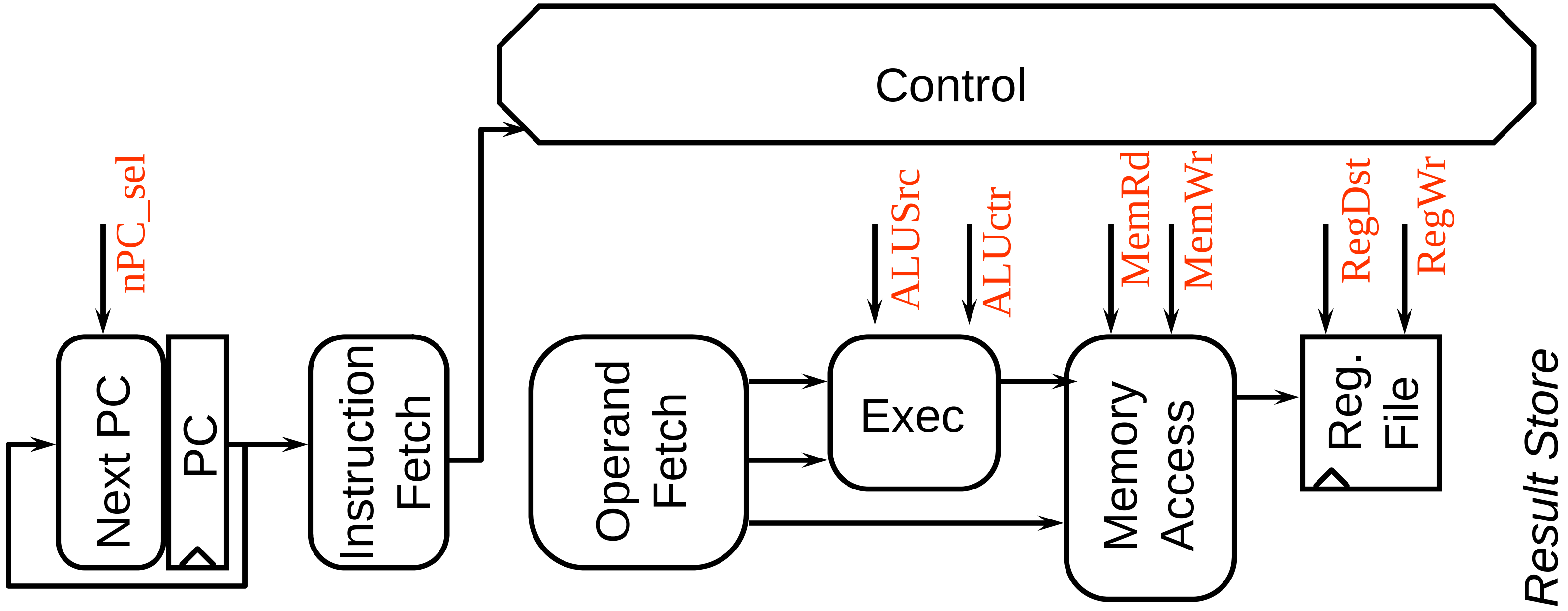
Reducing Cycle Time

- Cut combinational dependency graph and insert register / latch
- Do same work in two fast cycles, rather than one slow one


storage element
Acyclic Combinational Logic
storage element
=>
storage element
Acyclic Combinational Logic (A)
storage element
Acyclic Combinational Logic (B)
storage element

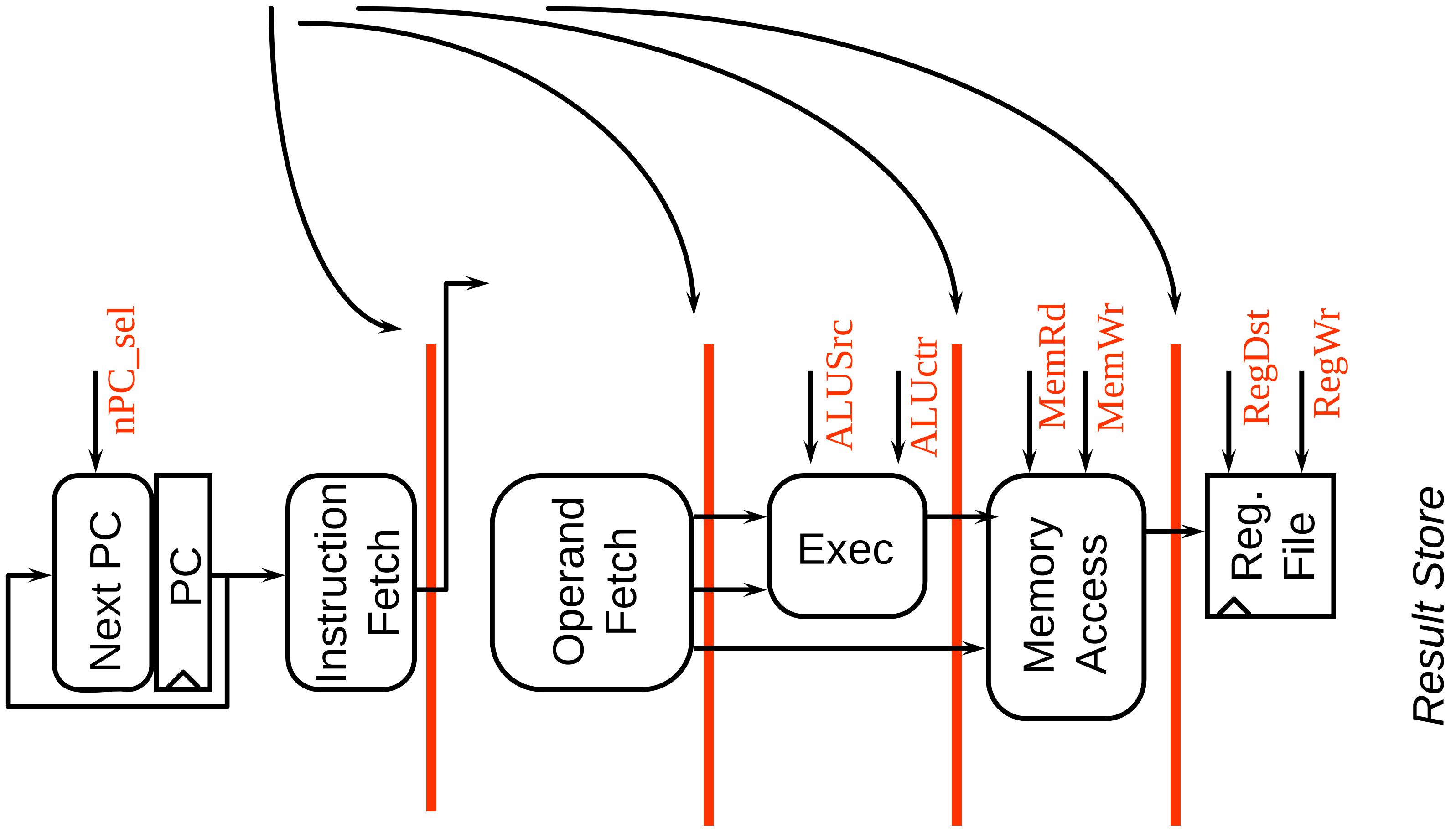
Basic Limits on Cycle Time

- Next address logic
  - PC <= branch ? PC + 4 + offset
    : PC + 4
- Instruction Fetch
  - InstructionReg <= Mem[PC]
- Operand Fetch
  - A <= R[rs], B <= R[rt]
- ALU execution
  - ALUOut <= A op B
- Data Memory Access
  - M <= Mem[ALUout]
- Result Store
  - R[rd] <= ALUOut



