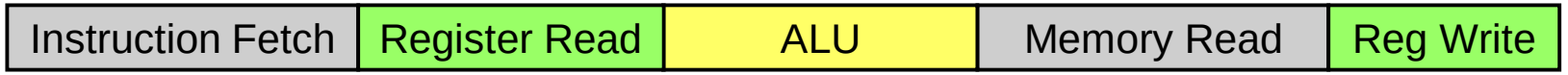
Multicycle Implementation

Drawbacks of Single Cycle Processor

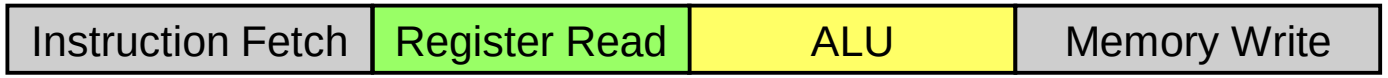
- Long cycle time
  - All instructions take as much time as the slowest

Arithmetic & Logical

Load ——— longest delay ———→



Store



Branch

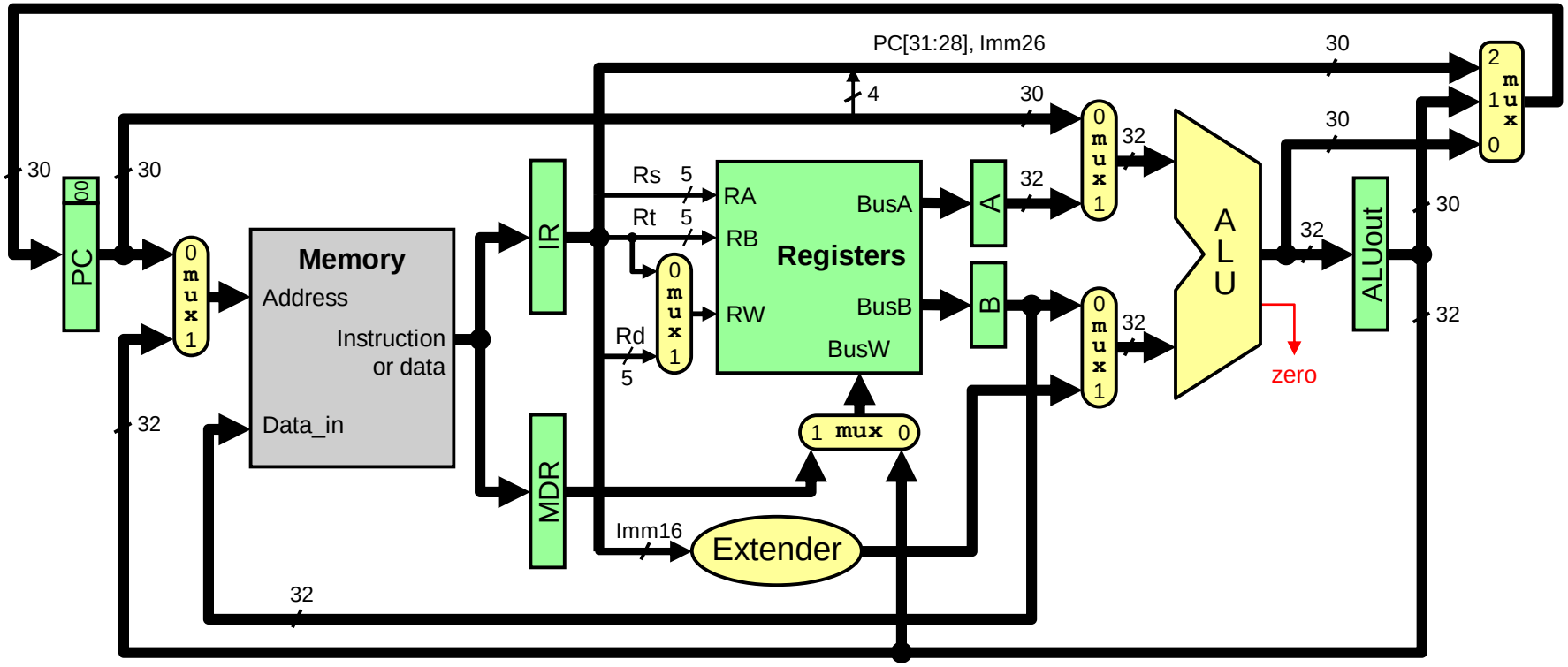
- Functional units are duplicated raising cost
  - Each functional unit can be used once per clock cycle

Solution = Multicycle Implementation

- Break instruction execution into five steps
  - Instruction fetch
  - Instruction decode and register read
  - Execution, memory address calculation, or branch completion
  - Memory access or ALU instruction completion
  - Load instruction completion
- One step = One clock cycle (clock cycle is reduced)
  - First 2 steps are the same for all instructions

Instruction	# cycles	Instruction	# cycles
ALU	4	Branch	3
Load	5	Store	4

MIPS Multicycle Datapath



Registers are used to store values at the end of each clock cycle for use during next cycle

Same memory is used for instructions and data

ALU is used to increment upper 30 bits of PC, to compute branch target and load/store address, and to execute ALU instructions

Multicycle Datapath Changes

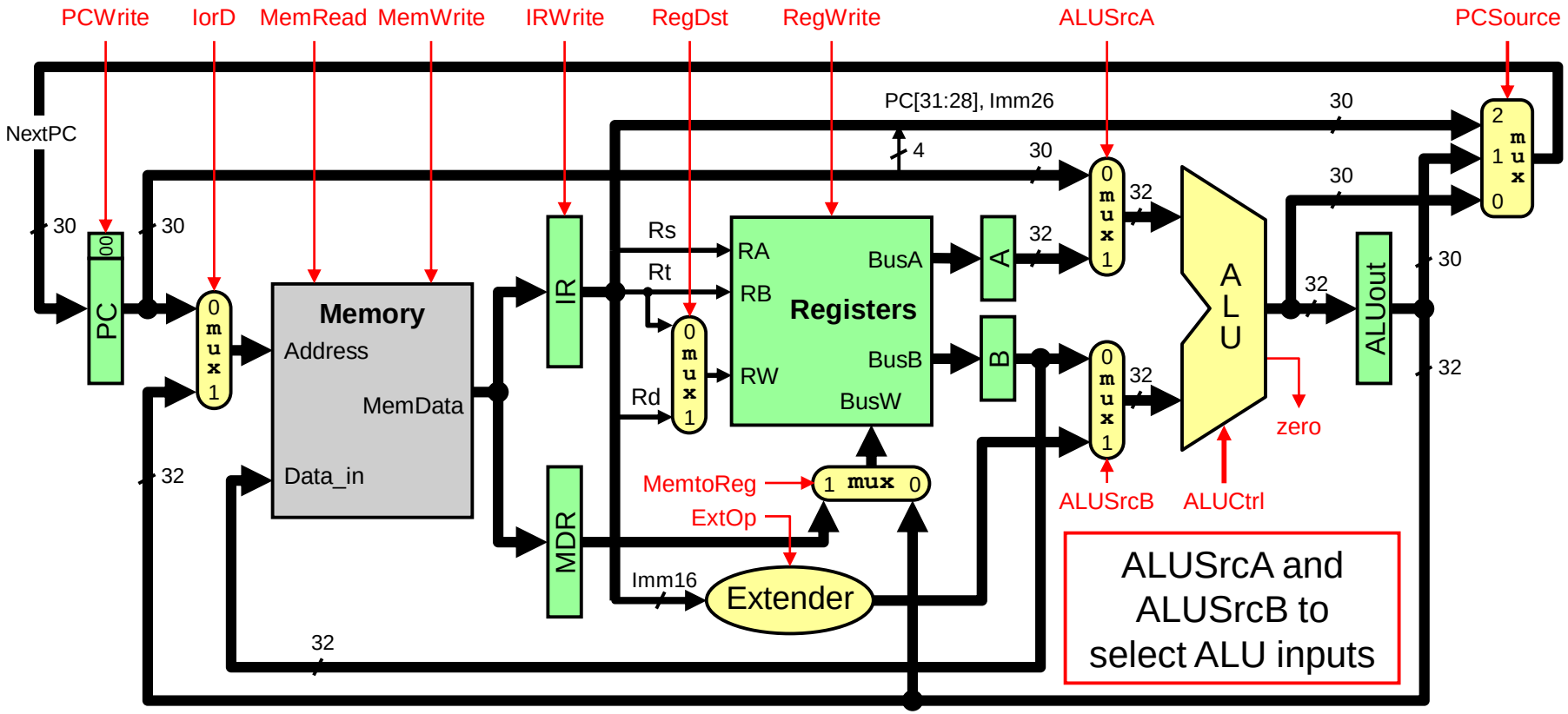
- Eliminating some of the components
  - Single memory unit for both instructions and data
  - Single ALU eliminating branch address adder and PC adder

