EE 457 Unit 5

Single-Cycle CPU
Datapath and Control

CPU Organization Scope

- We will build a CPU to implement our subset of the MIPS ISA
  - Memory Reference Instructions:
    - Load Word (LW)
    - Store Word (SW)
  - Arithmetic and Logic Instructions:
    - ADD, SUB, AND, OR, SLT
  - Branch and Jump Instructions:
    - Branch if equal (BEQ)
    - Jump unconditional (J)
- These basic instructions exercise a majority of the necessary datapath and control logic for a more complete implementation

CPU Implementations

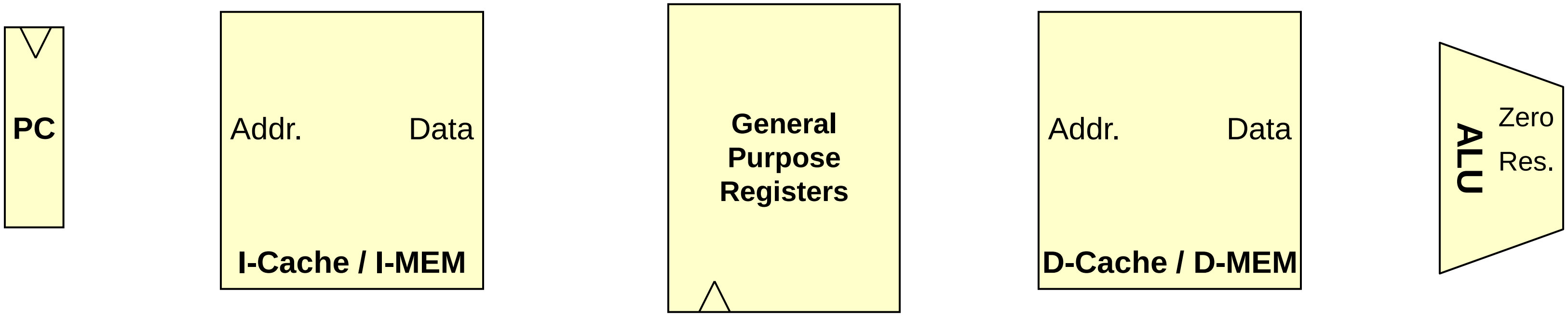
- We will go through two implementations
  - Single-cycle CPU (CPI = 1)
    - All instructions execute in a single, long clock cycle
  - Multi-cycle CPU (CPI = n)
    - Instructions can take a different number of ***short*** clock cycles to execute
- Recall that a program execution time is: (Instruction count) x (CPI) x (Clock cycle time)
  - In single-cycle implementation cycle time must be set for longest instruction thus requiring shorter instructions to wait
  - Multi-cycle implementation breaks logic into sub-operations each taking one short clock cycle; then each instruction takes only the number of clocks (i.e. CPI) it needs

Single-Cycle Datapath

- To start, let us think about what operations need to be performed for the basic instructions
- All instructions go through the following steps:
  - Fetch: Use PC address to fetch instruction
  - Decode & Register/Operand Fetch: Determine instruction type and fetch any register operands needed
- Once decoded, different instructions require different operations
  - ALU instructions: Perform Add, Sub, etc. and write result back to register
  - LW / SW: Calculate address and perform memory access
  - BEQ / J: Update PC (possible based on comparison)
- Let us start with fetching an instruction and work our way through the necessary components

Instruction Ordering

- Identify which components each instruction type would use and in what order: ALU-Type, LW, SW, BEQ



ALU-Type
(ADD \$5,\$6,\$7)

1. PC
2. I-Memory
3. Registers
4. ALU
5. WB to Reg.

LW
(LW \$5,40(\$7))

1. PC
2. I-Memory
3. Base. Reg.
4. ALU
5. Read Mem.
6. WB to Reg.

SW
(SW \$5,40(\$7))

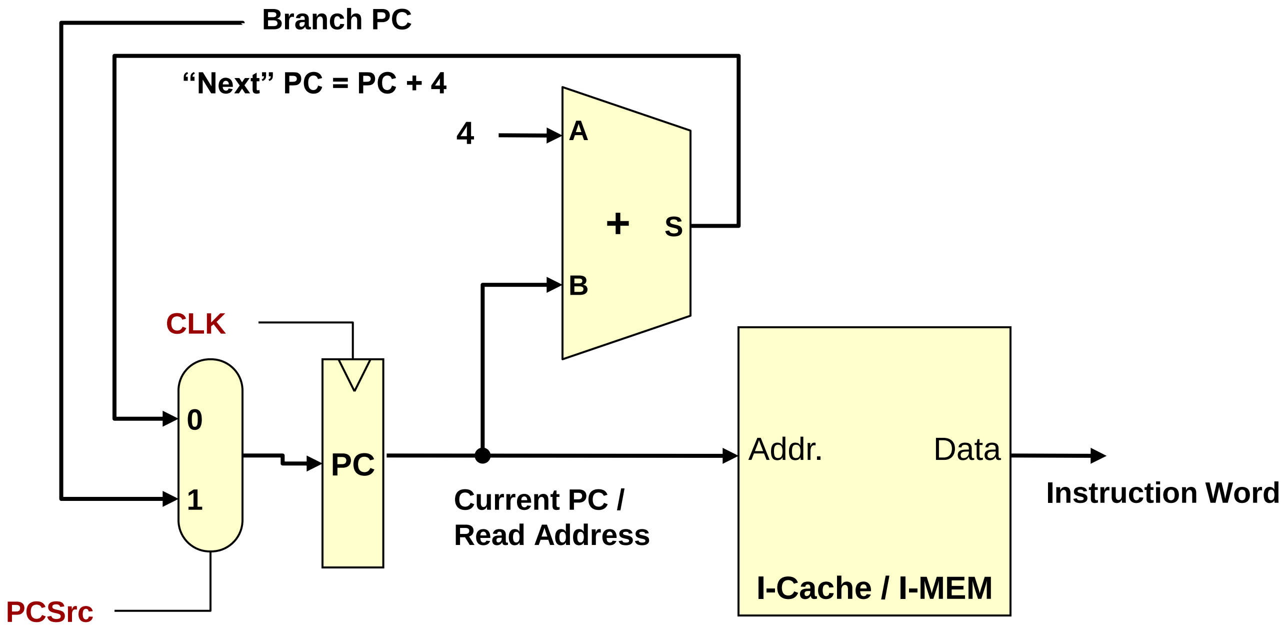
1. PC
2. I-Memory
3. Base. Reg.
4. ALU
5. Write Mem.

BEQ
(BEQ \$2,\$3,disp)

