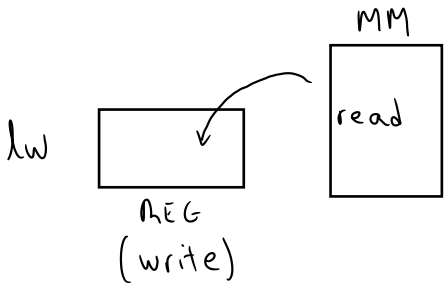
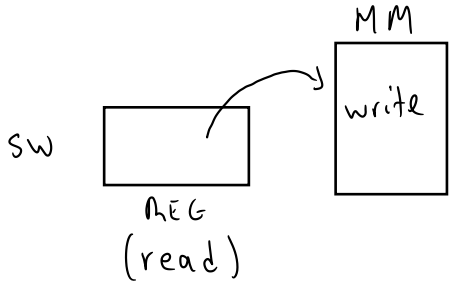
Execution Cycles

- Fetch instruction, update the PC register
  - Instruction memory (read only)
  - Get in with an address
  - The address is stored in the PC register
  - Get out with the instruction
  - PC register is updated [PC + 4]
- Decode
  - The instruction is parsed into fields (R-format has OPCODE 000000)
  - Get the source data
  - Register file (2 read and 1 write port)
  - * Must understand the implementation of the read and write ports (B56 on Patterson)
- Execute
  - ALU: compute logical and arithmetic operations
  - Output: result of the operation, it can represent target address (lw, sw)
    - check for zero
- Write back / memory access
  - Data memory or register file
  - Write-back: R-format
  - Memory access: lw and sw





From Patterson:

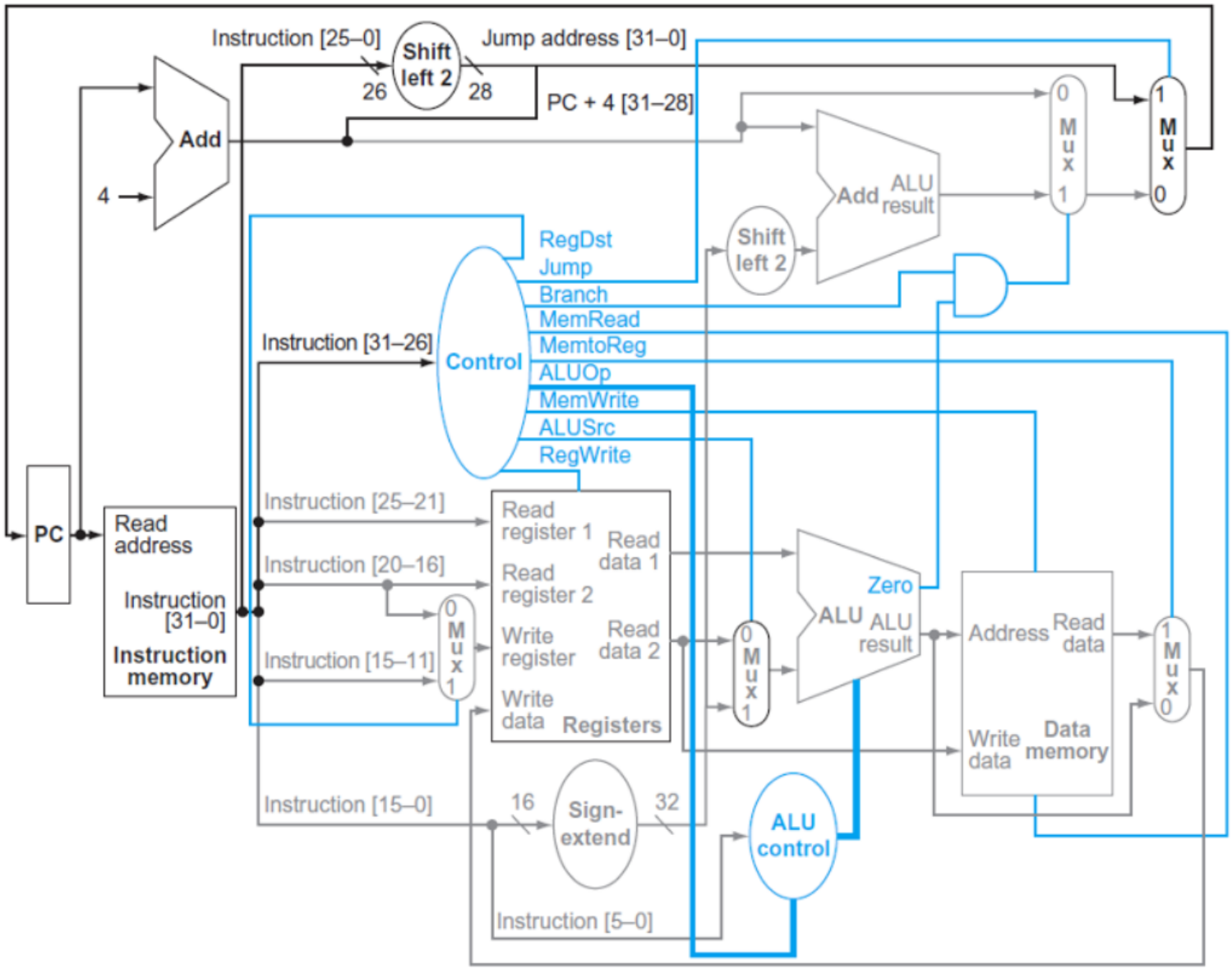
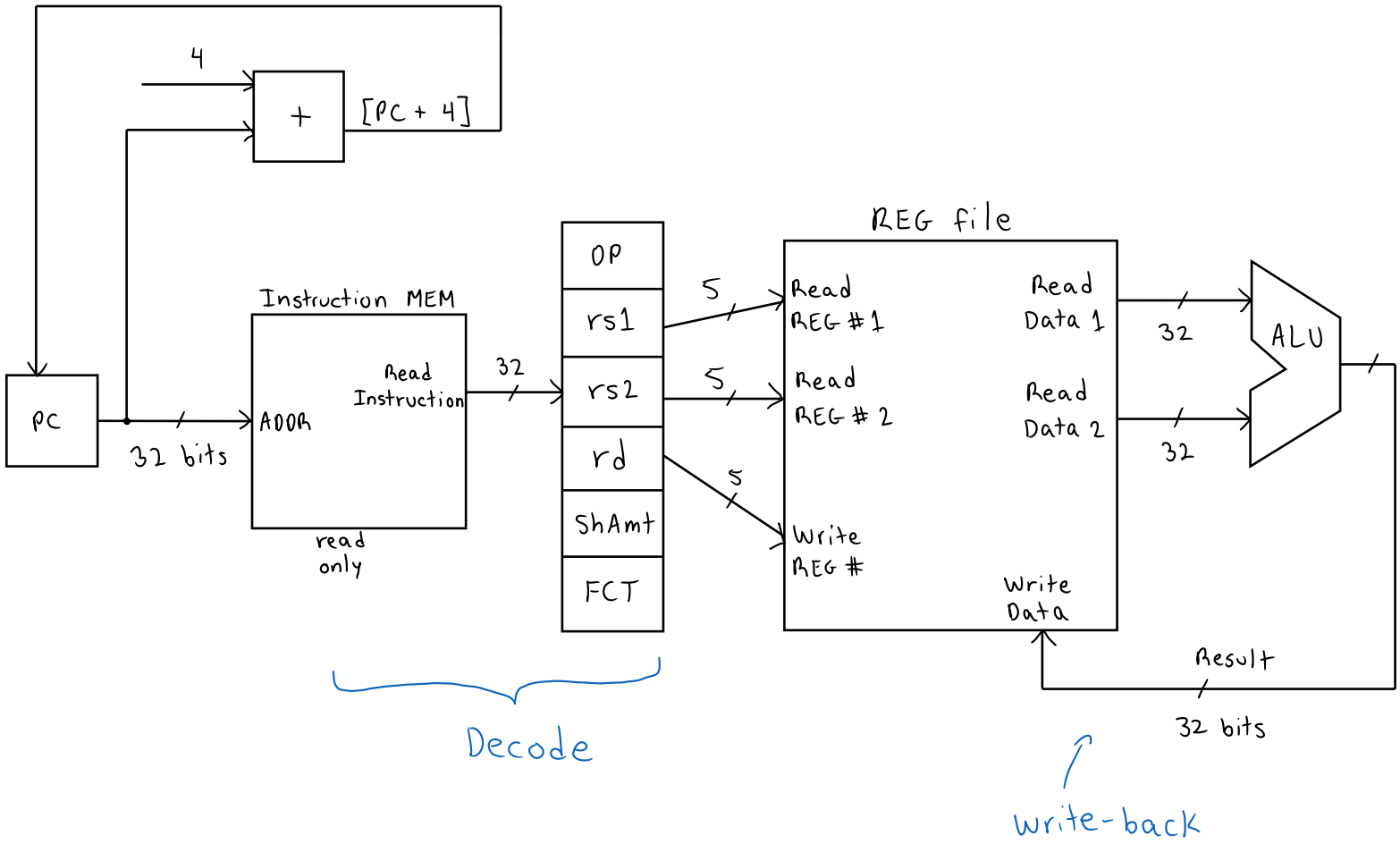