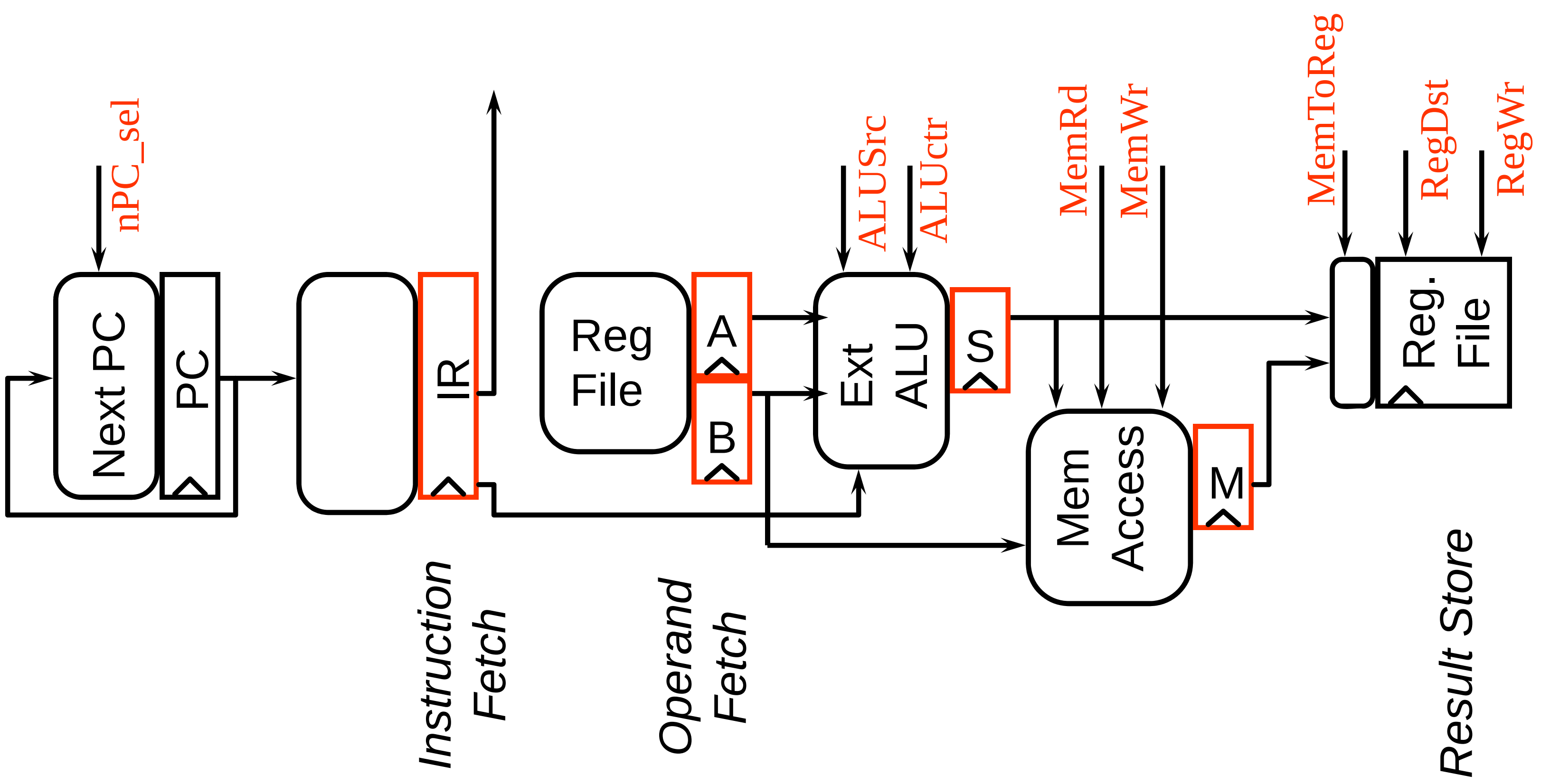
Partitioning the CPI=1 Datapath

- Add registers between smallest steps



Allow the instruction to take multiple cycles.

Example Multicycle Datapath

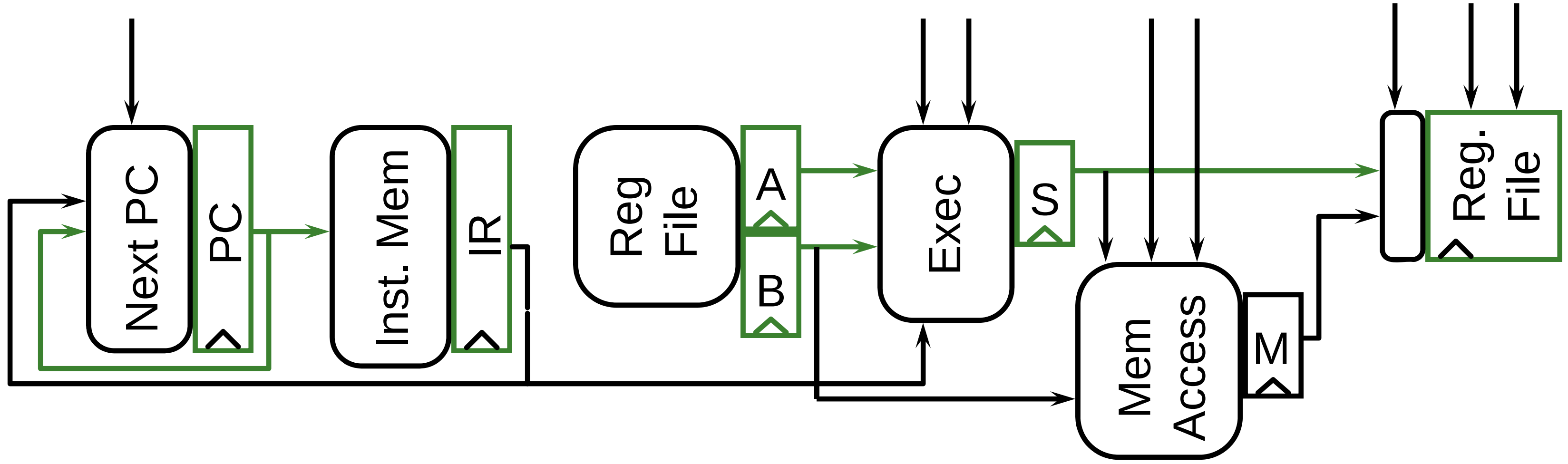


- Additional registers are added to store values between stages.