1. PC
2. I-Memory
3. Register Access
4. Compare
5. If Zero,
   Update PC=PC+d

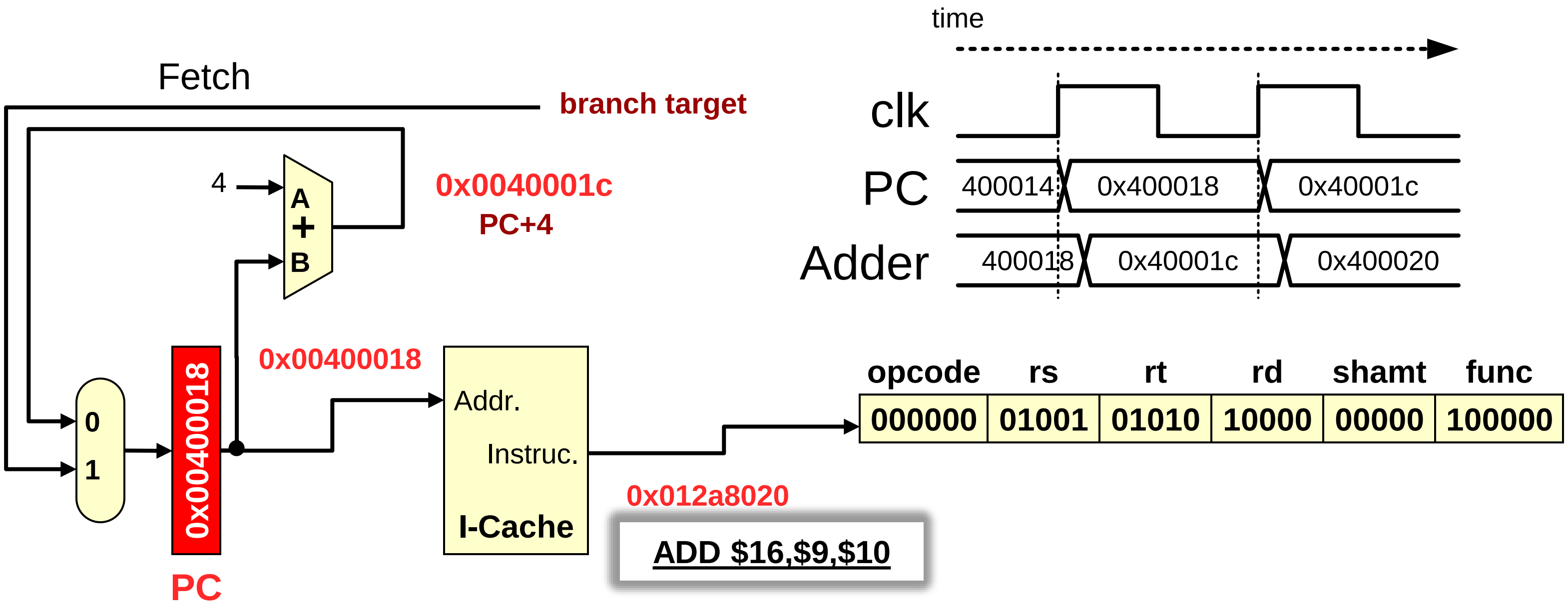
Modified Fetch Datapath

- Below is the fetch datapath modified to support branch instructions



Fetch

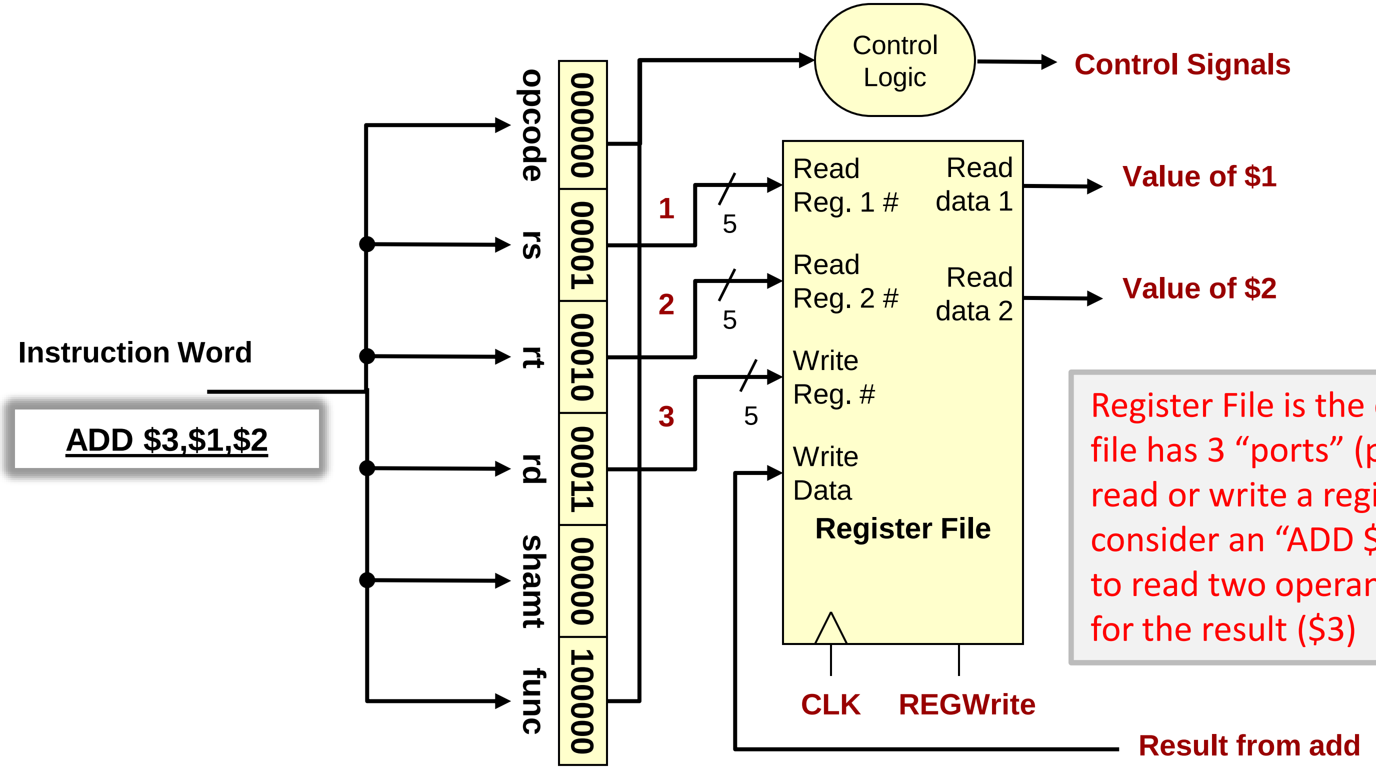
- Address in PC is used to fetch instruction while it is also incremented by 4 to point to the next instruction
- Remember, the PC doesn't update until the end of the clock cycle / beginning of next cycle
- Mux provides a path for branch target addresses



opcode	rs	rt	rd	shamt	func
000000	01001	01010	10000	00000	100000

Decode

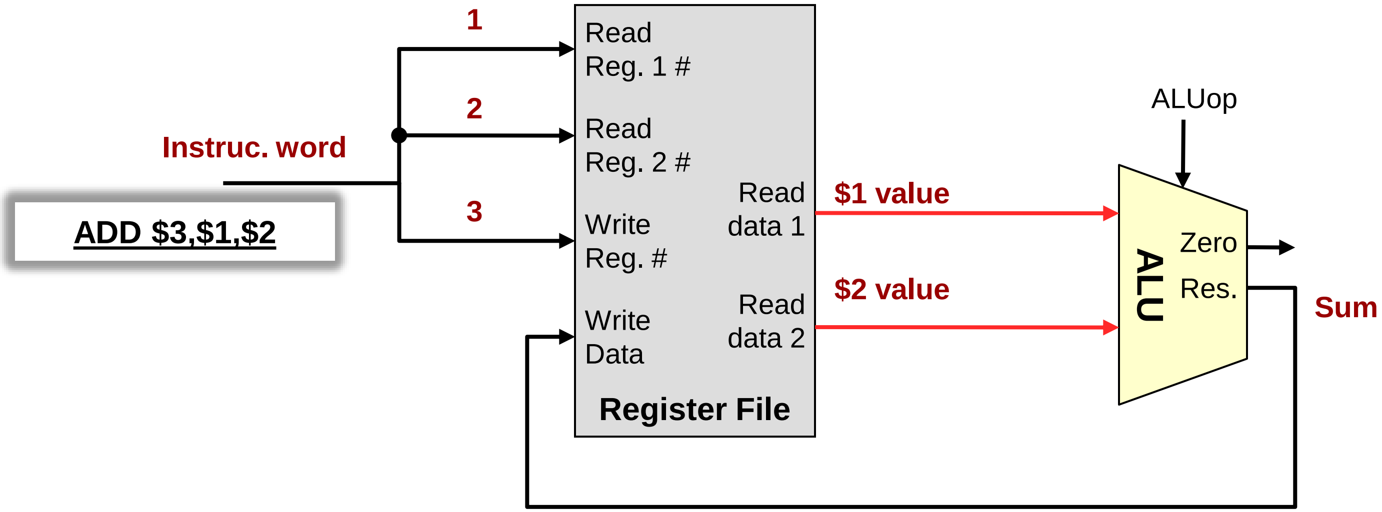
- Opcode and func. field are decoded to produce other control signals
- Execution of an ALU instruction (ADD $3,$1,$2) requires reading 2 register values and writing the result to a third
- REGWrite is an enable signal indicating the write data should be written to the specified register



Register File is the collection of GPR's. Our register file has 3 "ports" (port = ability to concurrently read or write a register). To see why we need 3, consider an "ADD $3,$1,$2". We need 2 read ports to read two operands (i.e. $1 + $2) and 1 write port for the result ($3)

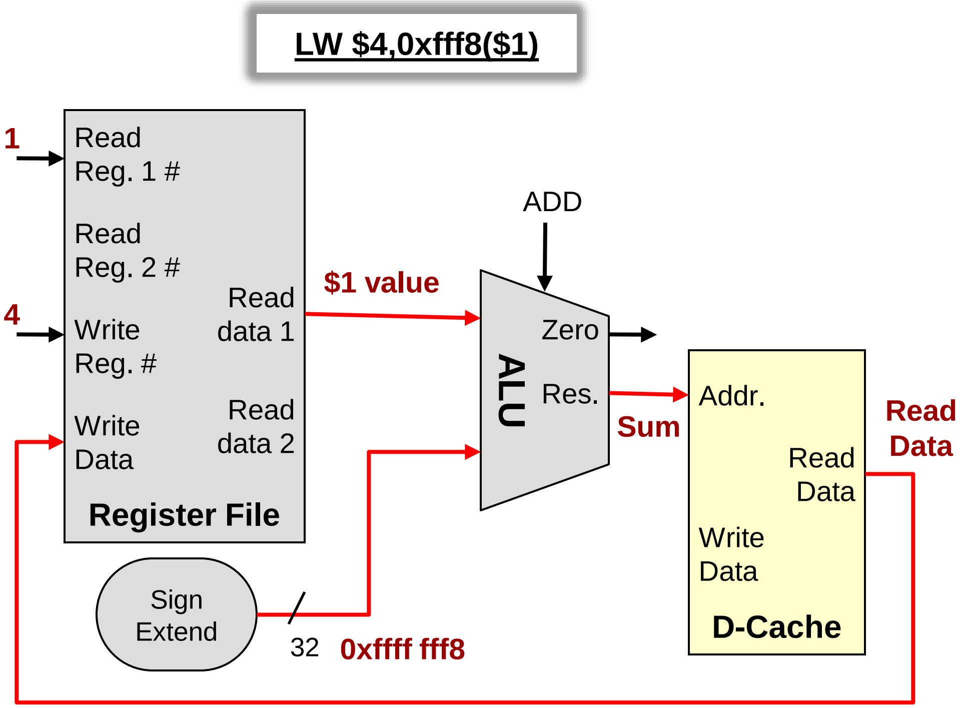
Datapath for ALU instruction

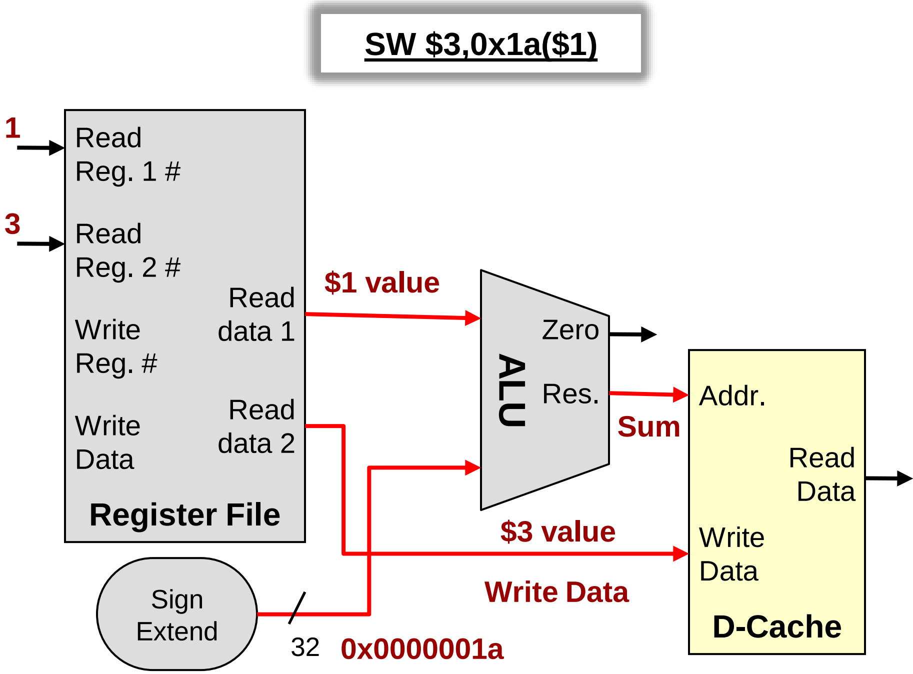
- ALU takes inputs from register file and performs the add, sub, and, or, slt, operations
- Result is written back to dest. register