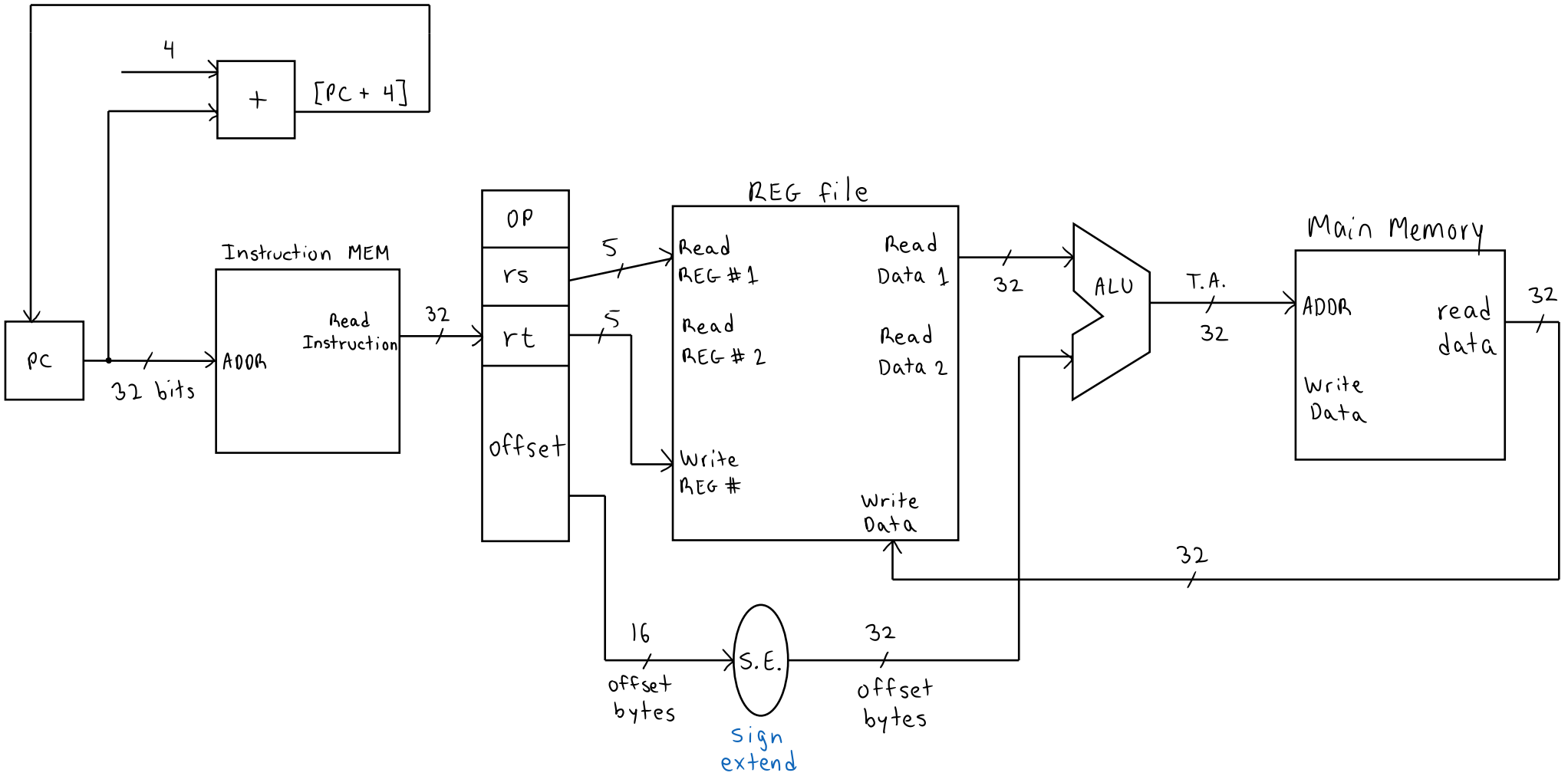