    Note: modern CPUs maintain separate instruction and data memories as well as separate address adders, but we reduce them here because the same component can be used for different purposes in different cycles
- Adding temporary registers
  - Instruction Register: IR
  - Memory Data Register: MDR
  - Register file output data registers: A and B
  - ALU output register: ALUout
  - Required to store major unit output values for use in next cycle

Multicycle Datapath Changes – cont'd

- This multicycle design can accommodate
  - One memory access per cycle
    - IR register saves fetched instruction
    - MDR register saves the read memory data
  - One register file access per cycle
    - Two registers can be read concurrently into A and B registers
  - One ALU operation per cycle
    - ALUout register saves the ALU output
- Additional multiplexers are also needed
  - Mux before the memory address to select PC or ALUout address
  - Mux before 1st ALU input to select PC to increment or A register
  - Extended mux before PC to increment PC, branch, or jump

Multicycle Datapath + Control Signals



More control signals than single-cycle CPU

PCSource to select PC input

IorD to select memory address as either PC for instruction or ALUout for data address

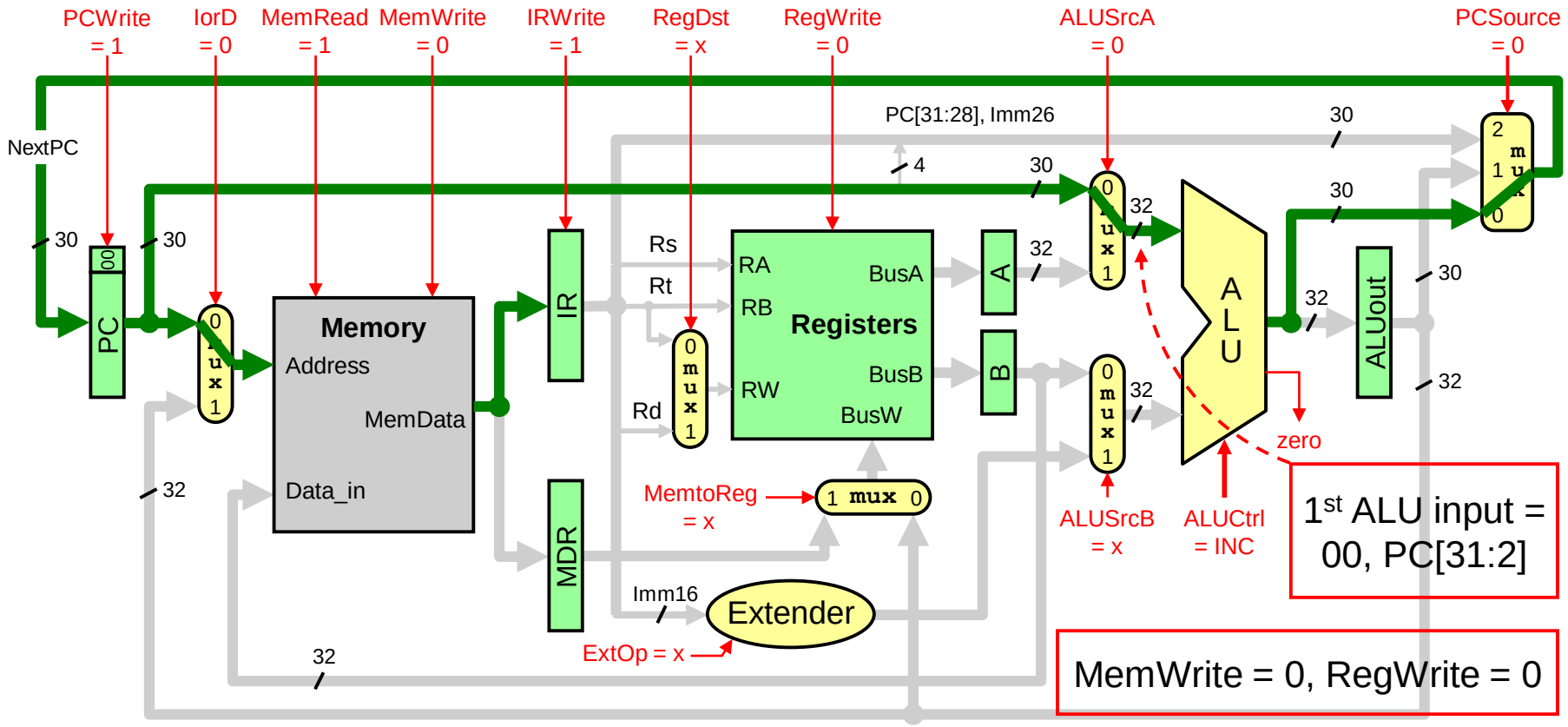
PCWrite and IRWrite to enable the writing of PC and IR registers

Control Signals

Signal	Effect when '0'	Effect when '1'
RegDst	Destination register = Rt	Destination register = Rd
RegWrite	None	Register(RW) ← BusW
ExtOp	16-bit immediate is zero-extended	16-bit immediate is sign-extended
ALUSrcA	1st ALU operand is PC (upper 30-bit)	1st ALU operand is the A register
ALUSrcB	2nd ALU operand is the B register	2nd ALU input is extended-imm16
MemRead	None	MemData ← Memory[address]
MemWrite	None	Memory[address] ← Data_in
MemtoReg	BusW = ALUout	BusW = MDR
IorD	Memory Address = PC	Memory Address = ALUout
IRWrite	None	IR ← MemData
PCWrite	None	PC ← NextPC

Signal	Value	Effect
PCSource	00	NextPC = PC[31:2] + 1 (increment upper 30 bits of PC)
	01	NextPC = ALUout = PC[31:2] + 1 + sign-extend(imm16) (for branch)
	10	NextPC = PC[31:28], imm26 (for jump)

1. Instruction Fetch Cycle



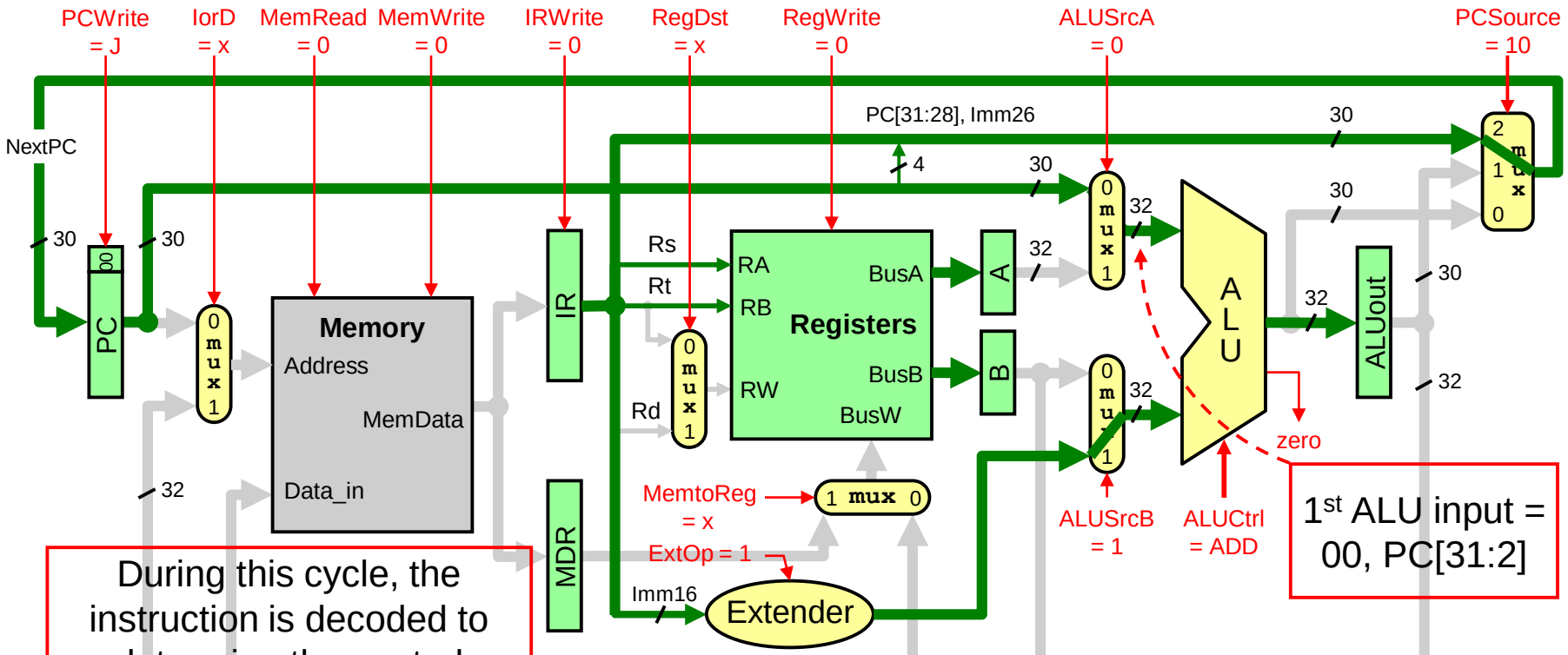
IR ← Memory[PC]
PC ← PC + 4

IorD = 0
MemRead = 1
IRWrite = 1

ALUSrcA = 0, ALU = INC
PCSource = 0, PCWrite = 1

Don't care about rest

2. Decode and Register Fetch



During this cycle, the instruction is decoded to determine the control signals for the next cycles