Memory Access Datapath

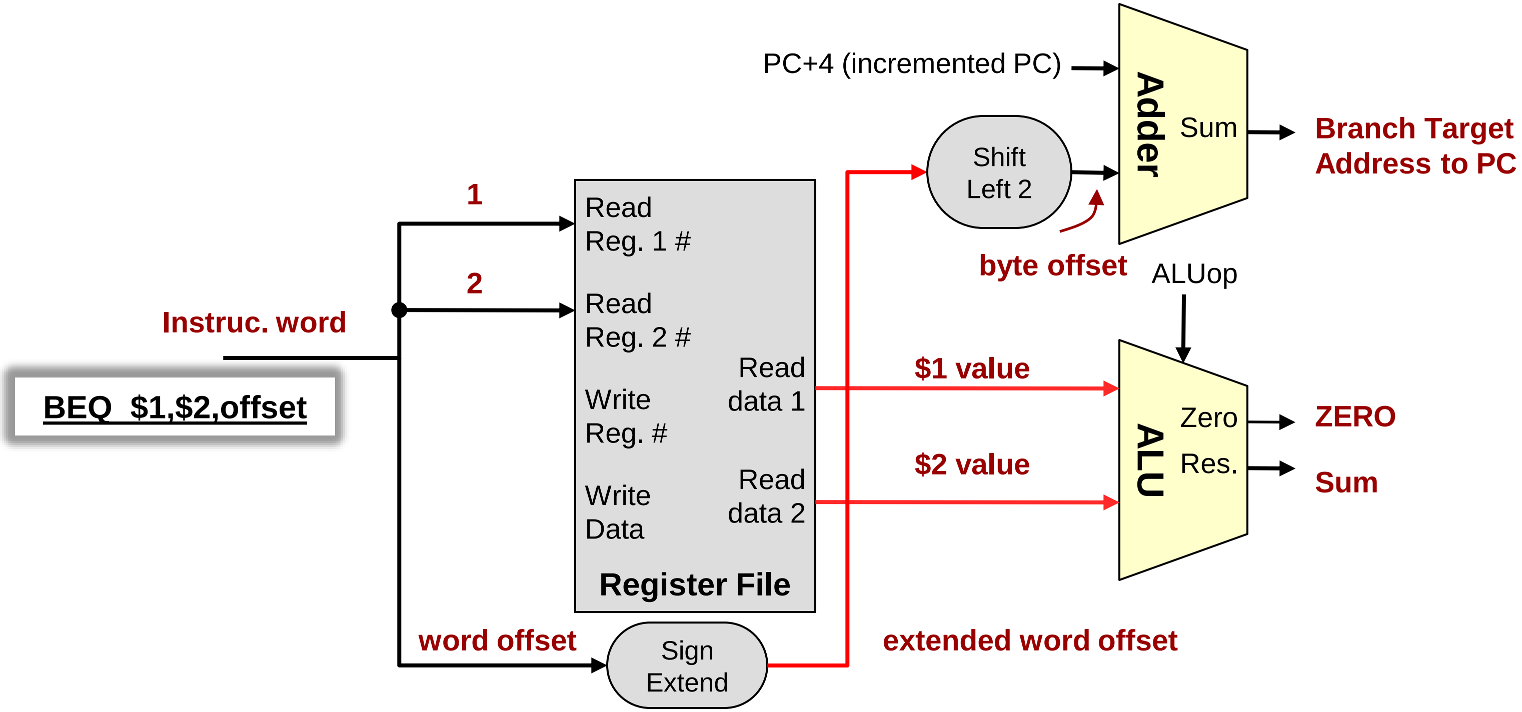
- Operands are read from register file while offset is sign extended
- ALU calculates effective address
- Memory access is performed
- If LW, read data is written back to register





Branch Datapath

- BEQ requires…
  - ALU for comparison (examine ‘zero’ output)
  - Sign extension unit for branch offset
  - Adder to add PC and offset
    - Need a separate adder since ALU is used to perform comparison

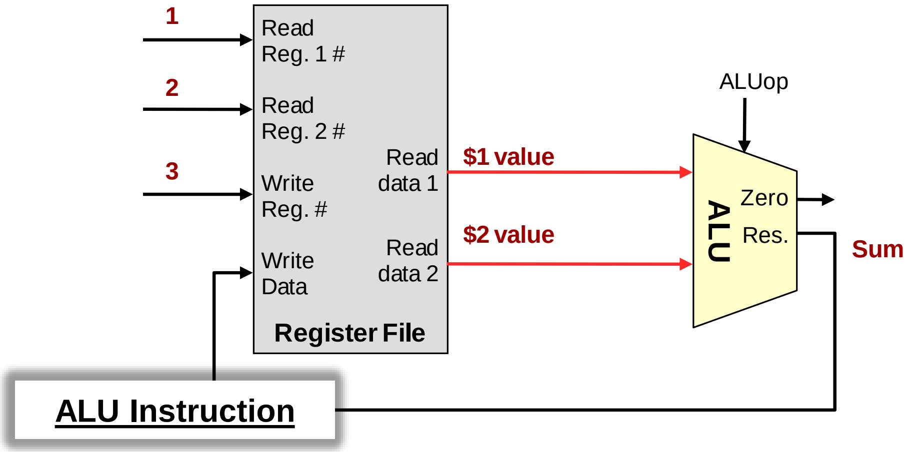


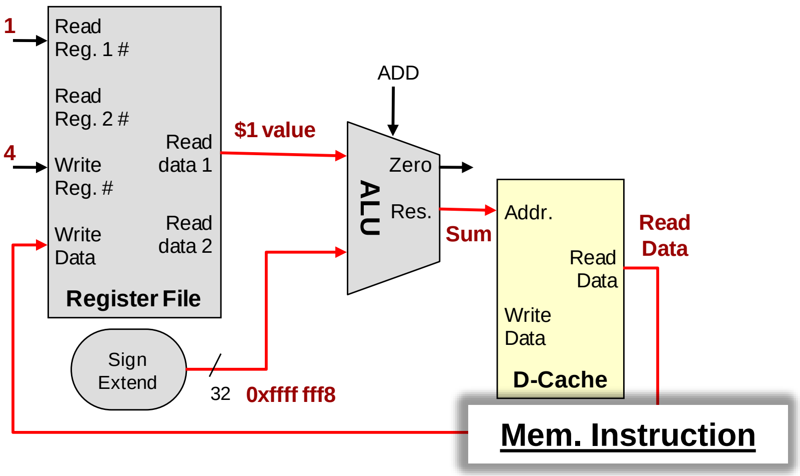
Combining Datapaths

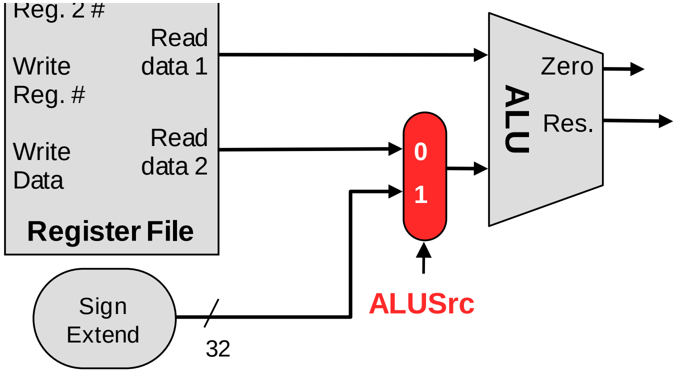
- Now we will take the datapaths for each instruction type and try to combine them into one
- Anywhere we have multiple options for a certain input we can use a mux to select the appropriate value for the given instruction
- Select bits must be generated to control the mux

ALUSrc Mux

- Mux controlling second input to ALU
  - ALU instruction provides Read Register 2 data to the 2nd input of ALU
  - LW/SW uses 2nd input of ALU as an offset to form effective address

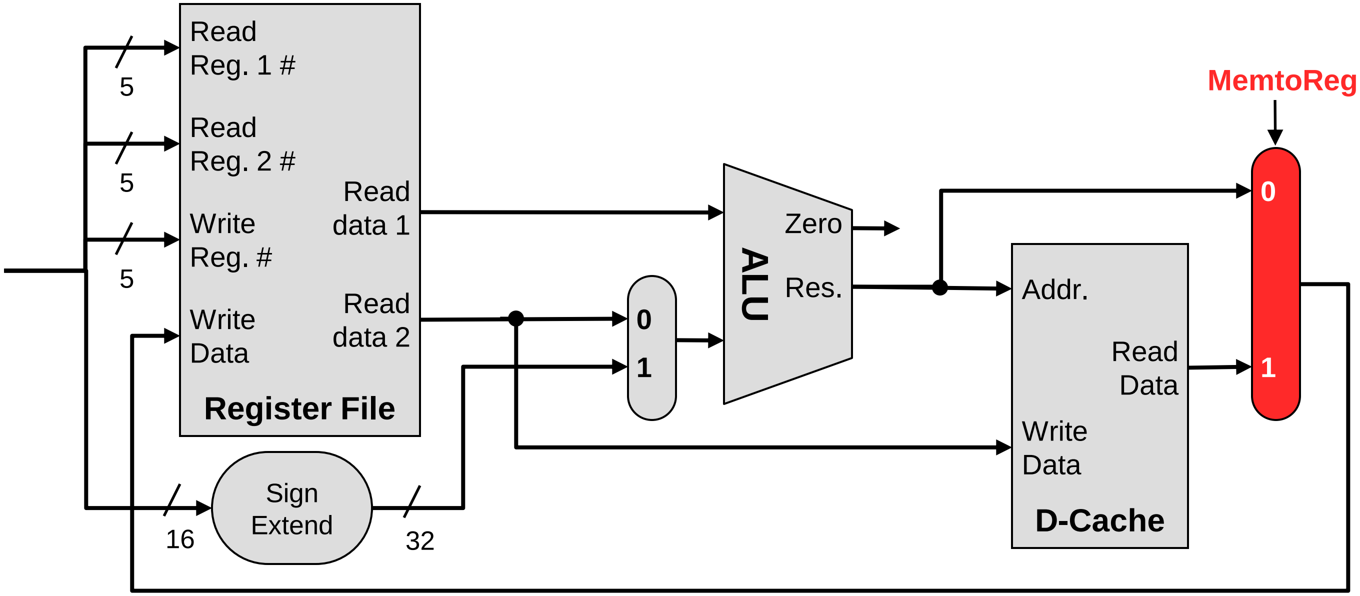






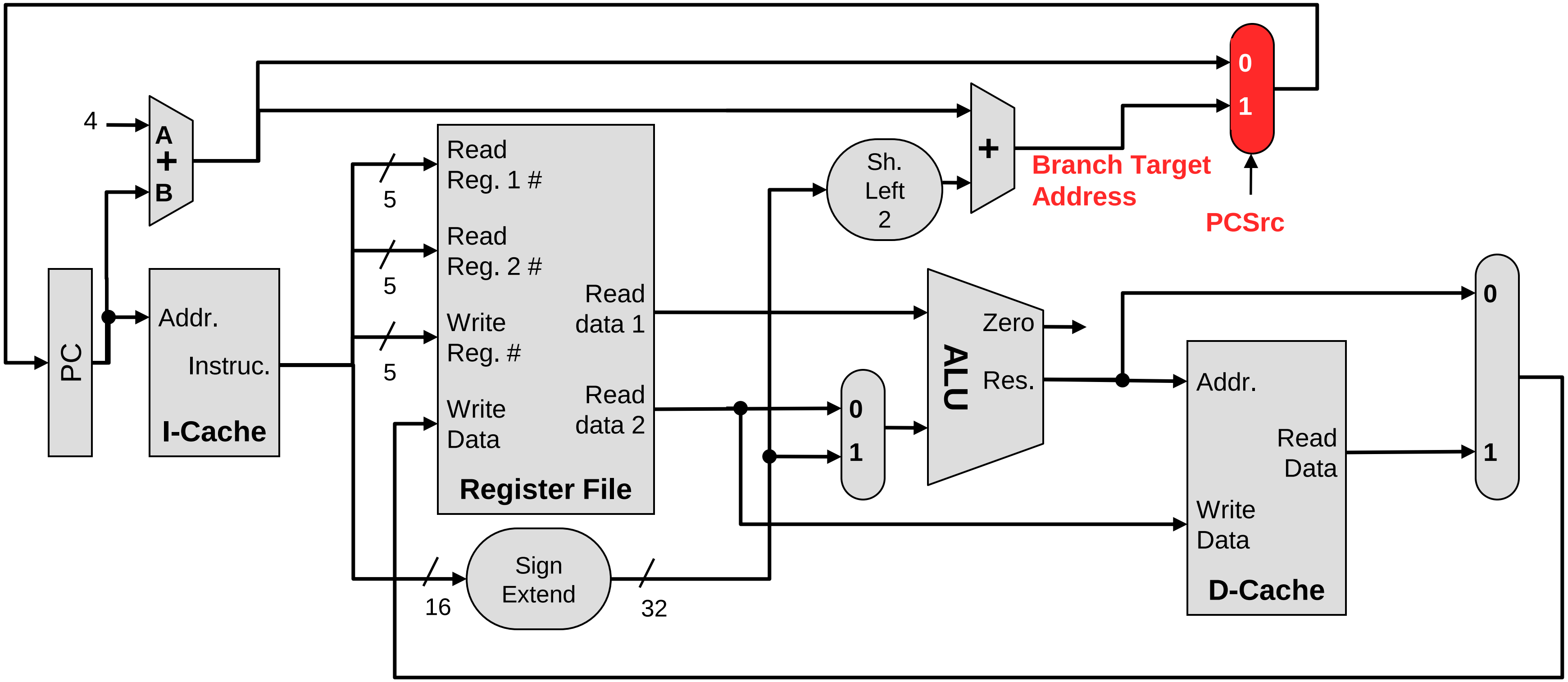
MemtoReg Mux

- Mux controlling writeback value to register file
  - ALU instructions use the result of the ALU
  - LW uses the read data from data memory