FIGURE 4.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexor (at the upper right) is used to choose between the jump target and either the branch target or the sequential instruction following this one. This multiplexor is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address.

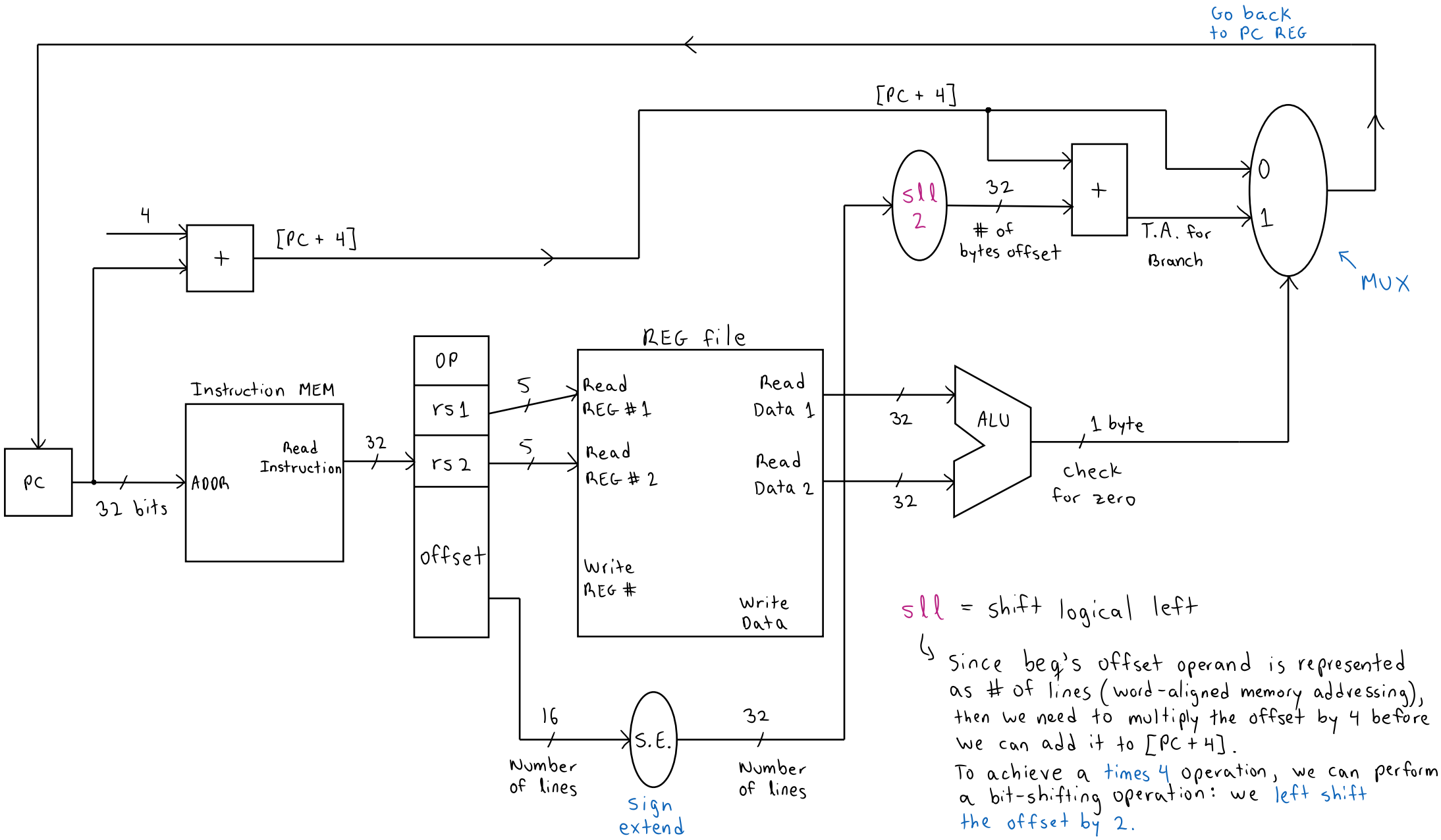
Datapath for R-format



Datapath for lw (Load Word)



Datapath for BEQ



sll = shift logical left

↳ Since beq's offset operand is represented as # of lines (word-aligned memory addressing), then we need to multiply the offset by 4 before we can add it to [PC + 4].

To achieve a times 4 operation, we can perform a bit-shifting operation: we left shift the offset by 2.