1st ALU input = 00, PC[31:2]

MemRead = MemWrite = IRWrite = RegWrite = 0

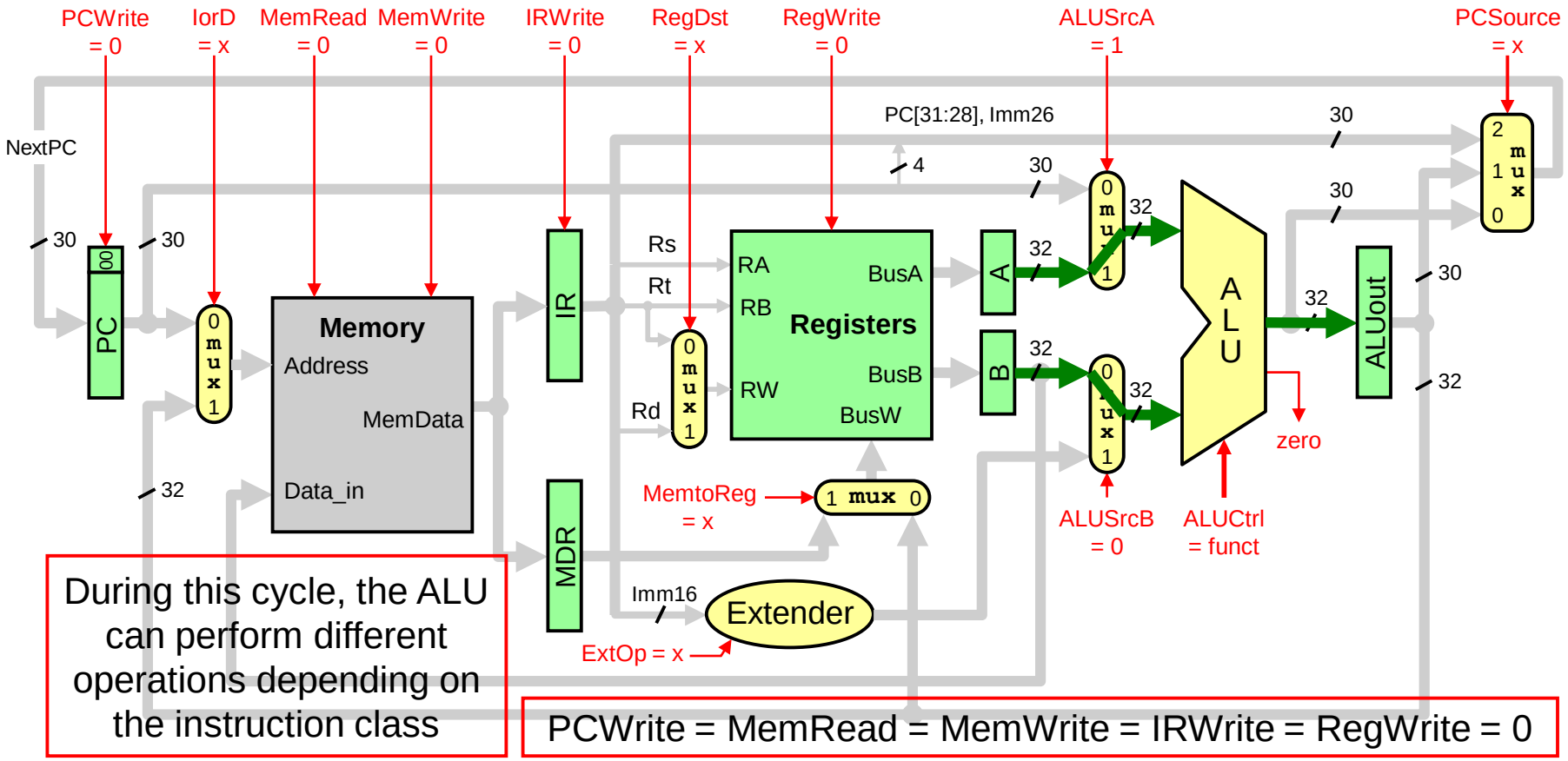
A ← Reg[Rs], B ← Reg[Rt]
A, B are written on every cycle

ALUout ← PC[31:2] + sign-ext(Im16) (branch address)
ALUSrcA = 0, ALUSrcB = 1, ExtOp = 1, ALU = ADD

Compute branch address in advance

PCSource = 10, PCWrite = J (Jump Completion)

3a. Execute Cycle for R-type



During this cycle, the ALU can perform different operations depending on the instruction class

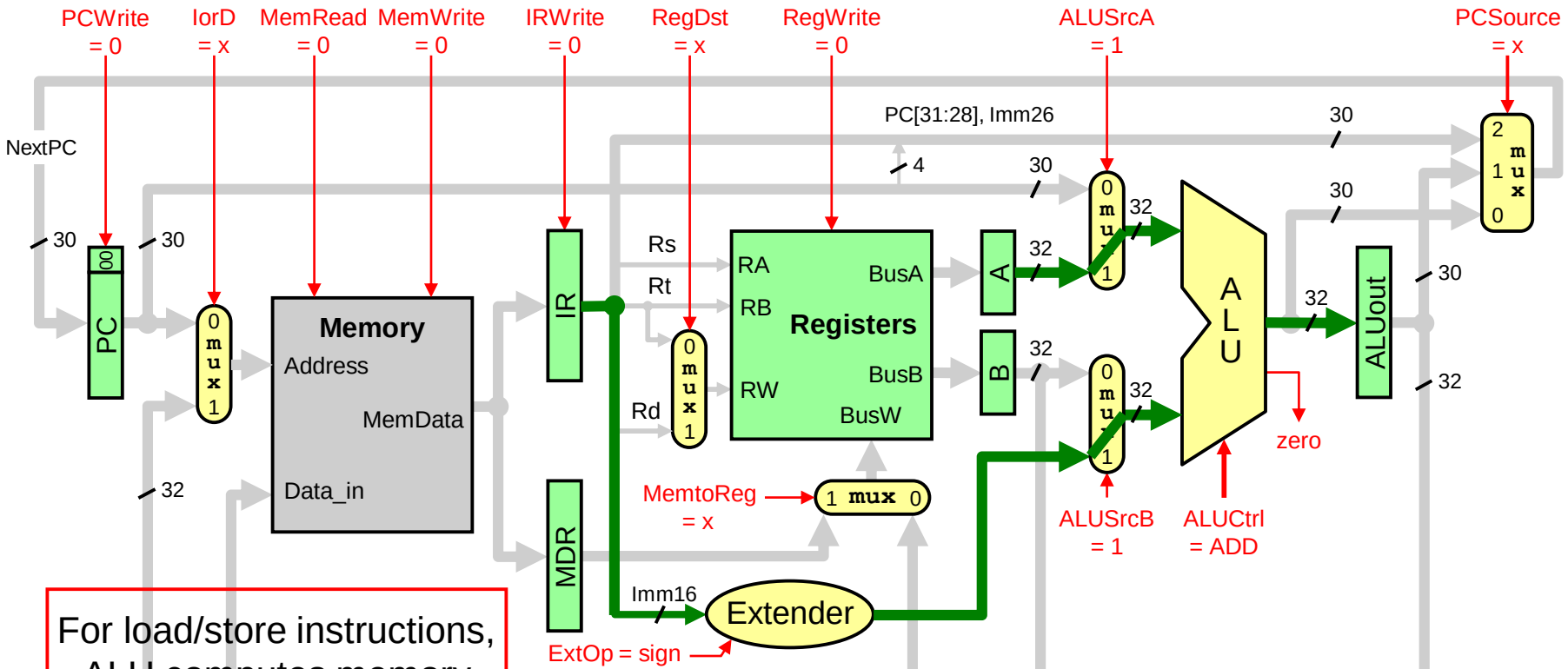
PCWrite = MemRead = MemWrite = IRWrite = RegWrite = 0

For R-type ALU instructions:
ALUout ← A funct B
ALUCtrl depends on the function field