R-type instructions (add, sub, . . .)

inst	**Logical Register Transfers**
ADD	**R[rd] ← R[rs] + R[rt]; PC ← PC + 4**

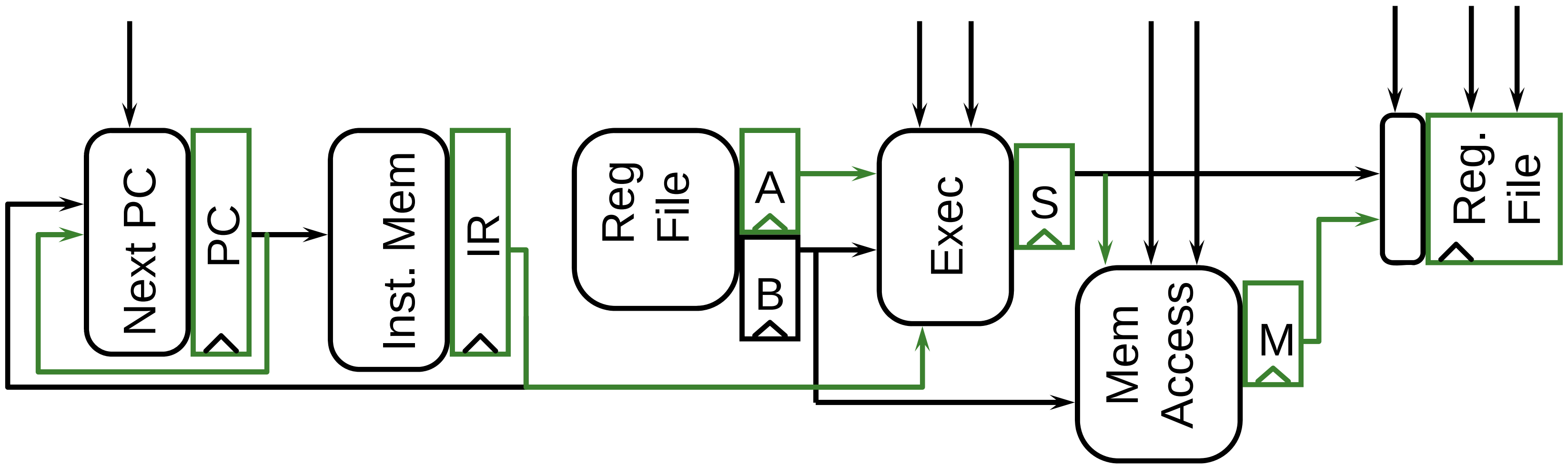
inst	**Physical Register Transfers**
	IR ← MEM[PC]
ADD	**A← R[rs]; B ← R[rt]**
	S ← A + B
	R[rd] ← S; PC ← PC + 4



Load instruction

inst	**Logical Register Transfers**
LW	**R[rt] ← MEM(R[rs] + sx(Im16);**
	PC ← PC + 4

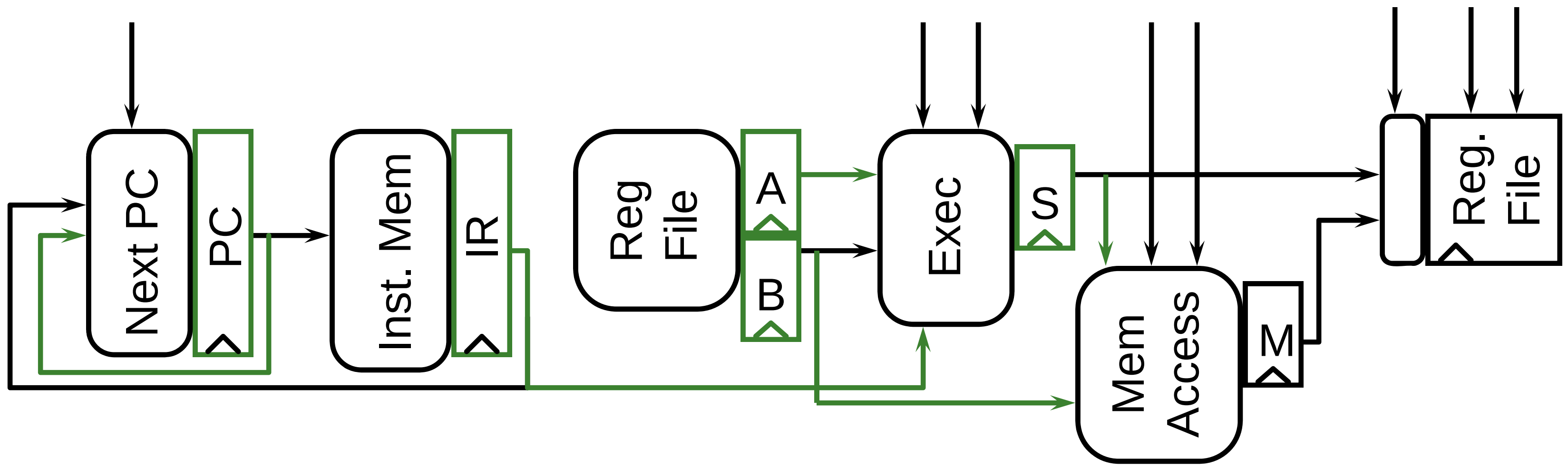
inst	**Physical Register Transfers**
	IR ← MEM[PC]
LW	**A← R[rs]; B ← R[rt]**
	S ← A + SignEx(Im16)
	M ← MEM[S]
	R[rd] ← M; PC ← PC + 4



Store instruction

inst	**Logical Register Transfers**
SW	**MEM(R[rs] + sx(Im16) ← R[rt];**
	PC ← PC + 4

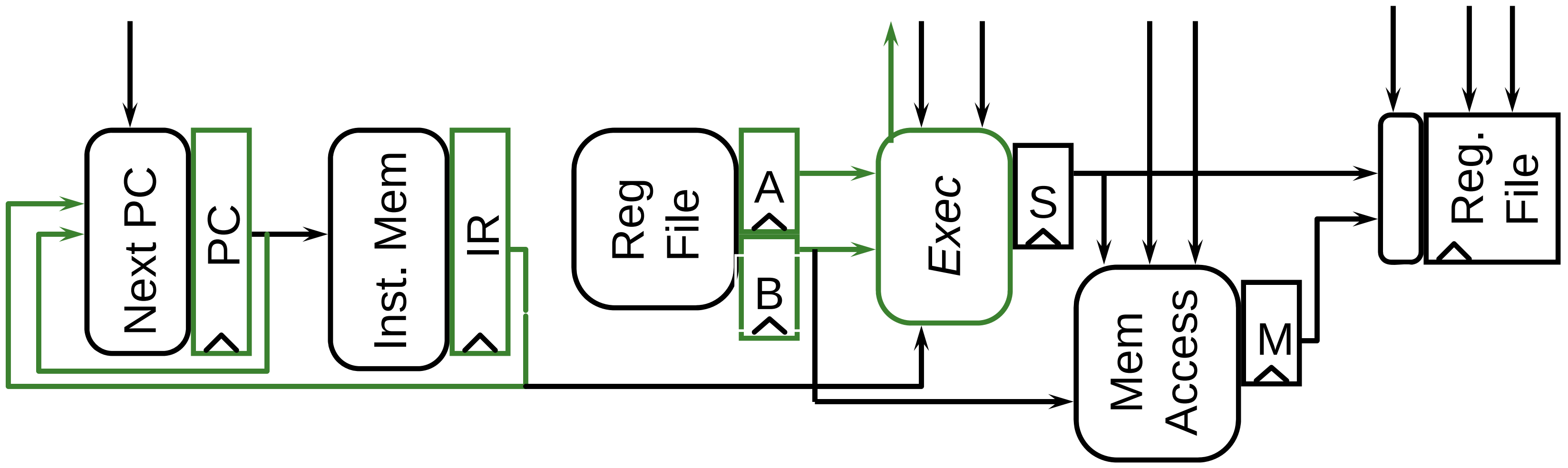
inst	**Physical Register Transfers**	
	IR ← MEM[PC]	
SW	**A ← R[rs]; B ← R[rt]**	
	S ← A + SignEx(Im16);	
	MEM[S] ← B	**PC ← PC + 4**