PCSrc Mux

- Next instruction can either be at the next sequential address (PC+4) or the branch target address (PC+offset)



RegDst Mux

- Different destination register ID fields for ALU and LW instructions

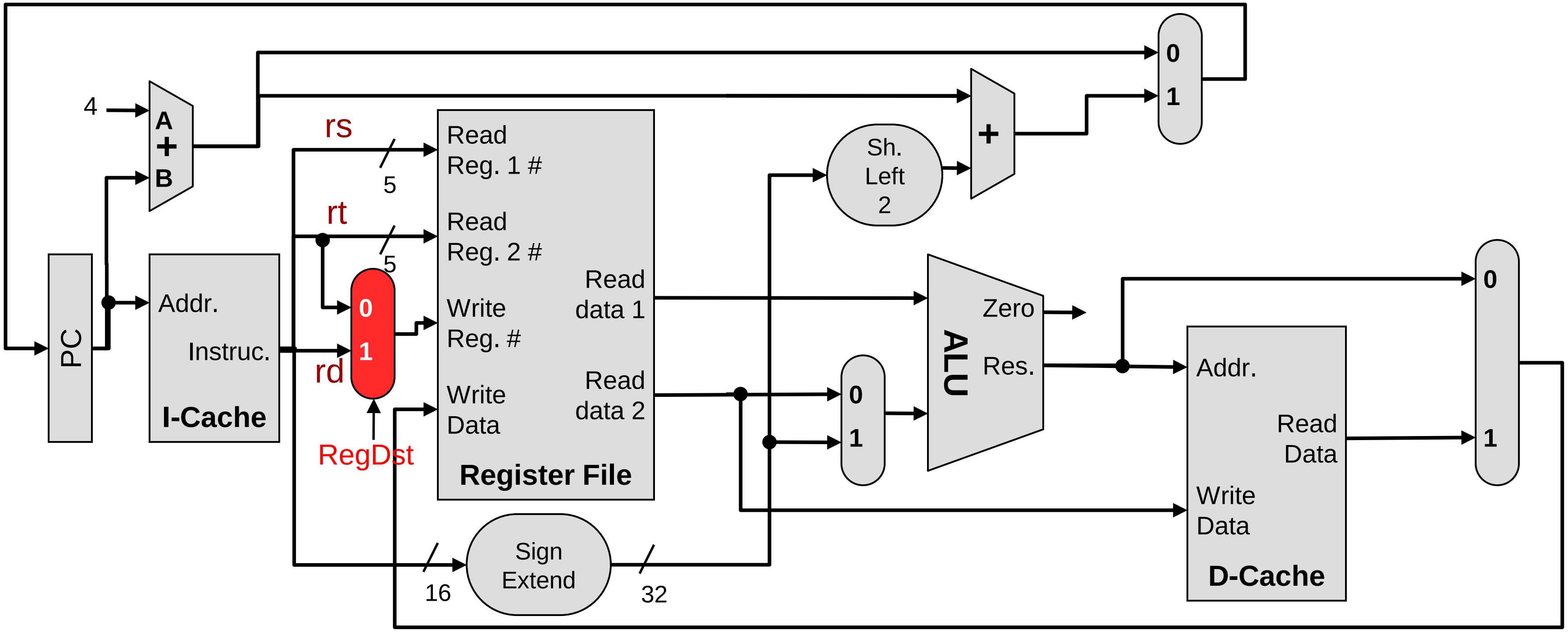
R-Type (ALU)

0	rs	rt	rd	shamt	func
31-26	25-21	20-16	15-11	10-6	5-0

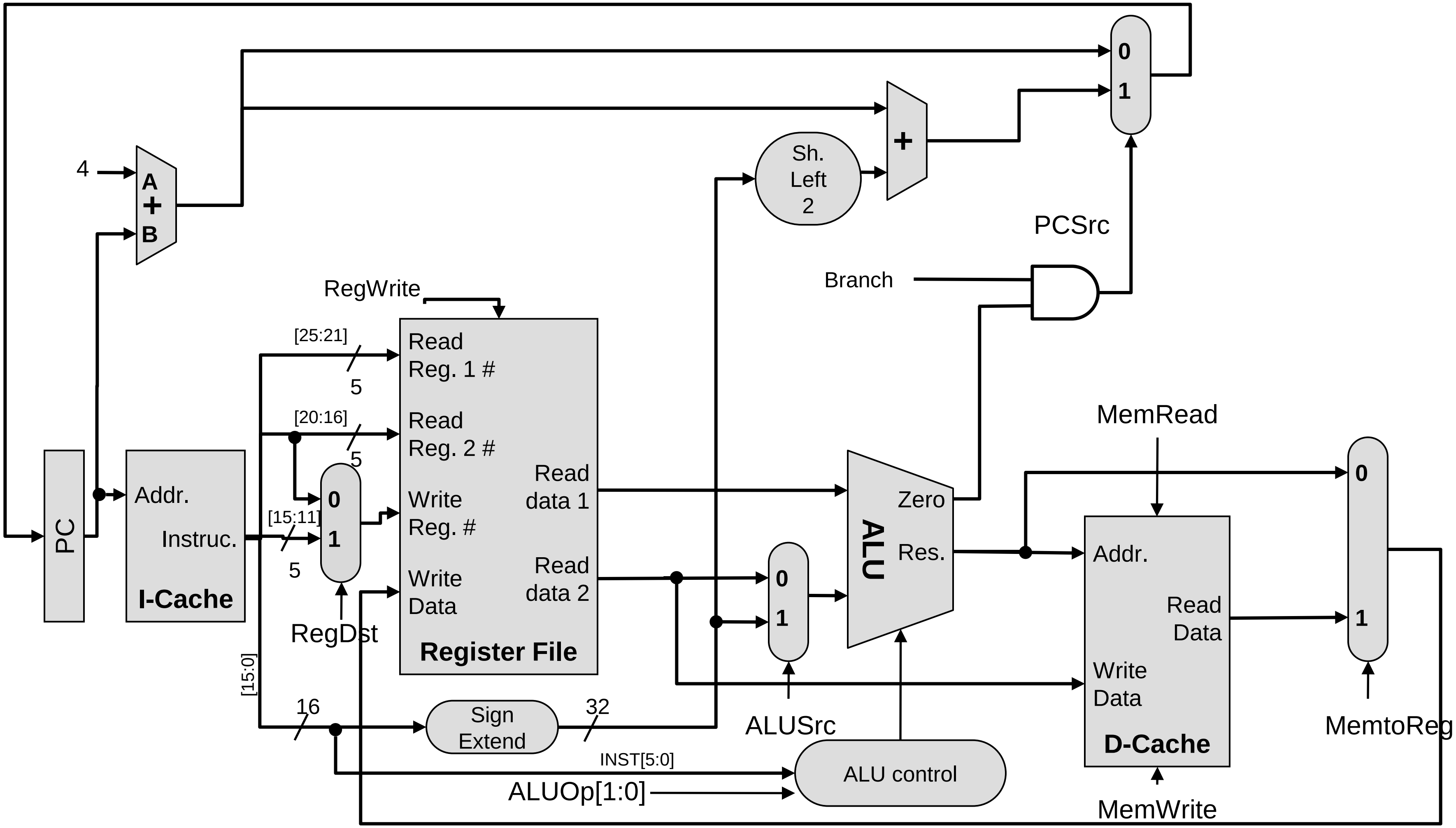
I-Type (LW)