ALUSrcA = 1,
ALUSrcB = 0,
ALU = funct

Don't care about rest

3b. Compute Address for Load/Store



For load/store instructions, ALU computes memory address during 3rd cycle

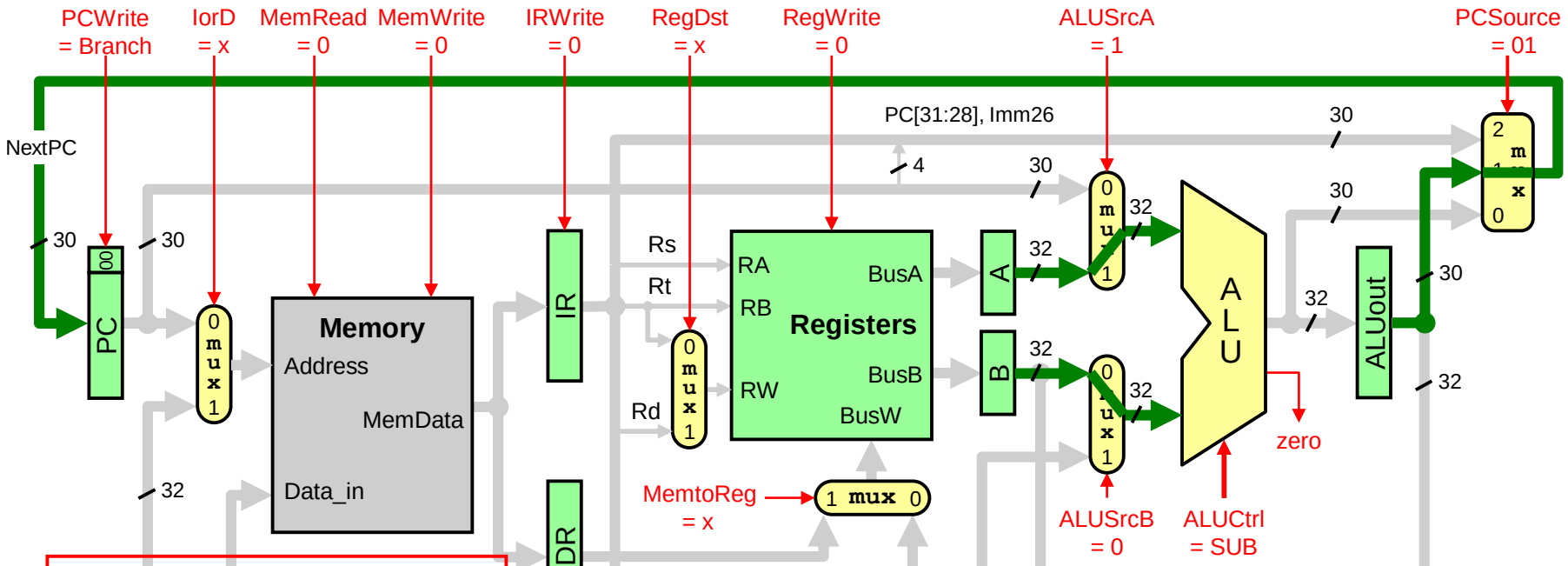
PCWrite = MemRead = MemWrite = IRWrite = RegWrite = 0

ALUout ← A + sign-extend(Immediate16)

Same control signals can be used with I-type ALU

ALUSrcA = 1, ALUSrcB = 1
ExtOp = sign, ALU = ADD

3c. Branch Completion



For a branch, ALU compares A with B and if the branch is taken then PC becomes ALUout

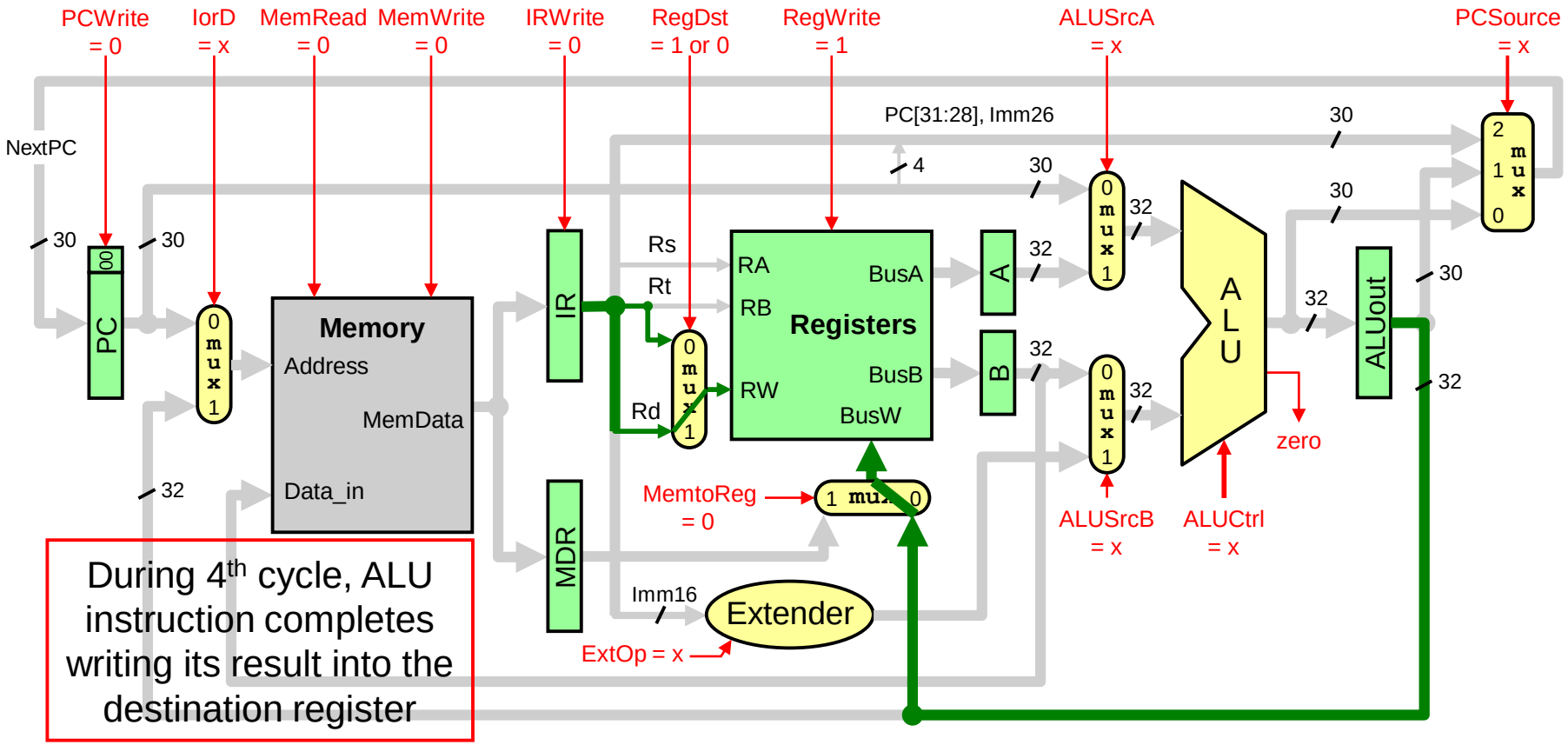
Branch depends on the zero condition
Branch = beq . zero + bne . $\overline{\text{zero}}$