Branch instruction

inst	**Logical Register Transfers**
BEQ	**if R[rs] == R[rt]**
	then PC ← PC + sx(Im16) || 00
	else PC ← PC + 4

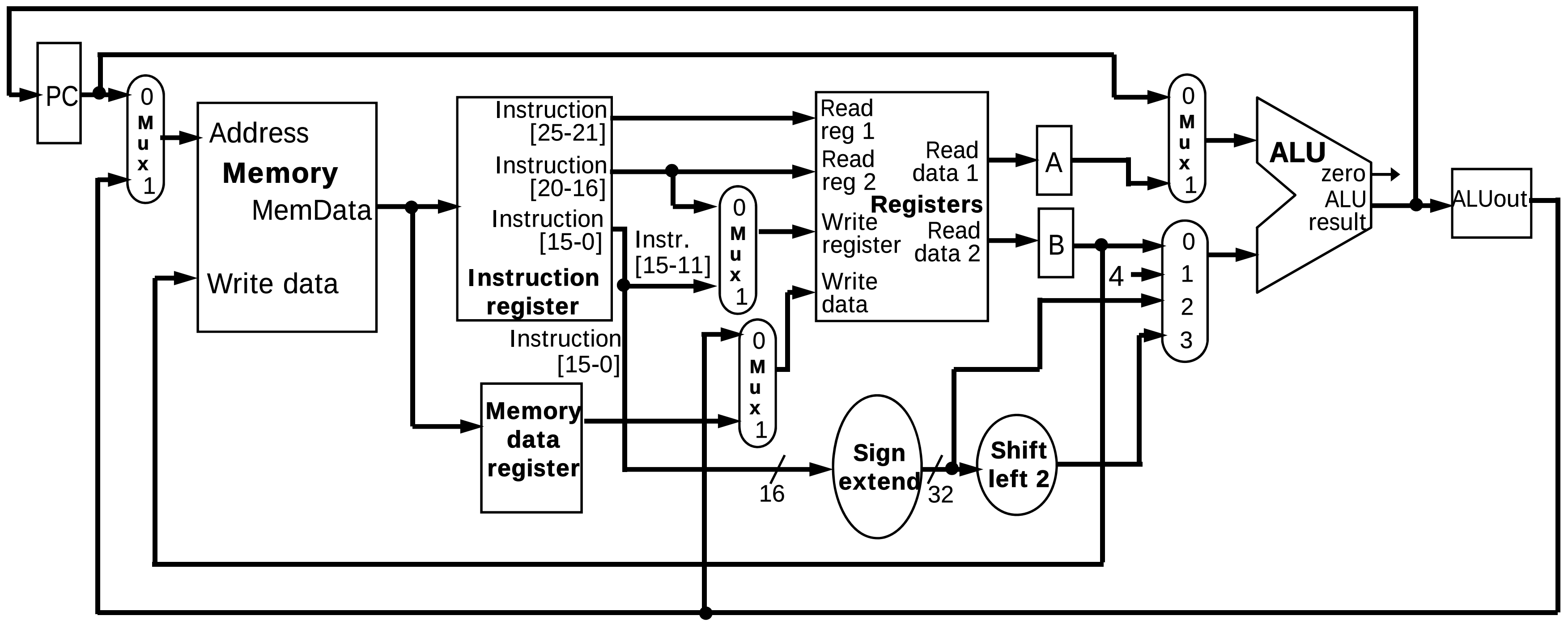
inst	**Physical Register Transfers**
	IR ← MEM[pc]
	A← R[rs]; B ← R[rt]
	Eq =(A - B = = 0)
BEQ&Eq	**PC ← PC + sx(Im16) || 00**



Multicycle Implementation

- Each step in a multicycle implementation will take 1 clock cycle.
- Multicycle implementation allows a functional unit to be used more than once per instruction, as long as it is used on different clock cycles.
- Using a functional unit more than once can help to reduce the amount of hardware required.
- The ability to allow instructions to take different number of clock cycles is another advantage of multicycle implementation.

Multicycle Datapath (Figure 5.26, p.320)



- Differences between single-cycle and multicycle datapath
  - A single memory unit is used for both instruction and data.
  - There is a single ALU, rather than an ALU and two adders.
  - One or more registers are added after every major functional unit to hold the output of that unit until the value is used in a subsequent clock cycle.