35 or 43	rs	rt	address offset
31-26	25-21	20-16	15-0

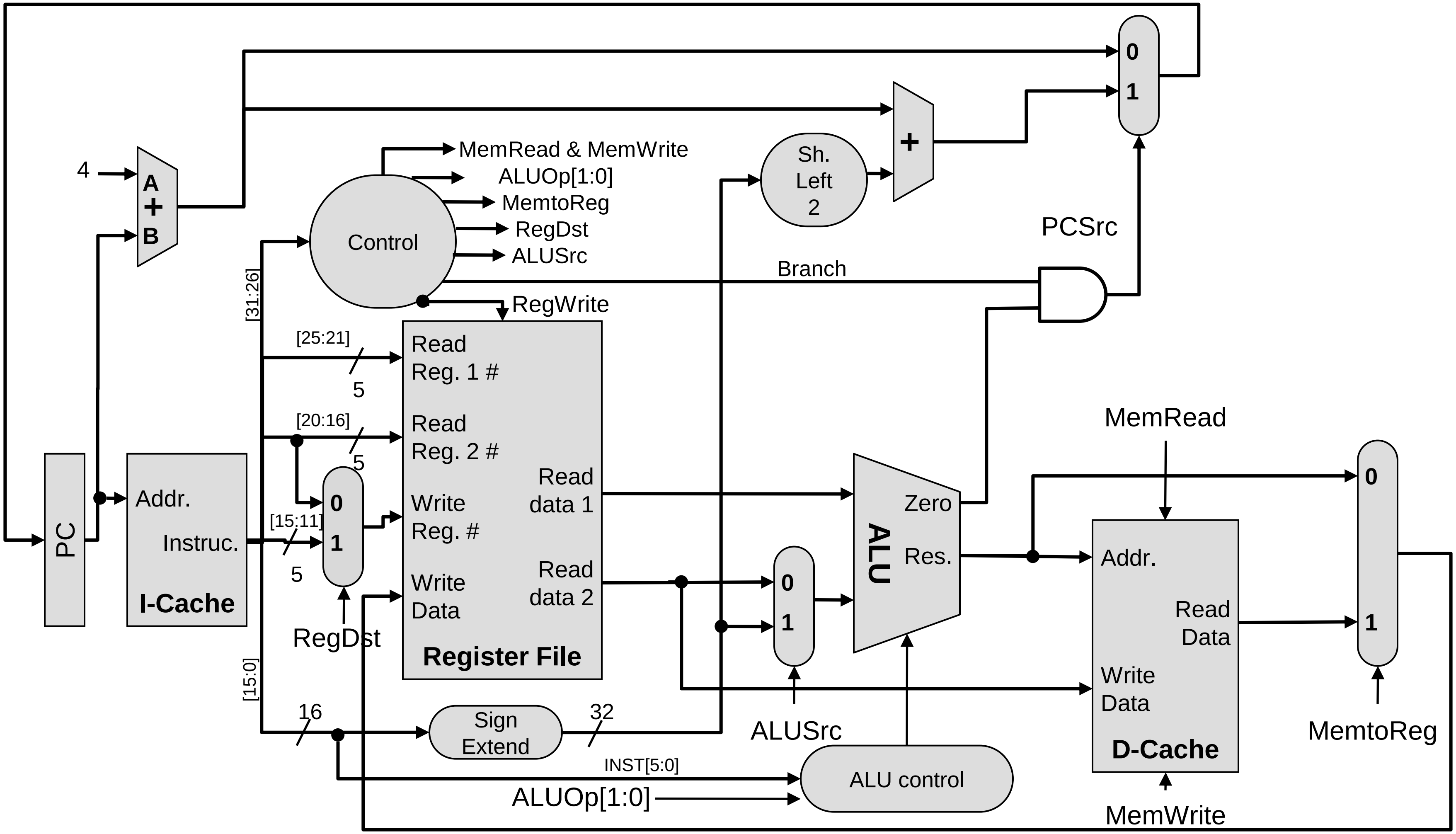
Destination Register Number



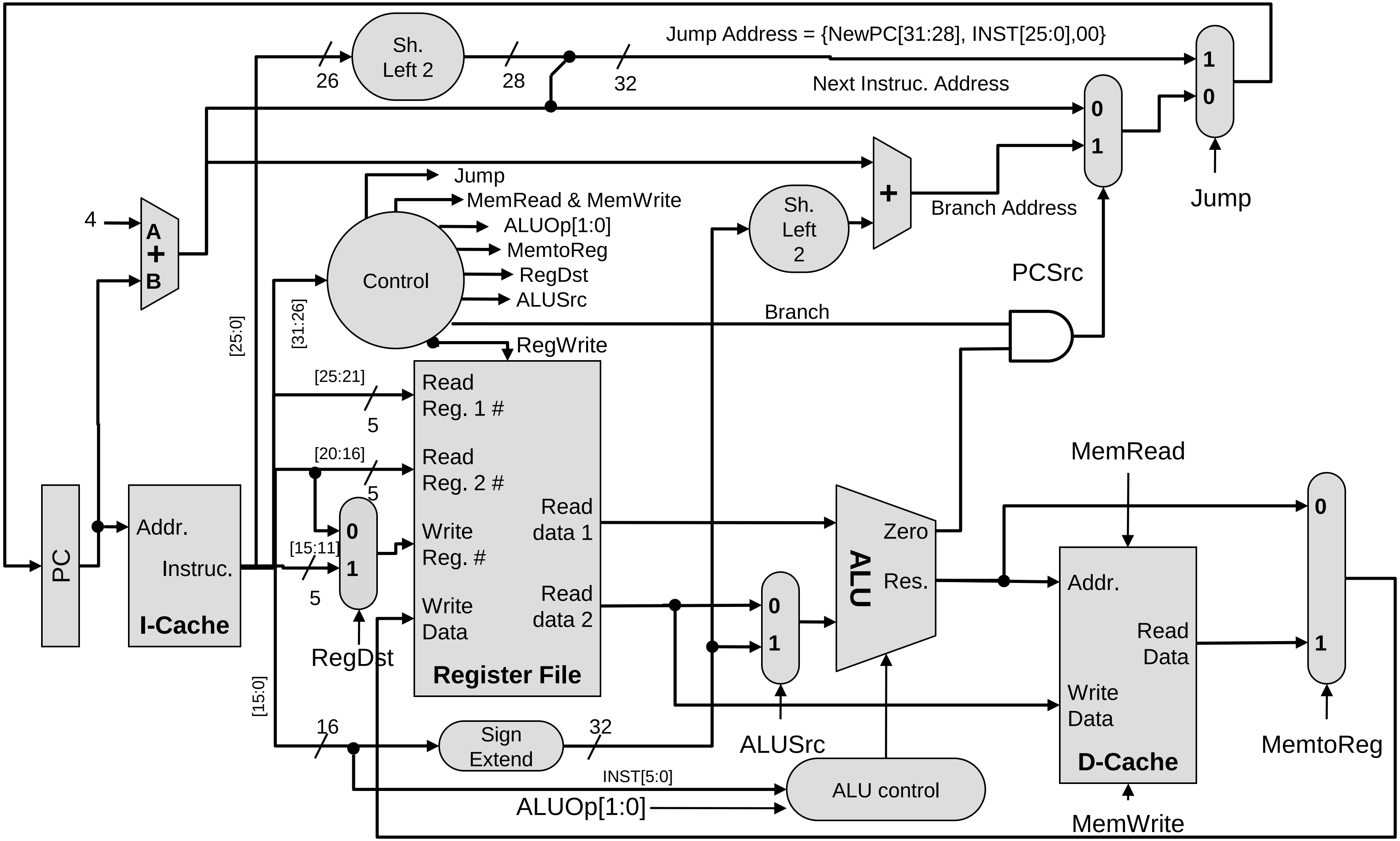
Single-Cycle CPU Datapath



Single-Cycle CPU Datapath



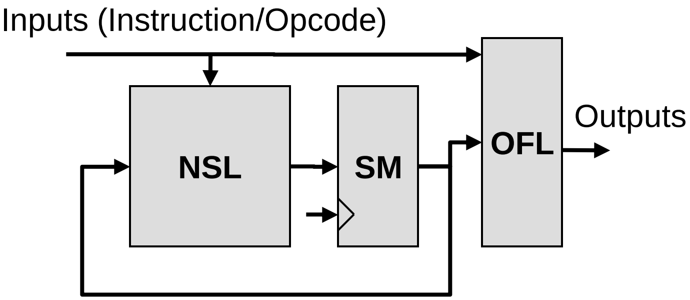
Jump Instruc. Implementation



SINGLE CYCLE CONTROL

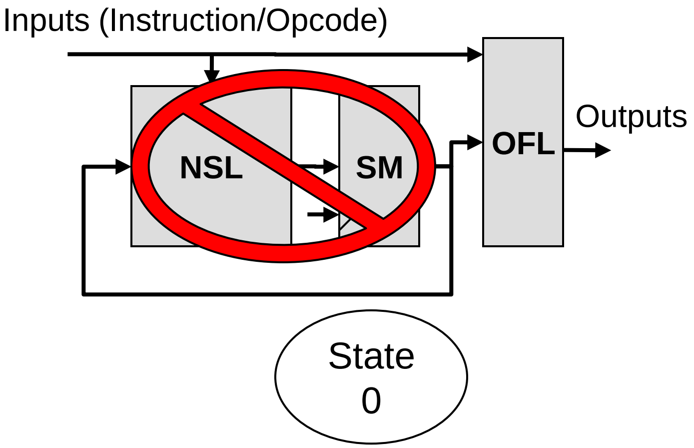
Control Unit Design for Single-Cycle CPU

- Control Unit: Maps instruction to control signals
- Traditional Control Unit
  - FSM: Produces control signals asserted at different times
  - Design NSL, SM, OFL
- Single-Cycle Control Unit
  - Every cycle we perform the same steps: Fetch, Decode, Execute
  - Signals are not necessarily time based but instruction based => only combinational logic



Traditional Control Unit

# of FF's in tightly-encoded state assignment:
5-8 states: _____, 9-16 states: _____

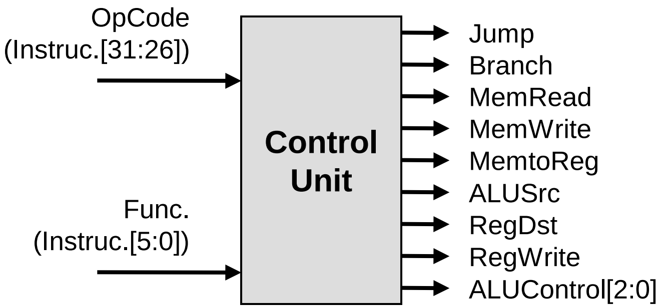


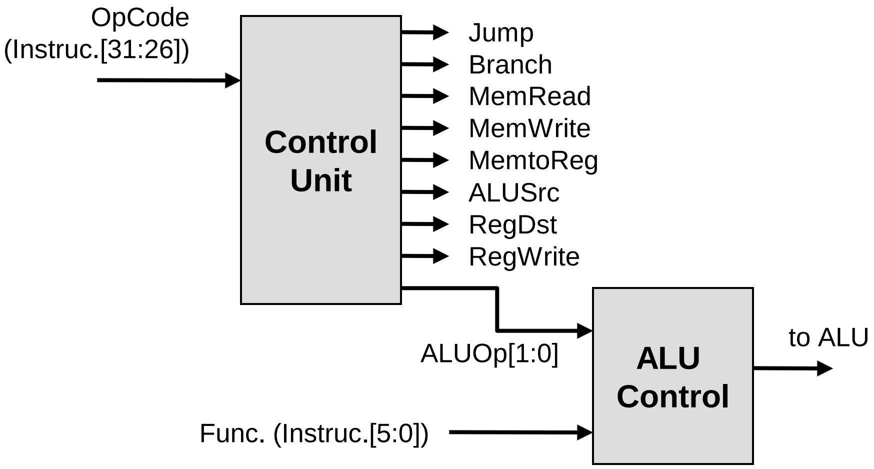
Single-Cycle Control Unit

Only 1 state => _____ FF's

Control Unit

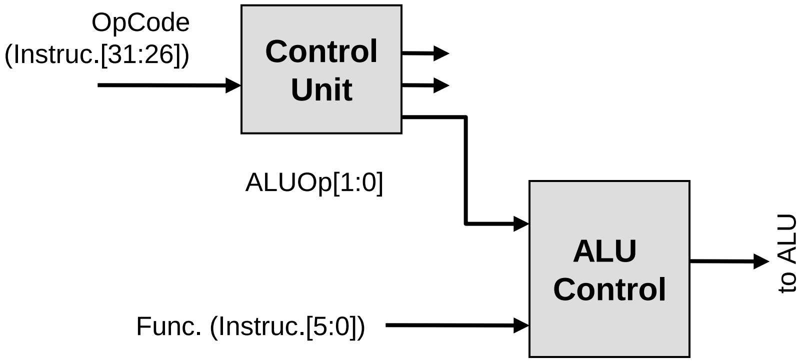
- Most control signals are a function of the opcode (i.e. LW/SW, R-Type, Branch, Jump)
- ALU Control is a function of opcode AND function bits.