MemRead = MemWrite = IRWrite = RegWrite = 0

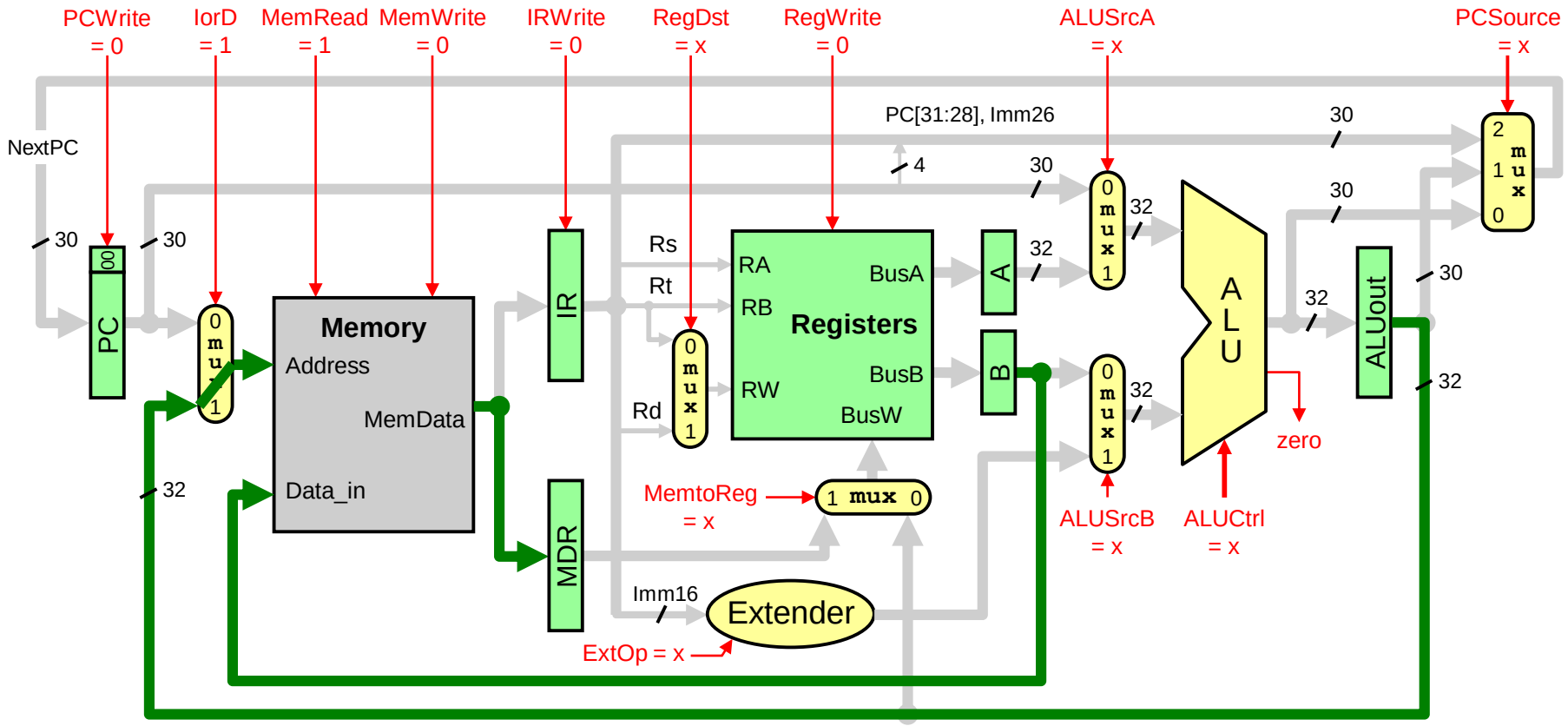
if (branch) PC ← ALUout
ALUout is branch target address computed during the second cycle

ALUSrcA = 1,
ALUSrcB = 0, PCSource = 01
ALUCtrl = SUB, PCWrite = Branch

4a. ALU Instruction Completion



4b. Memory Access for Load & Store



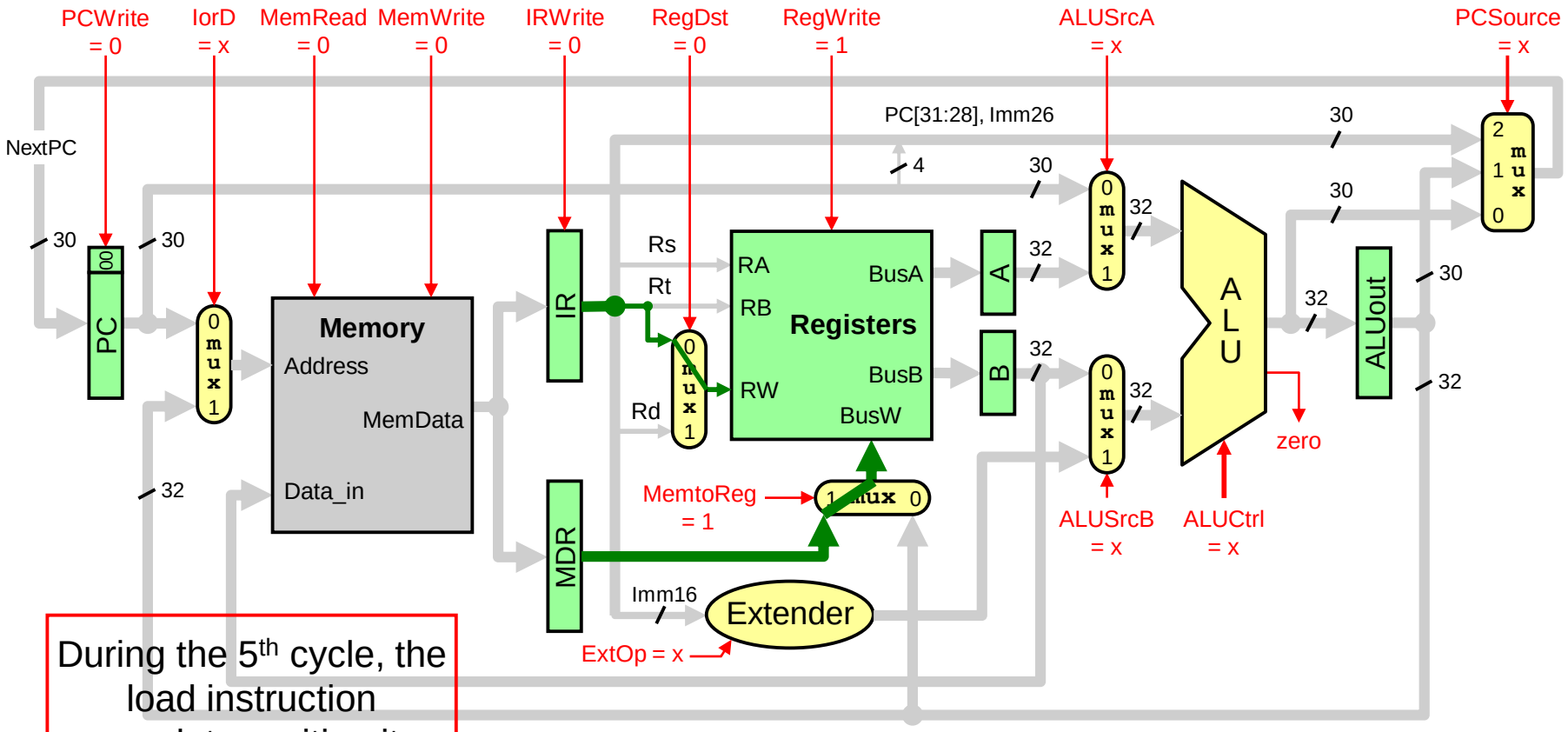
Load & store access memory during 4th cycle

IorD = 1, MemRead = 1 (load)
MemWrite = 1 (store)

MDR ← Memory[ALUout] (for load)
Memory[ALUout] ← B (for store)

PCWrite = IRWrite = RegWrite = 0

5. Load Instruction Completion



During the 5th cycle, the load instruction completes writing its result into register Rt