Multicycle Datapath with Control Lines

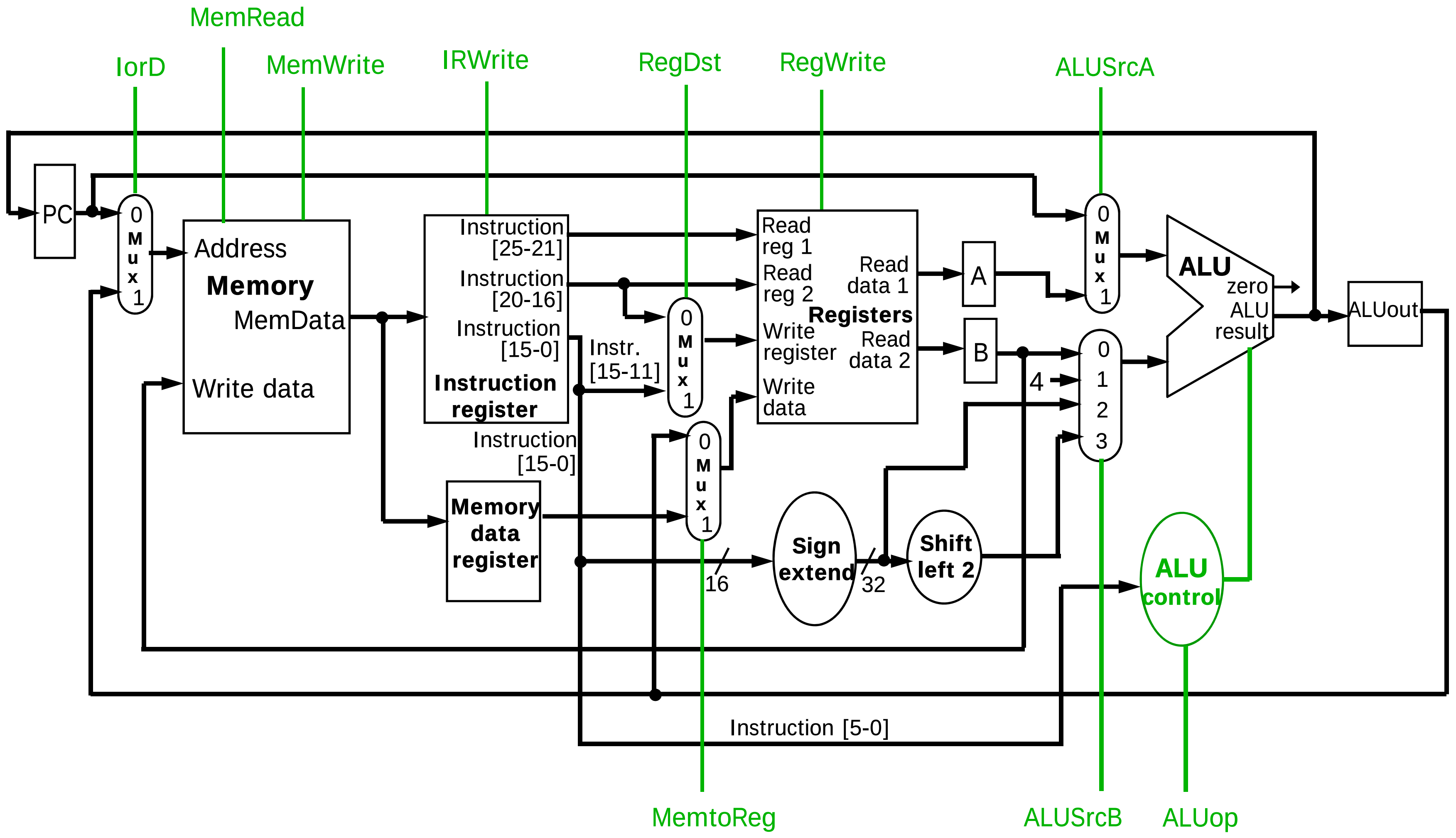
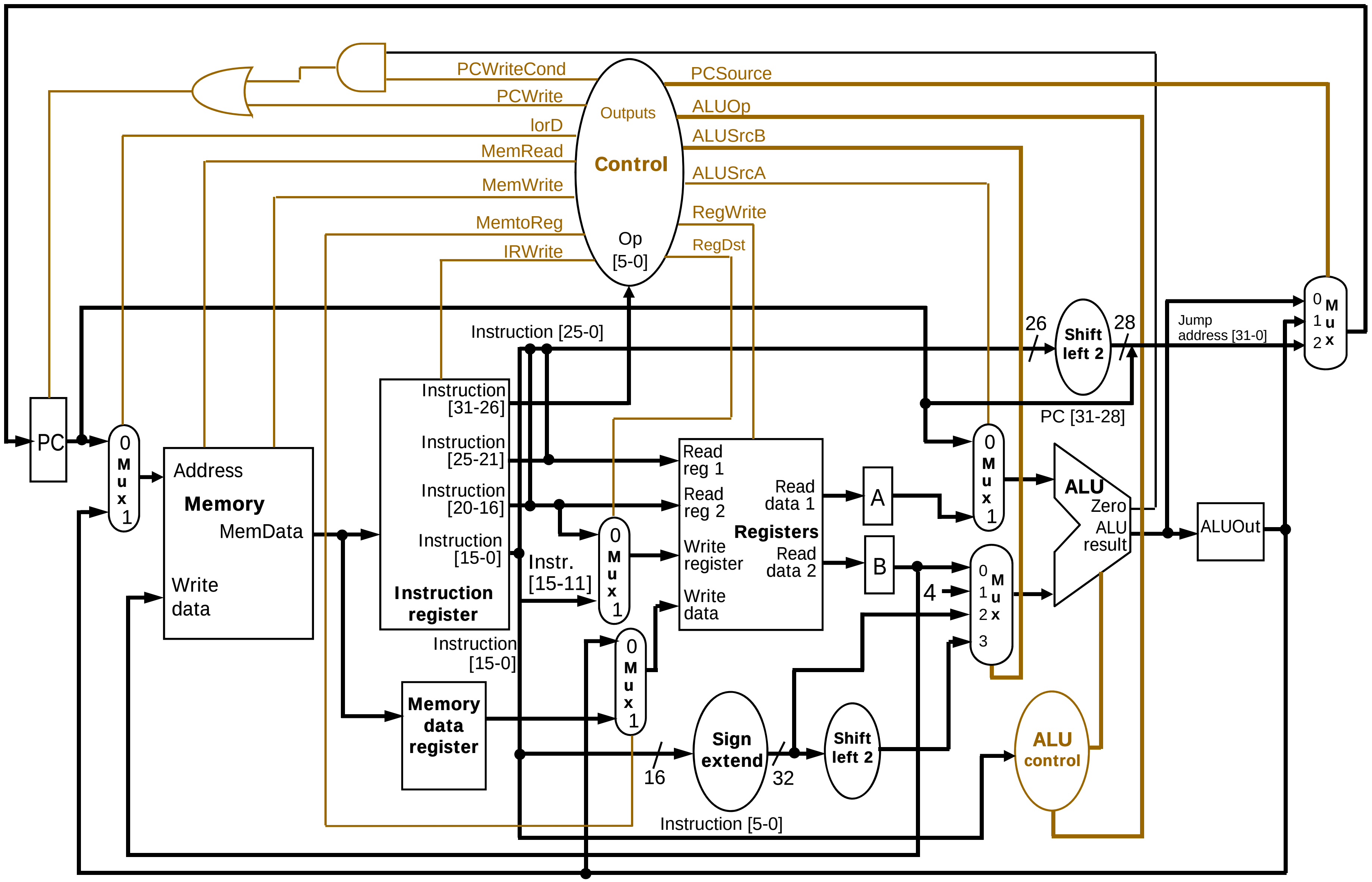


Figure 5.28: Complete Datapath & Control Signals for Multicycle Implementation (including jump instruction)



Execution Steps (1)

- Instruction Fetch

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

Execution Steps (2)

- Instruction Decode and Register Fetch

```
A = Reg[IR[25..21]];
B = Reg[IR[20..16]];
ALUOut = PC + (signExtend(IR[15..0]) << 2);
```

Execution Steps (3)

- Execution, memory address computation or branch completion
  - *Memory Reference:*

    ALUOut = A + signExtend(IR[15..0]);
  - *Arithmetic/Logical Operation:*

    ALUOut = A op B;
  - *Branch:*

    If (A == B) PC = ALUOut;
  - *Jump:*

    PC = PC[31 ..28] || (IR[25..0) << 2);

Execution Steps (4)

- Memory access or R-type instruction completion
  - *Memory Reference:*

    MDR = Memory[ALUOut];