ALU Control

- ALU Control needs to know what instruction type it is:
  - R-Type (op. depends on func. code)
  - LW/SW (op. = ADD)
  - BEQ (op. = SUB)
- Let main control unit produce ALUOp[1:0] to indicate instruction type, then use function bits if necessary to tell the ALU what to do



Instruction	ALUOp[1:0]
LW/SW	00
Branch	01
R-Type	10

Control unit maps instruction opcode to ALUOp[1:0] encoding

ALU Control Truth Table

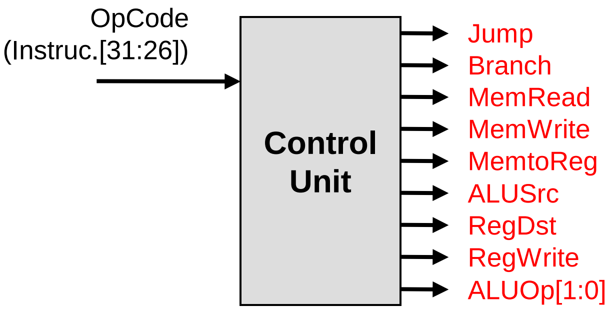
- ALUControl[2:0] is a function of: ALUOp[1:0] and Func.[5:0]

Instruc.	ALUOp[1:0]	Instruction Operation	Func.[5:0]	Desired ALU Action
LW	00	Load word	X	Add
SW	00	Store word	X	Add
Branch	01	BEQ	X	Subtract
R-Type	10	AND	100100	And
R-Type	10	OR	100101	Or
R-Type	10	Add	100000	Add
R-Type	10	Sub	100010	Subtract
R-Type	10	SLT	101010	Set on less than

Produce each ALUControl[2:0] bit from the ALUOp and Func. inputs

Control Signal Generation

- Other control signals are a function of the opcode
- We could write a full truth table or (because we are only implementing a small subset of instructions) simply decode the opcodes of the specific instructions we are implementing and use those intermediate signals to generate the actual control signals

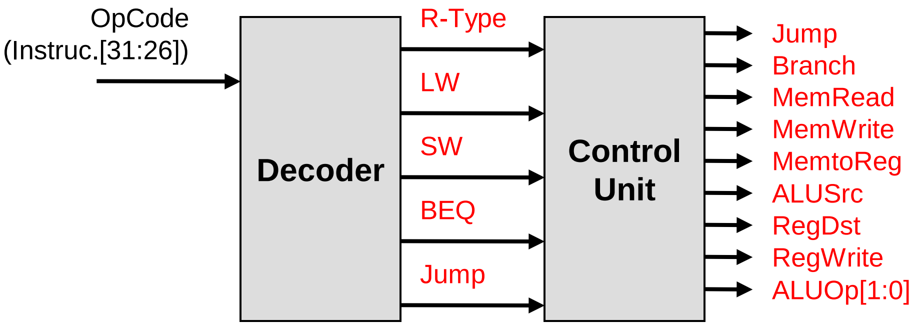


OpCode (Instruc.[31:26])

Control Unit

Jump
Branch
MemRead
MemWrite
MemtoReg
ALUSrc
RegDst
RegWrite
ALUOp[1:0]

Could generate each control signal by writing a full truth table of the 6-bit opcode



OpCode (Instruc.[31:26])

Decoder

R-Type
LW
SW
BEQ
Jump

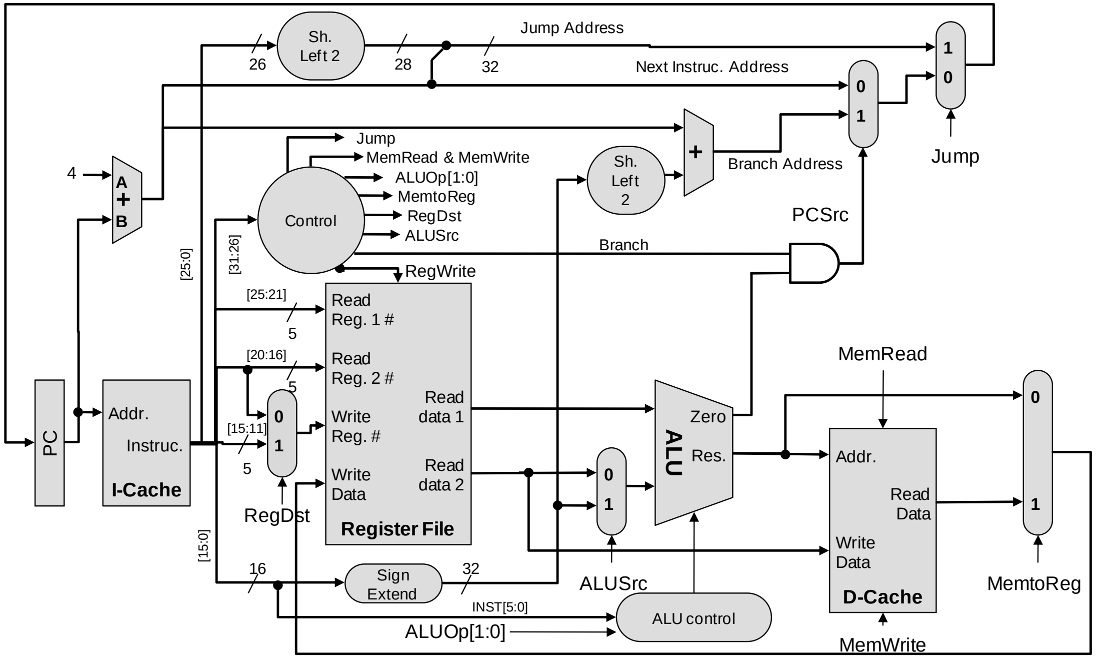
Control Unit

Jump
Branch
MemRead
MemWrite
MemtoReg
ALUSrc
RegDst
RegWrite
ALUOp[1:0]

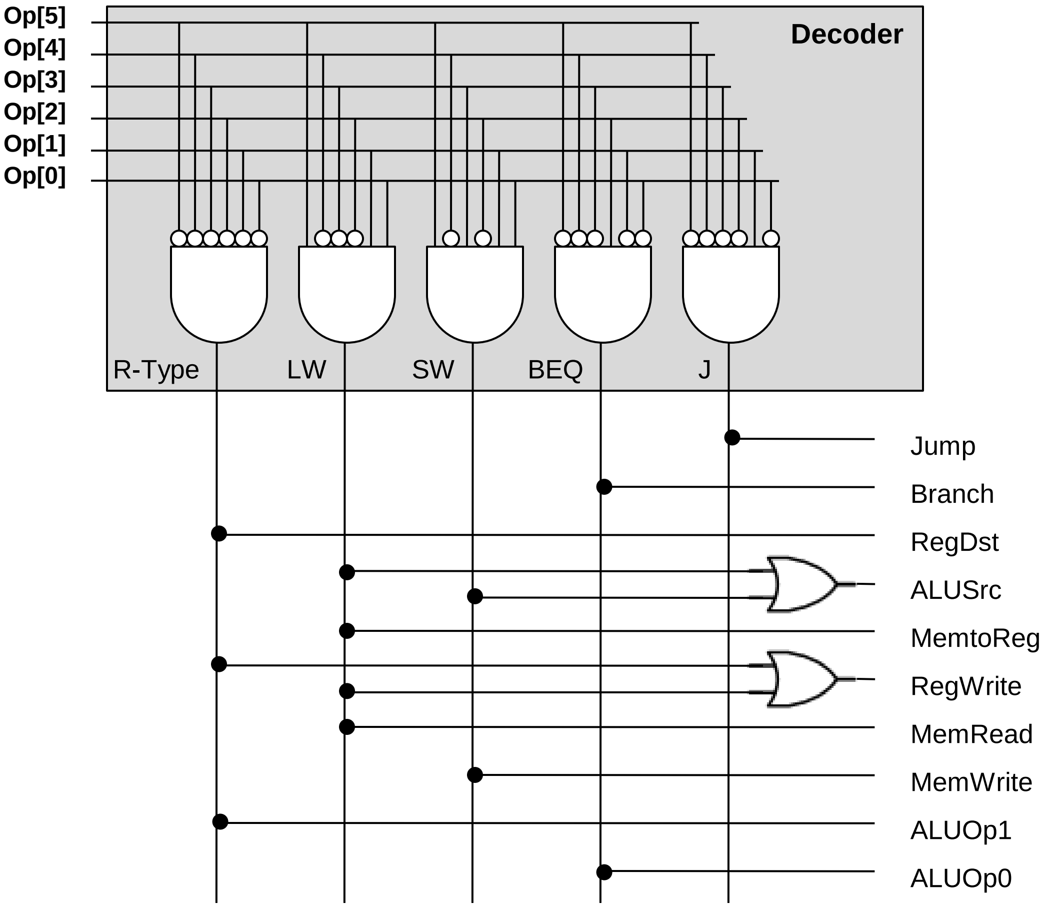
Simpler for human to design if we decode the opcode and then use individual “instruction” signals to generate desired control signals

Control Signal Truth Table

R-Type	LW	SW	BEQ	J	Jump	Branch	Reg Dst	ALU Src	Memto-Reg	Reg Write	Mem Read	Mem Write	ALU Op[1]	ALU Op[0]
1	0	0	0	0	0	0	1	0	0	1	0	0	1	0
0	1	0	0	0	0	0	0	1	1	1	1	0	0	0
0	0	1	0	0	0	0	X	1	X	0	0	1	0	0
0	0	0	1	0	0	1	X	0	X	0	0	0	0	1
0	0	0	0	1	1	X	X	X	X	0	0	0	X	X