Reg[Rt] ← MDR

RegDst = 0 (Rt)
MemtoReg = 1
RegWrite = 1

PCWrite = IRWrite = 0,
MemRead = MemWrite = 0
Don't care about rest

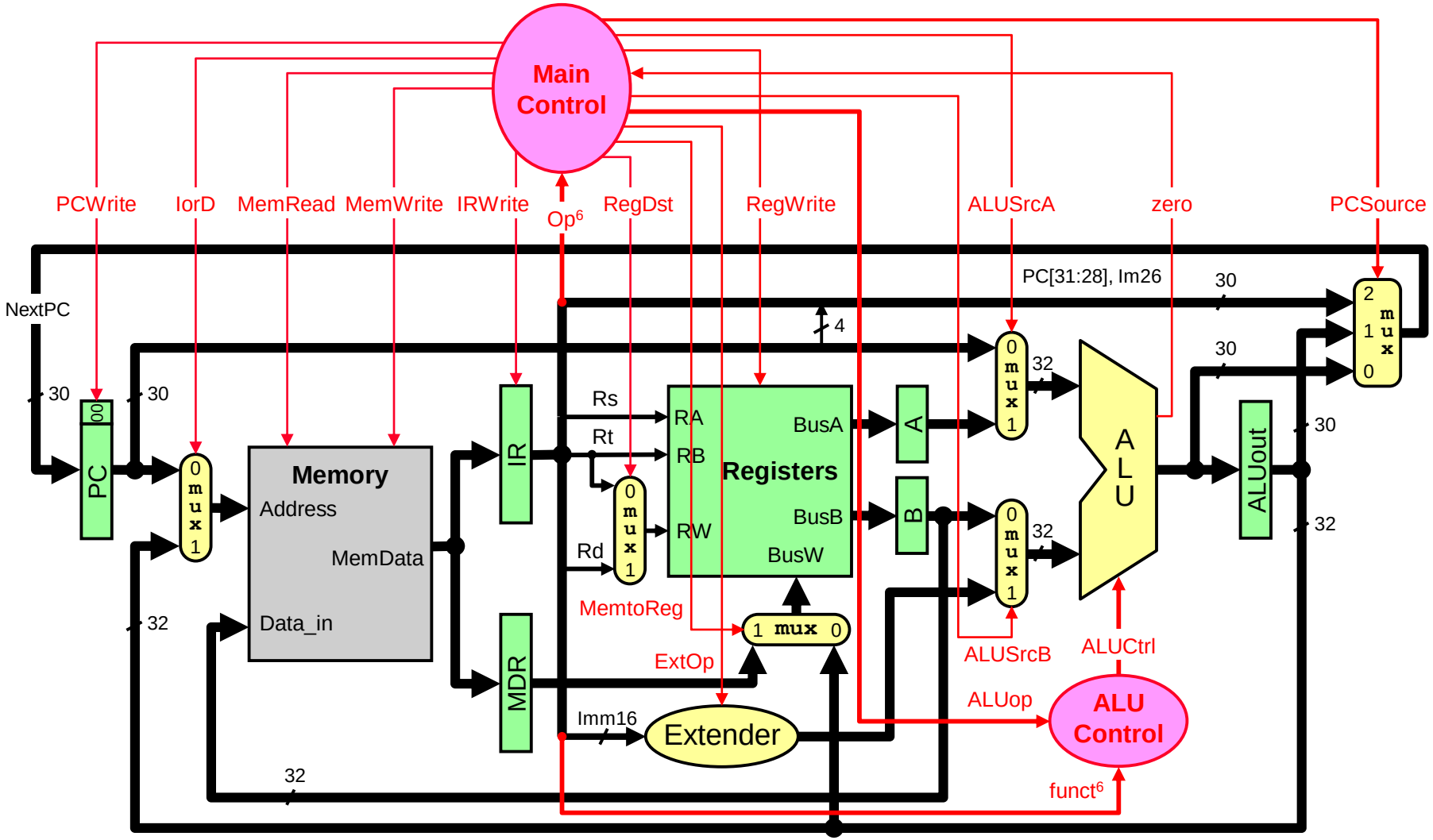
Instruction Execution Summary

Cycle	Action	Register Transfers
1	Fetch instruction	IR ← Memory[PC] , PC ← PC + 4
2	Decode instruction Fetch registers Compute branch address in advance Jump completion (case of a jump)	Generate control signals A ← Reg[Rs], B ← Reg[Rt] ALUout ← PC[31:2] + sign-extend(Imm16) PC ← PC[31:28], Im26
3	Case 1: Execute R-type ALU Case 2: Execute I-type ALU Case 3: Compute load/store address Case 4: Branch completion	ALUout ← A funct B ALUout ← A op extend(Imm16) ALUout ← A + sign-extend(Imm16) if (Branch) PC ← ALUout
4	Case 1: Write ALU result for R-type Case 2: Write ALU result for I-type Case 3: Access memory for load Case 4: Access memory for store	Reg[Rd] ← ALUout Reg[Rt] ← ALUout MDR ← Memory[ALUout] Memory[ALUout] ← B
5	Load instruction completion	Reg[Rt] ← MDR

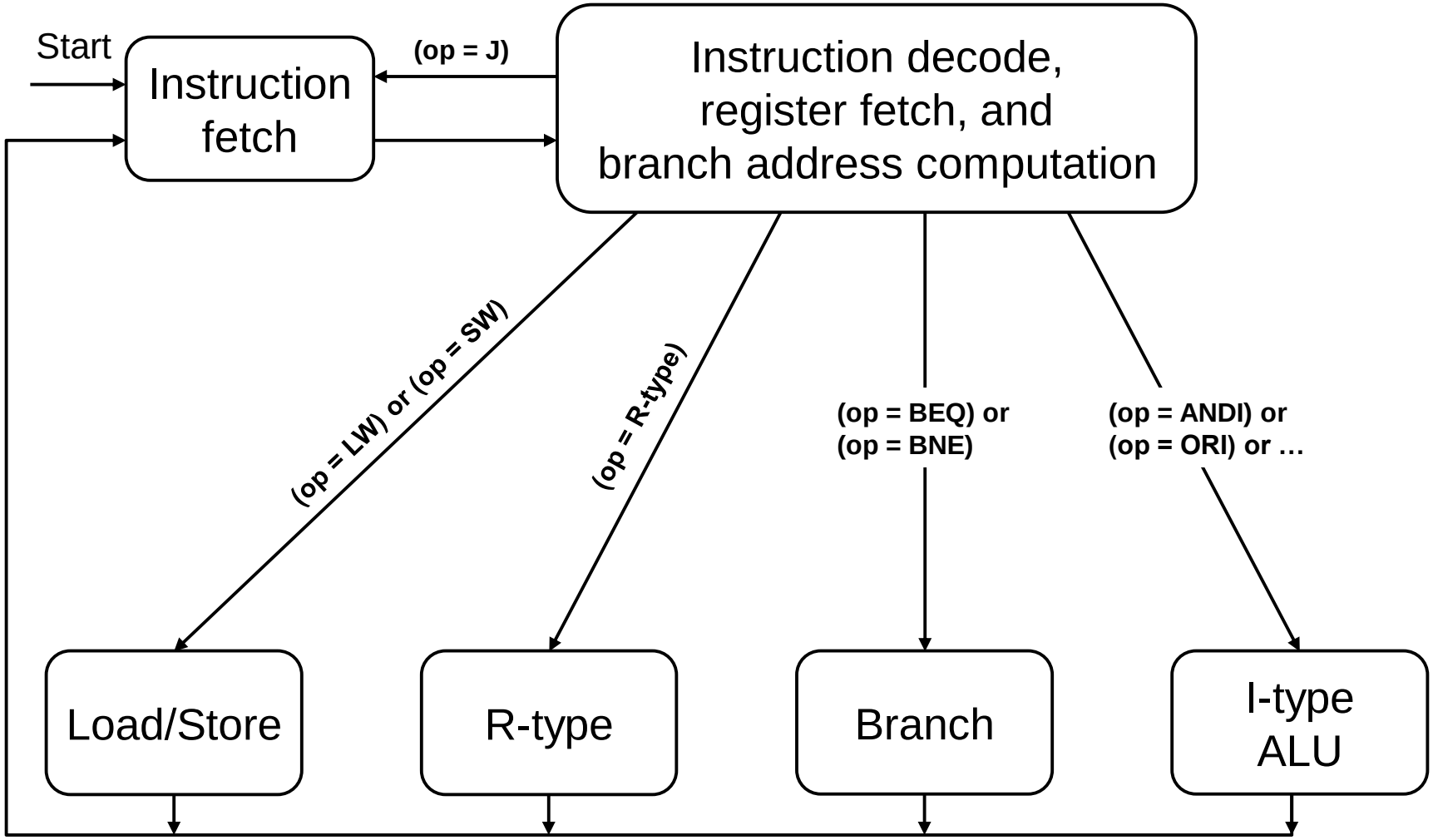
Defining the Control

- Control for multicycle datapath is more complex
  - Because instruction is executed as a sequence of steps
- Values of control signals depend upon:
  - What instruction is being executed
  - Which cycle is being performed
- Multicycle control is a Finite State Machine (FSM)
  - While single-cycle control is a combinational logic
- Two implementation techniques for multicycle control
  - Set of states and transitions implemented directly in logic
  - Microprogramming: a programming representation for control