    or

    Memory[ALUOut] = B;
  - *Arithmetic/Logical Instructions (R-type):*

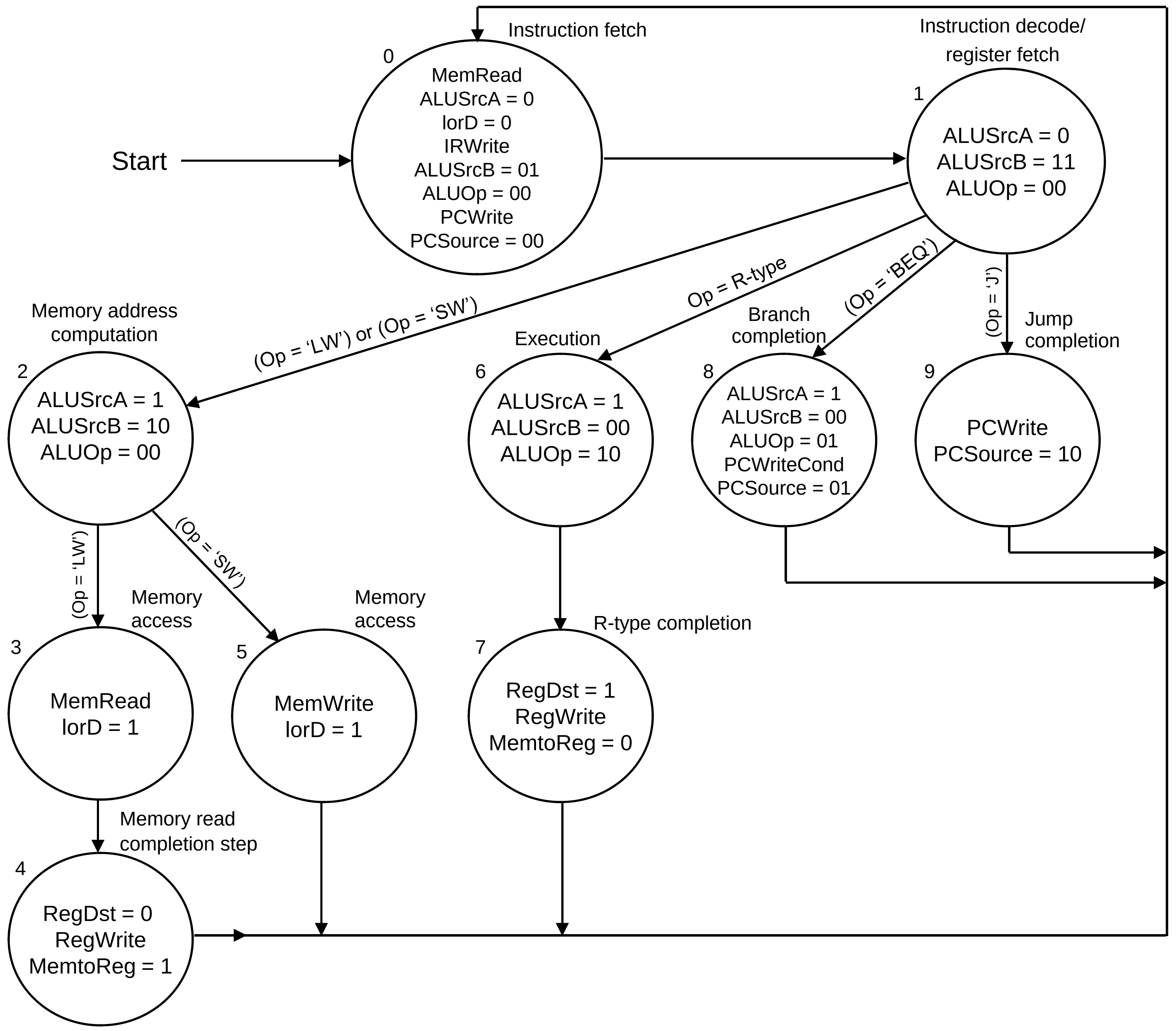
    Reg[IR[15..11]] = ALUOut;
  - *Branch, Jump:*

    Nothing

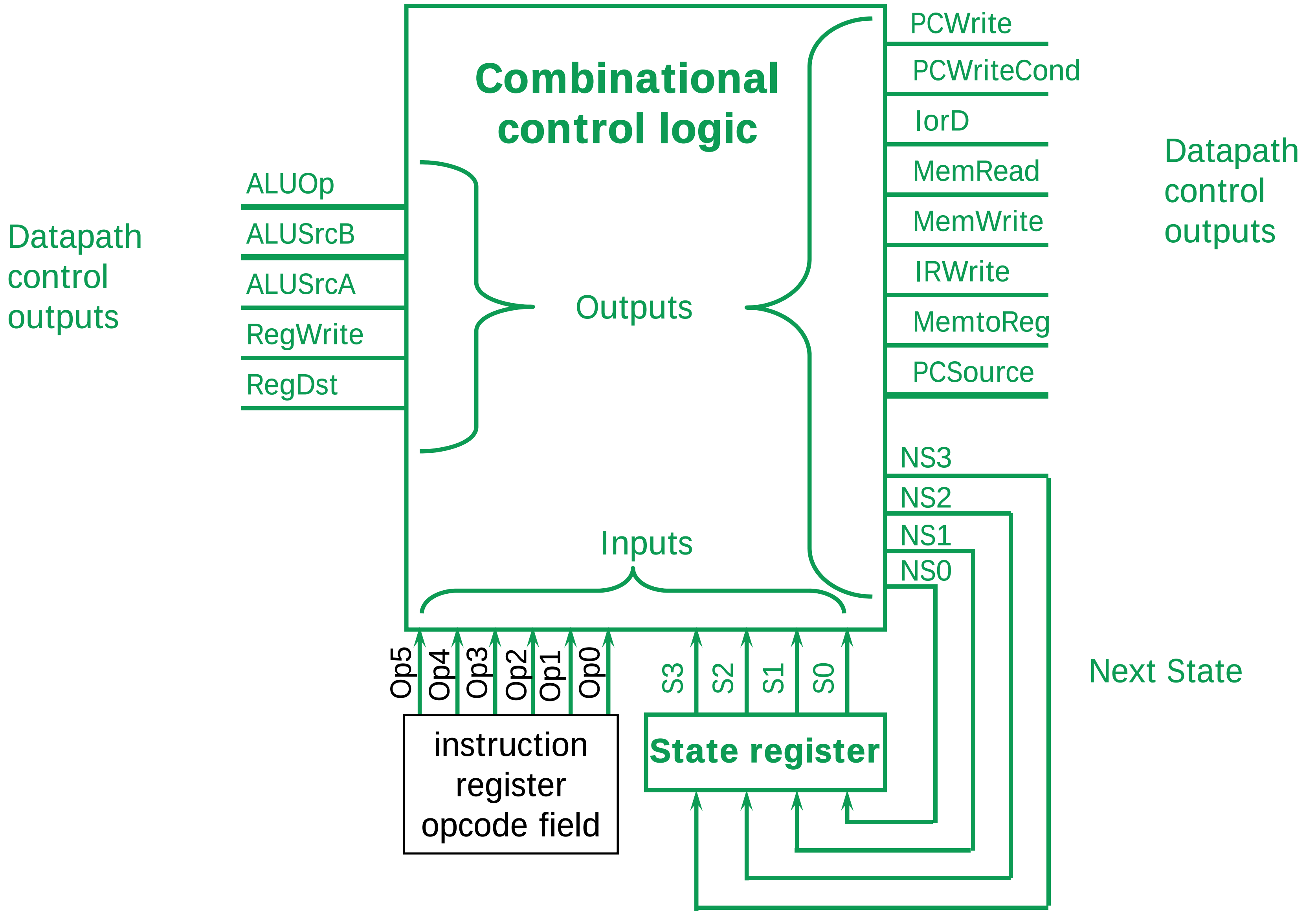
Execution Steps (5)

- Memory Read completion (Load only)
  - Reg[IR[20..16]] = MDR;

Finite State Machine Control for Multicycle Datapath



Implementation of Finite State Machine Controller



Logic Equation for Control Signal Outputs

Output	Current States
PCWrite	state0 + state9
PCWriteCond	state8
IorD	state3 + state5
MemRead	state0 + state3
MemWrite	state5
IRWrite	state0
MemtoReg	state4
PCSource1	state9
PCSource0	state8
ALUOp1	state6
ALUOp0	state8
ALUSrcB1	state1 + state2
ALUSrcB0	state0 + state1
ALUSrcA	state2 + state6 + state8
RegWrite	state4 + state7
RegDst	state7