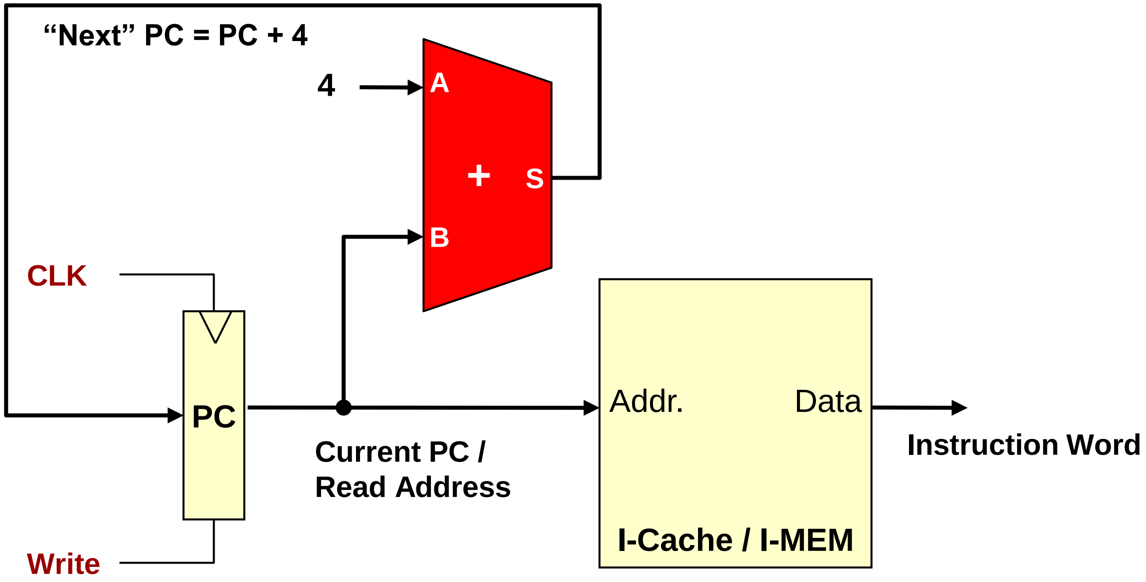
Control Signal Logic



DATAPATH QUESTIONS

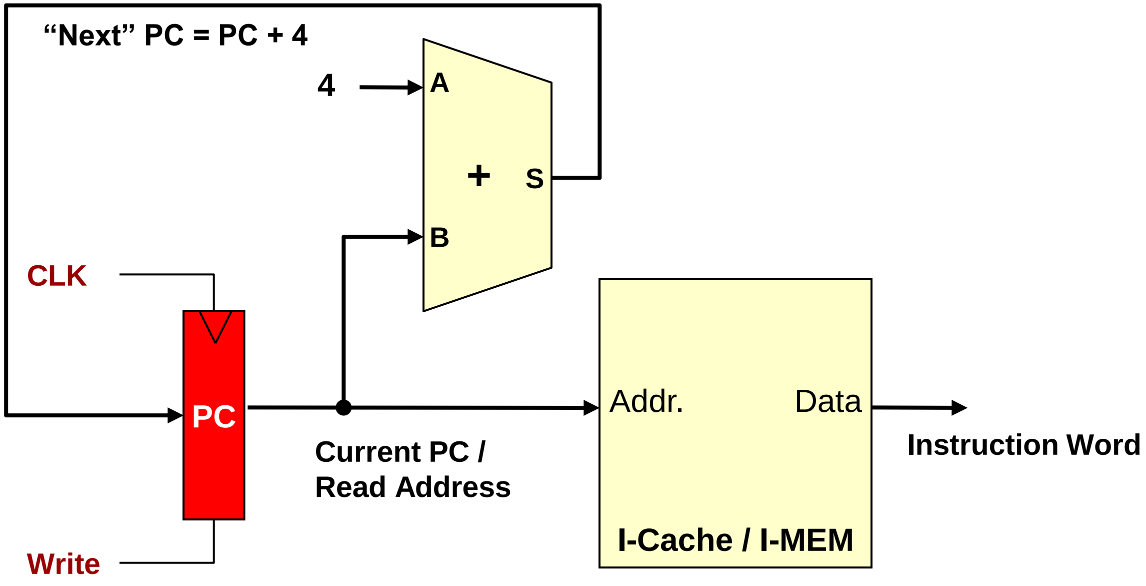
Fetch Datapath Question 1

- Can the adder used to increment the PC be an ALU and be used/shared for ALU instructions like ADD/SUB/etc.
  - In a single-cycle CPU, resources cannot be shared thus we need a separate adder and separate ALU



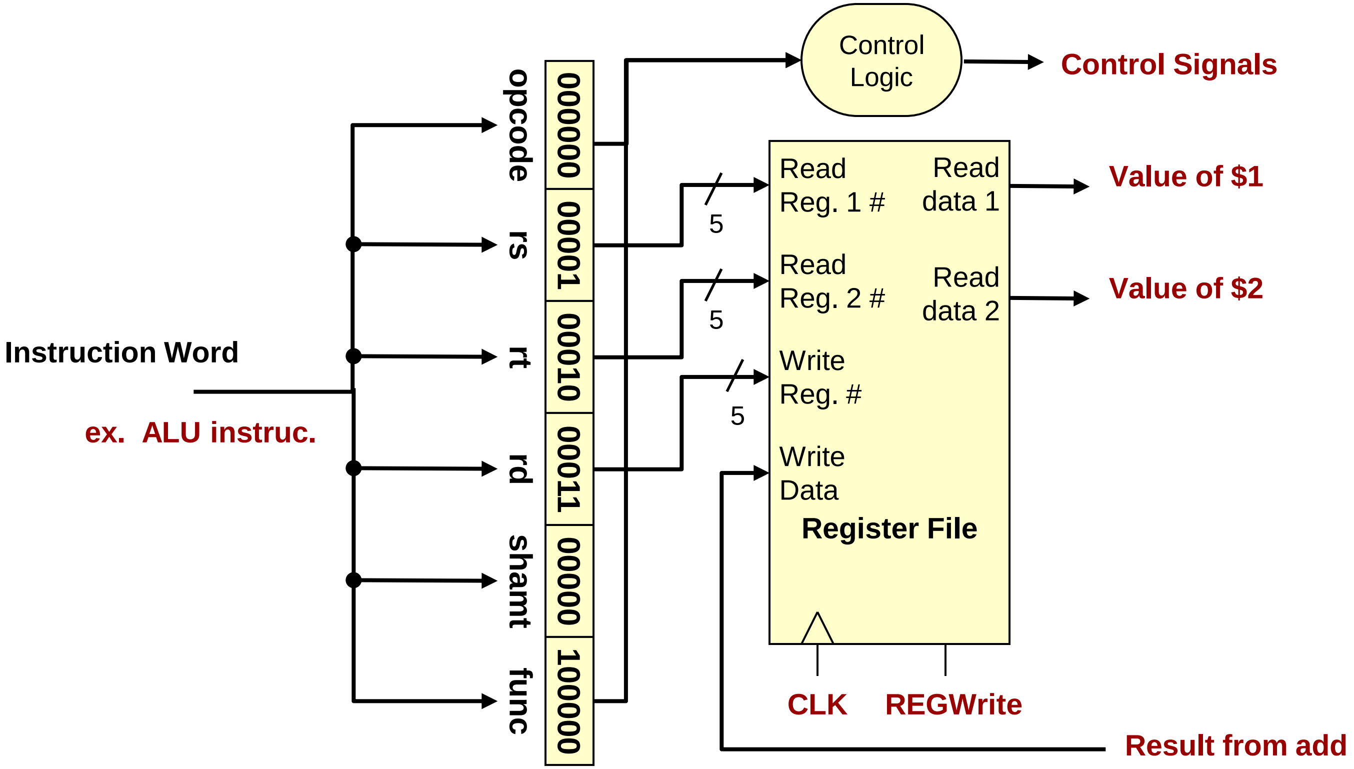
Fetch Datapath Question 2

- Do we need the "Write" enable signal on the PC register for our single-cycle CPU?
  - In the single-cycle CPU, the PC is updated EVERY clock cycle (since we execute a new instruction each cycle). Thus we are writing the PC every cycle and don't need the write signal.



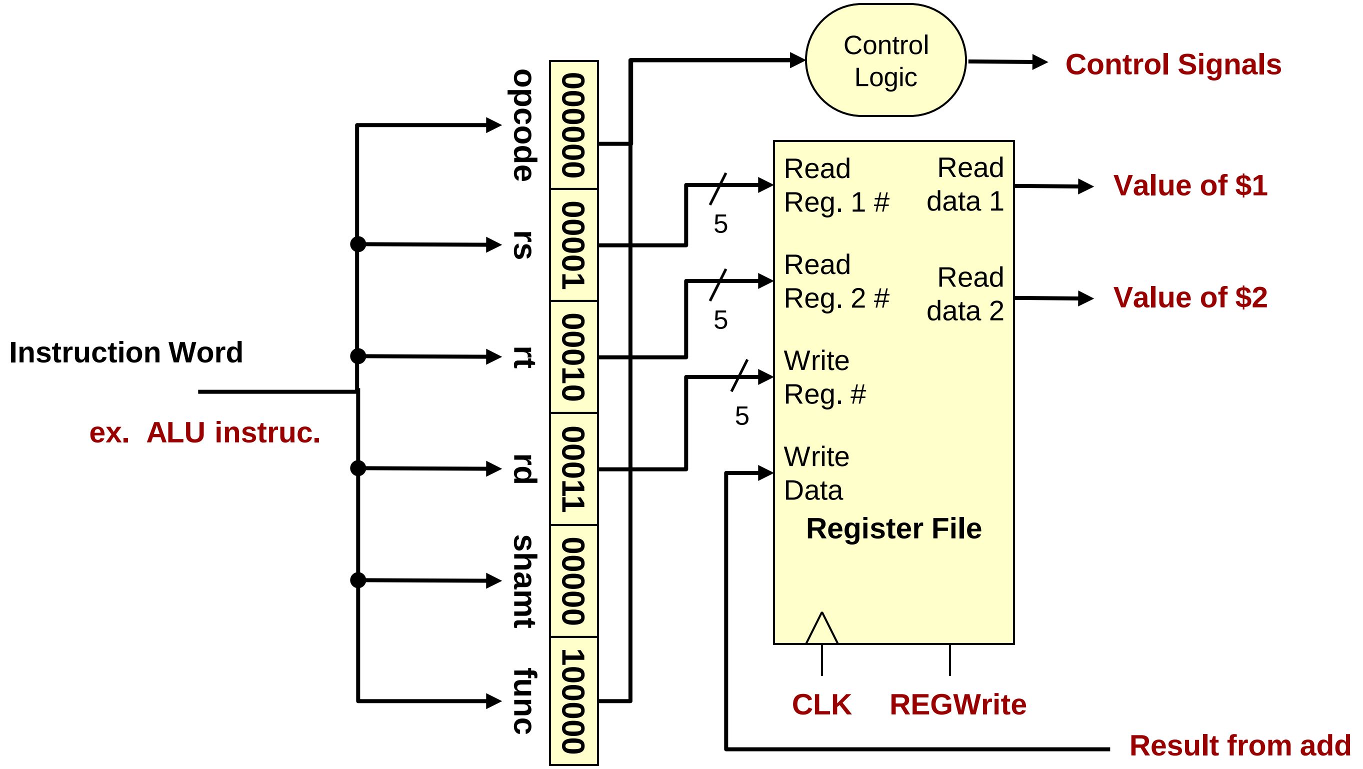
RegFile Question 1

- Why do we need the write enable signal, REGWrite?
  - We have certain instructions like BEQ or SW that do not cause a register to be updated. Thus we need the ability to NOT change a register.



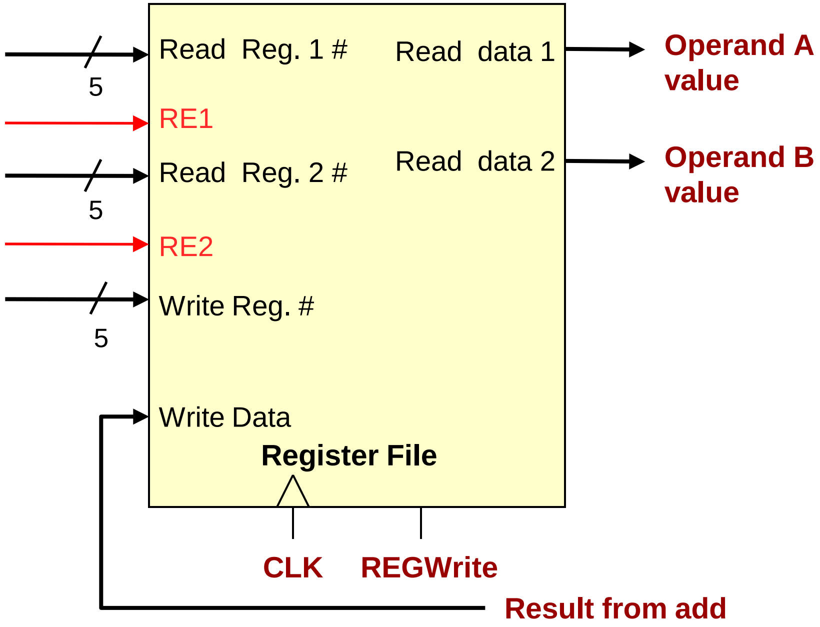
RegFile Question 2

- Can write to registers be level sensitive or does it have to be edge-sensitive?
  - It must be edge-sensitive since a register may be source and destination (i.e. add $1,$1,$2). If it was level sensitive we would have an uncontrolled feedback loop.