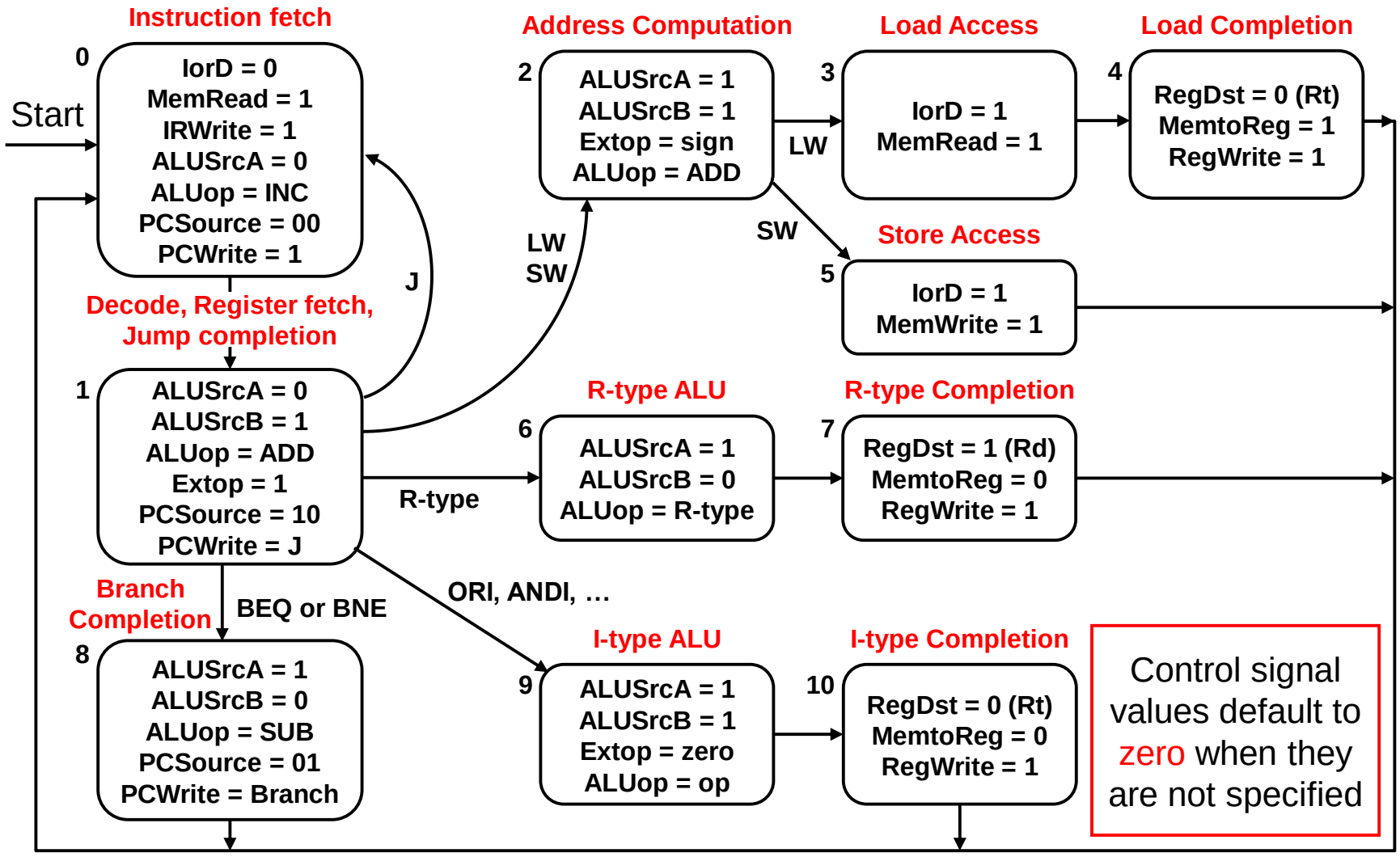
Multicycle Datapath + Control



High Level View of FSM Control

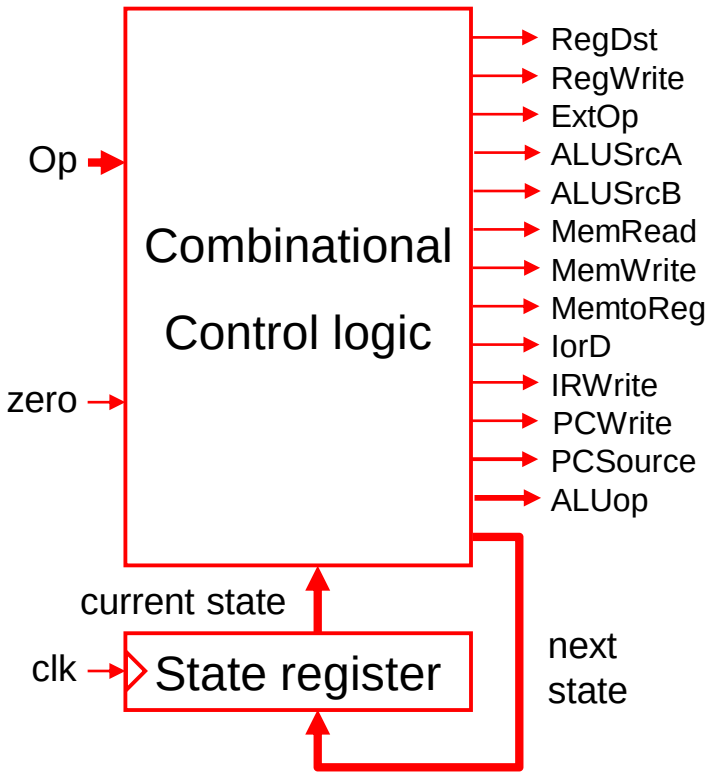


State Diagram for Multicycle Control



Finite State Machine Controller

- Implemented as …
  - Comb control logic
  - State register

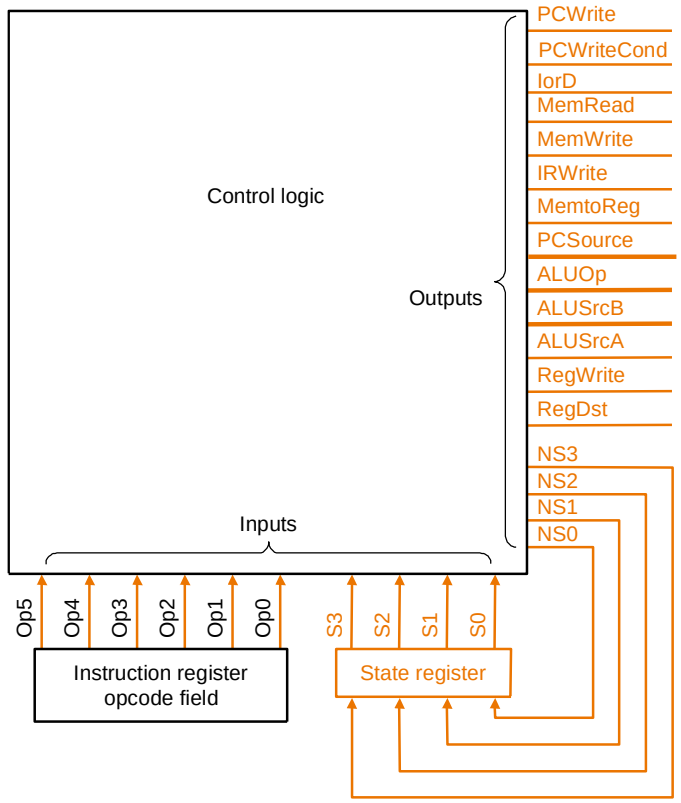


State Transition and Output Table

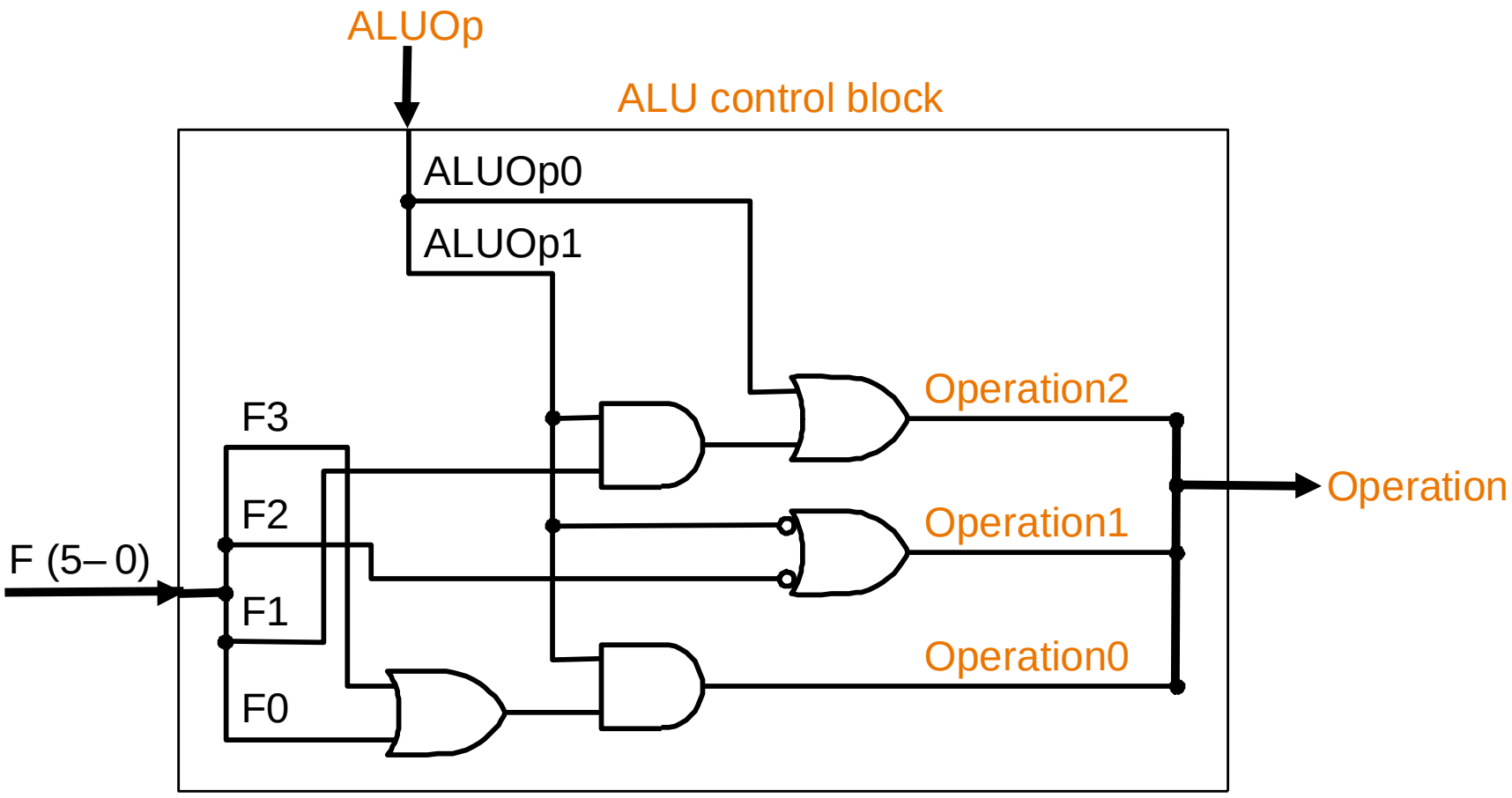
current state	Op	zero	next state	RegDst	ExtOp	RegWrite	ALUSrcA	ALUSrcB	MemRead	MemWrite	MemtoReg	IorD	IRWrite	PCWrite	PCSource	ALUop
0	x	x	1	x	x	0	0	x	1	0	x	0	1	1	00	INC
1	lw, sw	x	2	x	1	0	0	1	0	0	x	x	0	0	10	ADD
1	Rtype	x	6	x	1	0	0	1	0	0	x	x	0	0	10	ADD
1	beq bne	x	8	x	1	0	0	1	0	0	x	x	0	0	10	ADD
1	j	x	0	x	1	0	0	1	0	0	x	x	0	1	10	ADD
1	ori, …	x	9	x	1	0	0	1	0	0	x	x	0	0	10	ADD
2	lw	x	3	x	1	0	1	1	0	0	x	x	0	0	x	ADD
2	sw	x	5	x	1	0	1	1	0	0	x	x	0	0	x	ADD
3	x	x	4	x	x	0	x	x	1	0	x	1	0	0	x	x
4	x	x	0	0	x	1	x	x	0	0	1	x	0	0	x	x
5	x	x	0	x	x	0	x	x	0	1	x	1	0	0	x	x
6	x	x	7	x	x	0	1	0	0	0	x	x	0	0	x	Rtype
7	x	x	0	1	x	1	x	x	0	0	0	x	0	0	x	x
8	bne beq	0 1	0	x	x	0	1	0	0	0	x	x	0	Br	01	SUB
9	x	x	10	x	0	0	1	1	0	0	x	x	0	0	x	Op
10	x	x	0	0	x	1	x	x	0	0	0	x	0	0	x	x

Finite State Machine for Control

- Implementation:

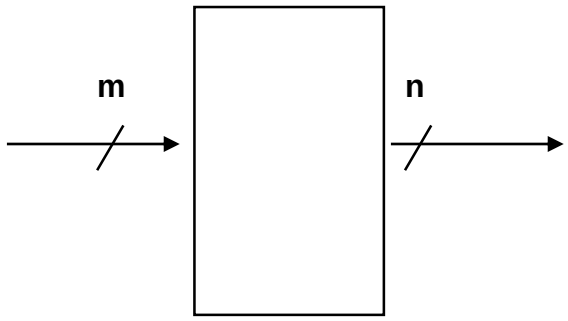


ALU Control Block



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

m is the "heigth", and n is the "width"

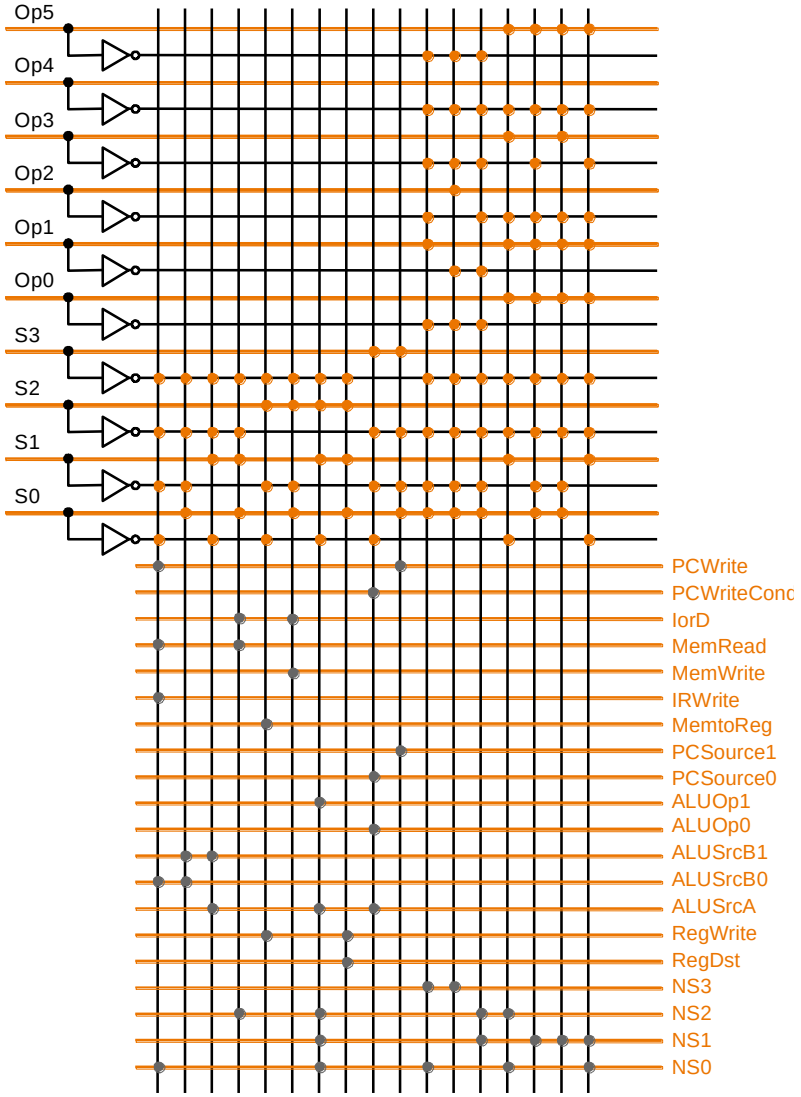
ROM Implementation

- 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
- Rather wasteful, since for lots of the entries, the outputs are the same
  — i.e., opcode is often ignored

PLA Implementation

- Programmable Logic Array



Performance multi cycle

Assume the following latencies for the data path components.