For Example:

$$\text{PCWrite} = \overline{S3}\cdot\overline{S2}\cdot\overline{S1}\cdot\overline{S0} + S3\cdot\overline{S2}\cdot\overline{S1}\cdot S0$$

Logic Equation for Next State Outputs

Output	Current States	Op
NextState0	state4 + state5 + state7 + state8 + state9	
NextState1	state0	
NextState2	state1	(Op = ‘lw’) + (Op = ‘sw’)
NextState3	state2	(Op = ‘lw’)
NextState4	state3	
NextState5	state2	(Op = ‘sw’)
NextState6	state1	(Op = ‘R-type’)
NextState7	state6	
NextState8	state1	(Op = ‘beq’)
NextState9	state1	(Op = ‘jump’)

For Example:

$\text{NextState1} = \text{State0} = \overline{S3}\cdot\overline{S2}\cdot\overline{S1}\cdot\overline{S0}$

$\text{NextState3} = \text{State2} \cdot (\text{Op[5-0]}=\text{‘lw’})$

$= \overline{S3} \cdot \overline{S2} \cdot S1 \cdot \overline{S0} \cdot Op5 \cdot \overline{Op4} \cdot \overline{Op3} \cdot Op2 \cdot Op1$

Performance Evaluation

- What is the average CPI?
  - state diagram gives CPI for each instruction type
  - workload gives frequency of each type

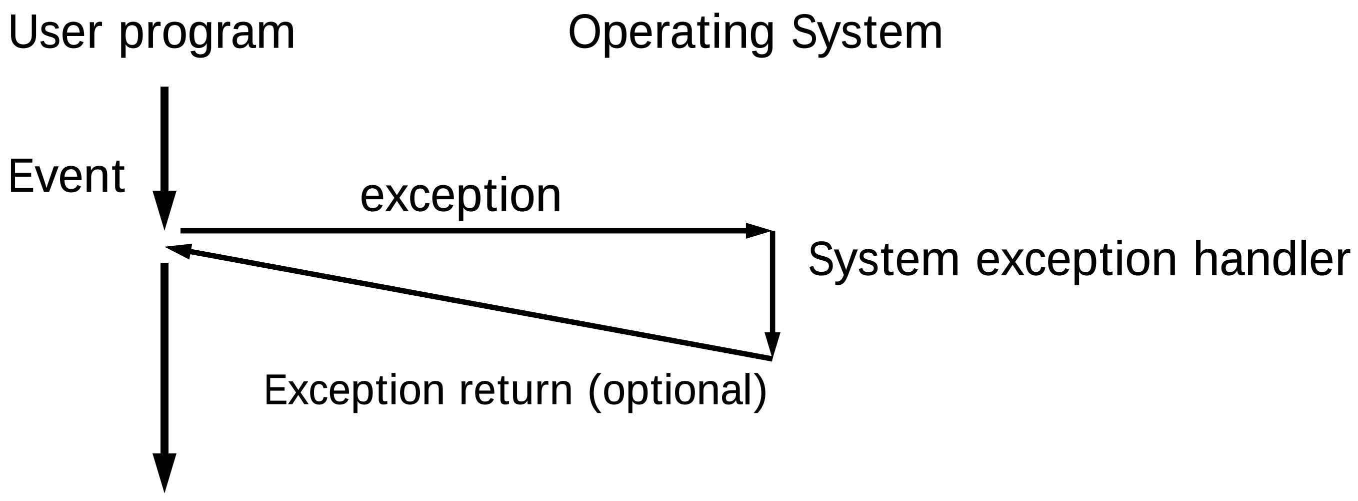
Type	CPI_i for type	Frequency	CPI_i x $freq_i$
Arith/Logic	4	40%	1.6
Load	5	30%	1.5
Store	4	10%	0.4
branch	3	20%	0.6
		Average CPI: 4.1	

Exceptions and Interrupts

- Exceptions are '**exceptional events**' that disrupt the normal flow of a program
- Terminology varies between different machines
- Examples of Interrupts
  - User hitting the keyboard
  - Disk drive asking for attention
  - Arrival of a network packet
- Examples of Exceptions
  - Divide by zero
  - Overflow
  - Invalid instruction
  - Page fault (non-resident page in memory)

Exception Flow

- When an exception (or interrupt) occurs, control is transferred to the OS



MIPS convention

- Exception means any unexpected change in control flow, without distinguishing internal or external;
- Use the term interrupt only when the event is externally caused.

Type of event	*From where?*	*MIPS terminology*
I/O device request	External	Interrupt
Invoke OS from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or Interrupt