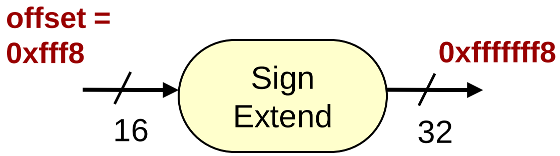
RegFile Question 3

- Since we need a write enable, do we need read enables (i.e. RE1, RE2)
  - We do not need read enables because reading a value does not change the state of the processor. It may be unnecessary even if no source registers are needed (e.g. Jmp), reading data out of the register file should not cause harm.



Sign Extension Unit

- In a ‘LW’ or ‘SW’ instructions with their base register + offset format, the instruction only contains the offset as a 16-bit value
  - Example: LW $4,-8($1)
  - Machine Code: 0x8c24fff8
    - -8 = 0xfff8
- The 16-bit offset must be extended to 32-bits before being added to base register

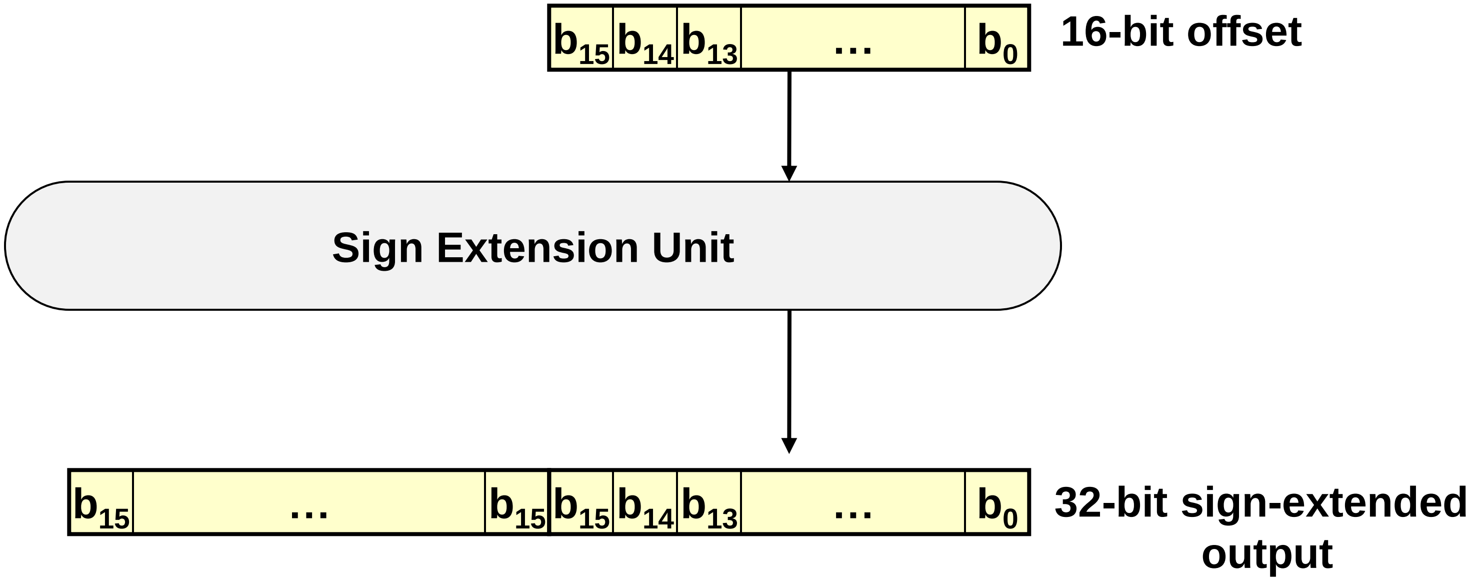


LW $4,0xfff8($1)

100011	**00001**	**00100**	**1111 1111 1111 1000**
opcode	**rs**	**rt**	**offset**

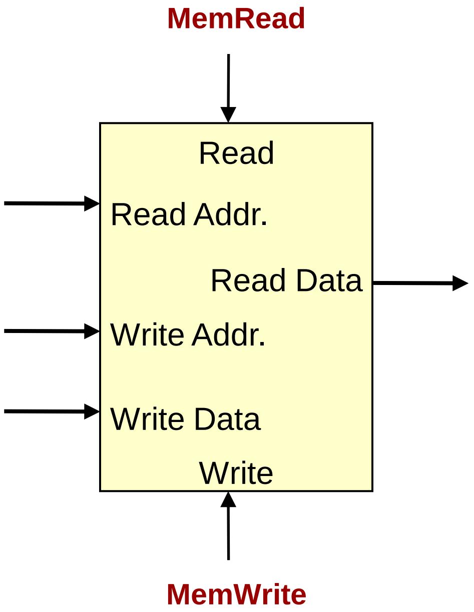
Sign Extension Question

- What logic is inside a sign-extension unit?
  - How do we sign extend a number?
  - Do you need a shift register?



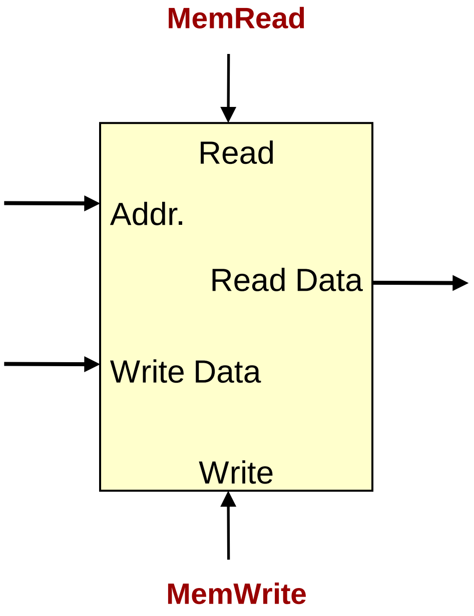
Data Memory Questions

- Do we need separate instruction and data memory or can we just use one (i.e. most personal computers only have one large set of RAM)?
- Do we need separate read/write address inputs or can we have just one address input used for both operations?
- Do we need separate read/write data input/output or a bidirectional input (for write) / output (for read)?
- Can we do away with the "read" control signal (similar to how we did away with read enables for register file)?



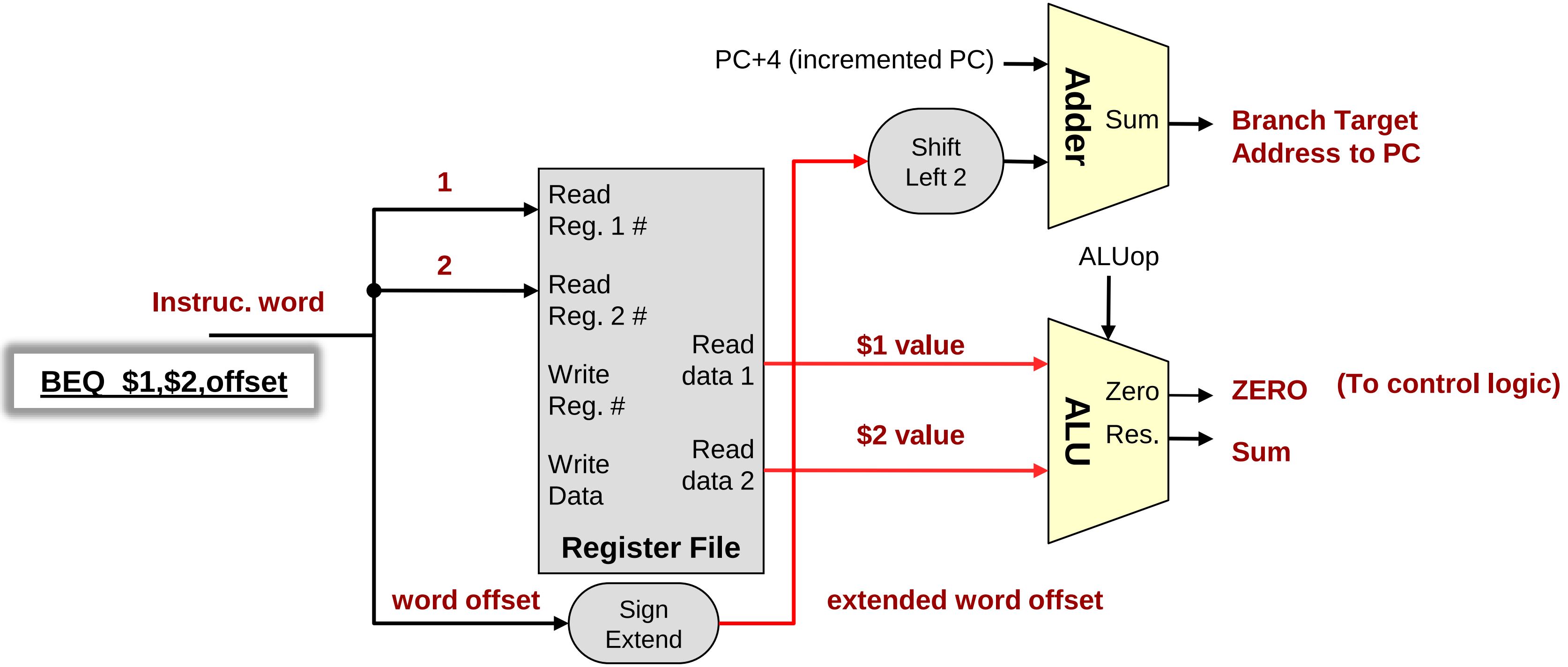
Data Memory Answers

- We do need separate memories for instruction and data memories since we want to fetch an instruction and read/write data in the same clock (i.e. can't share the memory)
- In the case of a single cycle CPU, we only perform one read/write at a time thus we can share address inputs and, if we want, make the data input/output bidirectional, however we can also have separate data input/outputs
- Without a read control signal the memory would always be reading based on the address input (which will be arbitrary values for non-memory instructions). This can have serious side effects such as invalid address and, since this memory is likely a cache, cache misses, etc.



Branch Datapath Question

- Is it okay to start adding branch offset even before determining whether the branch is taken or not?
  - Yes, it does not hurt because the ZERO signal will control whether that Branch Target is used to update the PC or not



Credits

- These slides were derived from Gandhi Puvvada's EE 457 Class Notes