Memory : 20ns. Register File: 12ns, ALU: 16 ns, Temp. Reg: 2ns.
MUX time and all other components is 0ns.
Assume a program has a $2x10^8$ and frequencies of instruction types as follows: ALU 50%, Load 20%, Store 10% and Branch 20%.
Compare processor performance assume single cycle and multicycle data paths.
CPI single cycle is 1 and Cycle Time is (LW) 2 x 20 + 2 x 12 + 16 = 80 ns.
Average CPI multi cycle is = . 5x 4 + .2 x 5 + .1 x 4 + .2 x 3 = 4
Cycle time multicycle is Max (20, 12, 16, 20, 12) + 2 = 22 ns.
ET single cycle = $2 \times 10^8 \times 1 \times 80 \times 10^{-9}$ = 16 seconds.
ET multicycle = $2 \times 10^8 \times 4 \times 22 \times 10^{-9}$ = 17.6 seconds.
Which is faster why? Multi cycle is slower due the large variations in stage times.
The stage times must be chosen to have equal or very close latencies to each others
80/5 = 16 ns per stage and 2 ns for temp registers. Then ideal cycle time is 18ns.
ET multicycle = $2 \times 10^8 \times 4 \times 18 \times 10^{-9}$ = 14.4 seconds.
Speedup = 16/14.4 = 1.11 **11% faster multicycle than single cycle.**

Multicycle Implementation Summary

- Reduces hardware
  - One unified memory for instruction and data, and one ALU
- Breaks instruction execution into steps (step = 1 cycle)
- Internal registers in data path
  - Save intermediate data for later cycles
- Finite State Machine (FSM) specification of control
- Implementation of control
  - Hardwired control ⇒ a sequential machine
  - Microprogramming (See textbook)
- Reduces clock cycle and time

Cycle time = Maximum delay due to any stage + Delay for writing state registers

When compared to single-cycle implementation (Maximum instruction delay)