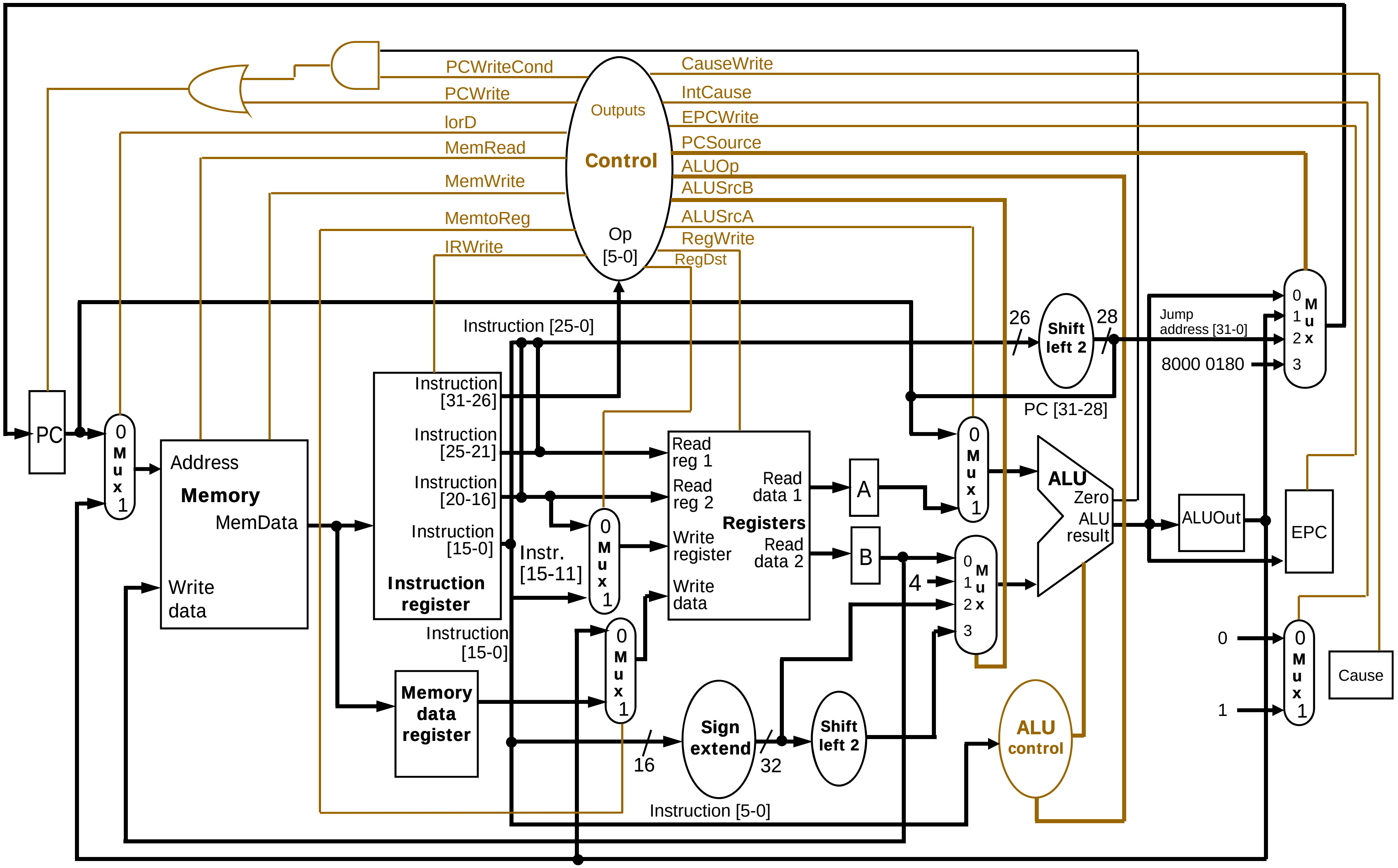
Handling Exceptions and Interrupts

- When do we jump to an exception?
- Upon detection, invoke the OS to “service the event”
  - What about in the middle of executing a multi-cycle instruction
    - Difficult to abort the middle of an instruction
  - Processor checks for event at the end of every instruction
  - Processor provides **EPC & Cause** registers to inform OS of cause
- EPC – a 32-bit register used to hold the address of the affected instruction.
- Cause – a register used to record the cause of the exception. To simplify the discussion, assume
  - undefined instruction=0
  - arithmetic overflow=1

Handling Exceptions and Interrupts

- Status - interrupt mask and enable bits and determines what exceptions can occur.
- Control signals to write EPC, Cause, and Status
- Be able to write exception address into PC, increase mux set PC to exception address ($8000\ 0180_{hex}$).
- May have to undo PC = PC + 4, since want EPC to point to offending instruction (not its successor); PC = PC – 4

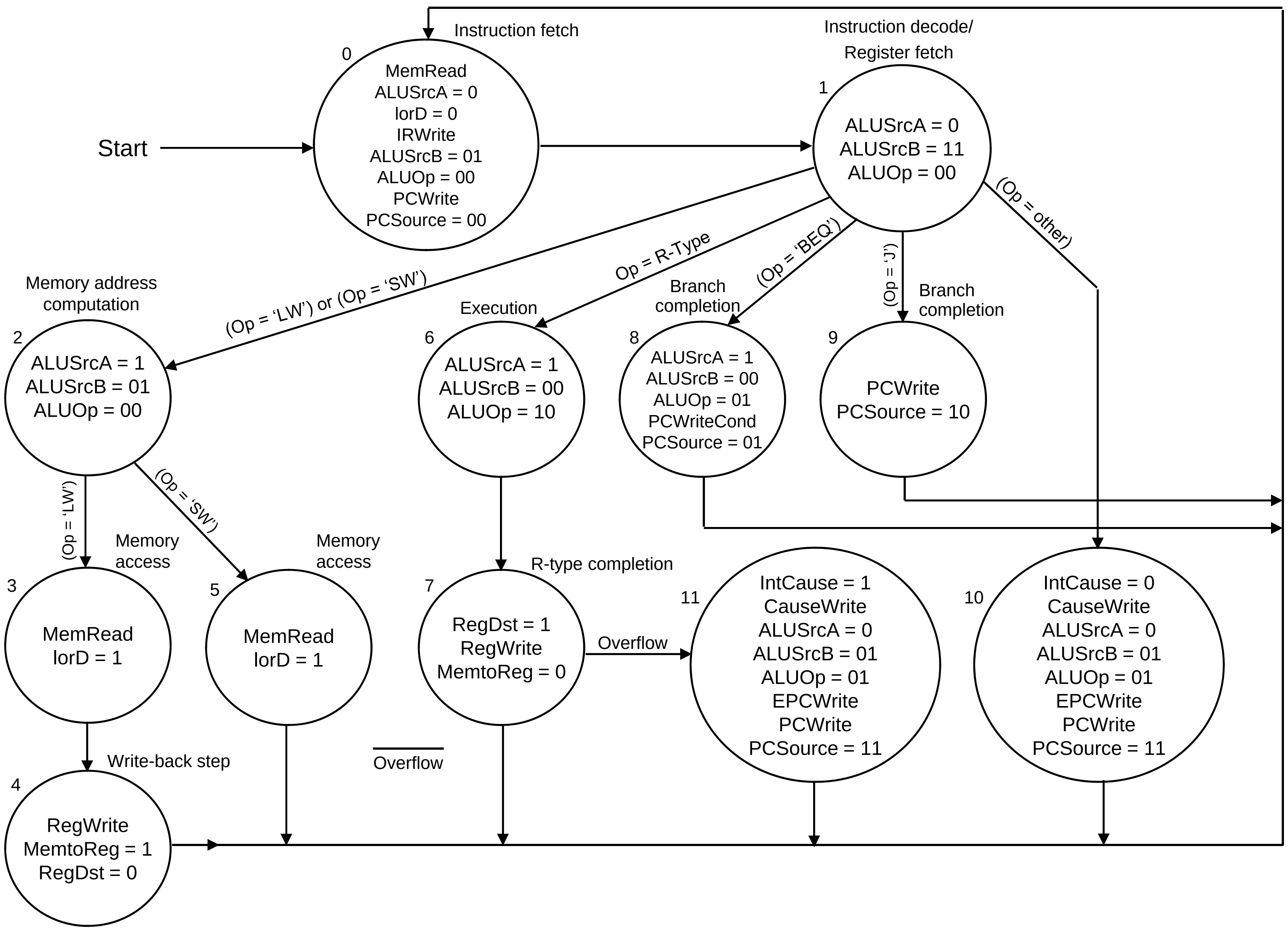
Figure 5.39: The multicycle datapath with the addition needed to implement exceptions



How Control Detects Exceptions

- **Undefined Instruction** – detected when no next state is defined from state 1 for the op value.
  - We handle this exception by defining the next state value for all op values other than lw, sw, 0 (R-type), j, and beq as new state 10.
  - Shown symbolically using “other” to indicate that the op field does not match any of the opcodes that label arcs out of state 1.
- **Arithmetic overflow** – included logic in the ALU to detect overflow, and a signal called Overflow is provided as an output from the ALU. This signal is used in the modified finite state machine to specify an additional possible next state.
- Note: Challenge in designing control of a real machine is to handle different interactions between instructions and other exception-causing events such that control logic remains small and fast.
  - Complex interactions makes the control unit the most challenging aspect of hardware design

Figure 5.40: Finite state machine to handle exception detection



Summary

- Disadvantages of the Single Cycle Processor
  - Long cycle time
  - Cycle time is too long for all instructions except the Load
- Multicycle implementations have the advantage of using a different number of cycles for executing each instruction.
- Multicycle Processor:
  - Divide the instructions into smaller steps
  - Execute each step (instead of the entire instruction) in one cycle
- Control is specified by finite state diagram (Microprogramming is used for complex instruction set)
- The most widely used machine implementation is neither single cycle, nor multicycle – it's the **pipelined** implementation (next improvement we will study).

Optional Homework

- In Ch.5: 2, 8, 10, 27, 30, 33, 36, 43