Register Field in Instructions

- First register (Read Register 1): Bits [25-21]
  - Used by all instructions to read the first operand
- Second register (Read Register 2): Bits [20-16]
  - Used by:
    - R-format instructions (eg. add, sub)
    - store (sw) to get the value to write memory
    - branch (beq)

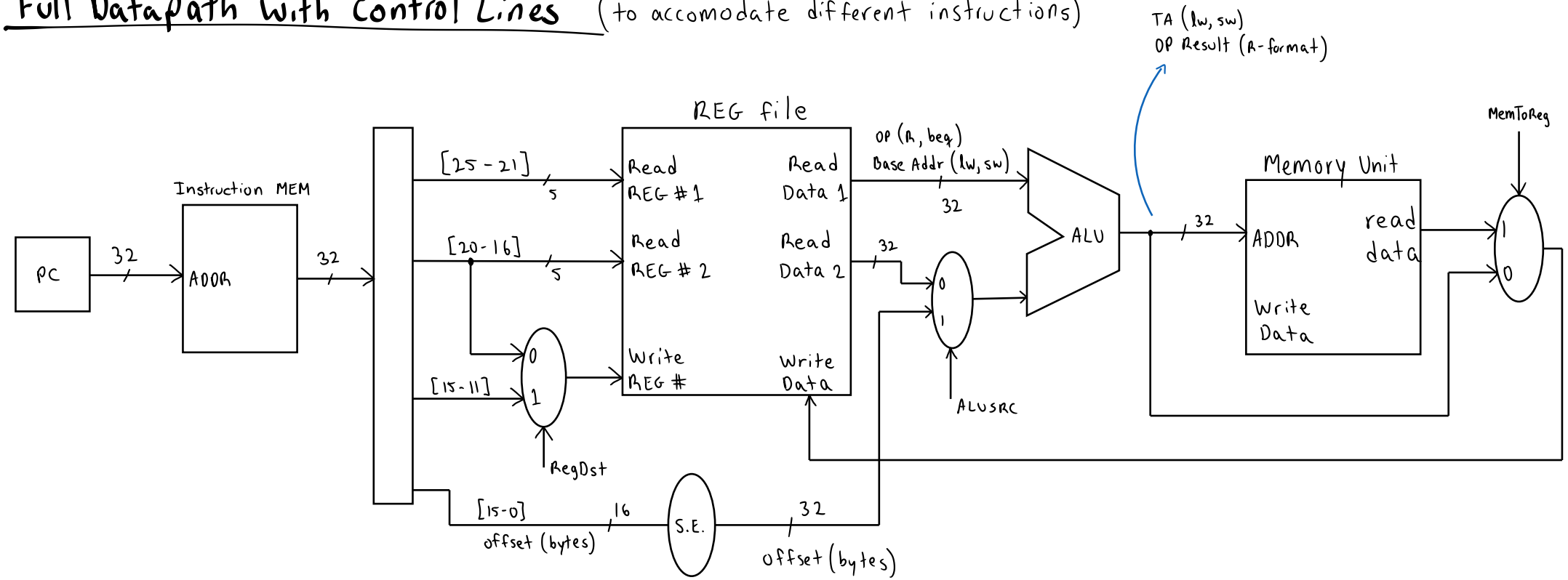
Write Registers

- Load instruction (lw)
  - Destination Register in bits [20-16]
- R-format
  - Destination Register in bits [15-11]

Notes for Exam 2

- Be able to draw the datapath for any individual instruction
- Know what each multiplexer does, inputs/outputs, and control line behavior
- For individual datapaths, include only relevant parts, no unnecessary control lines
- Be able to trace PC updates through the nested multiplexers using control line values

Full Datapath with Control Lines (to accomodate different instructions)



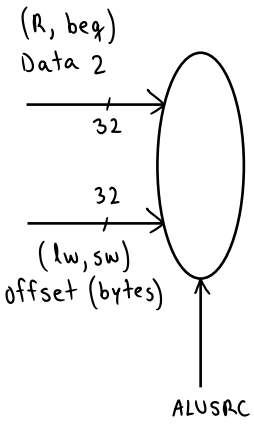
ALUSRC

R-format :] Data 2 with Reg # [20-26]

beq :

lw:] 32 bits offset (bytes)

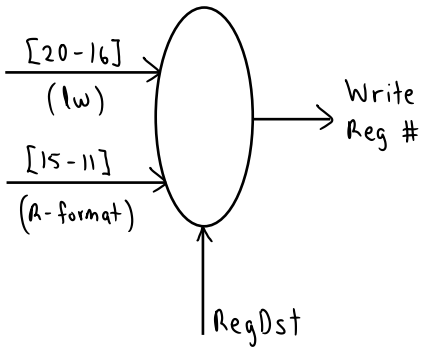
sw:



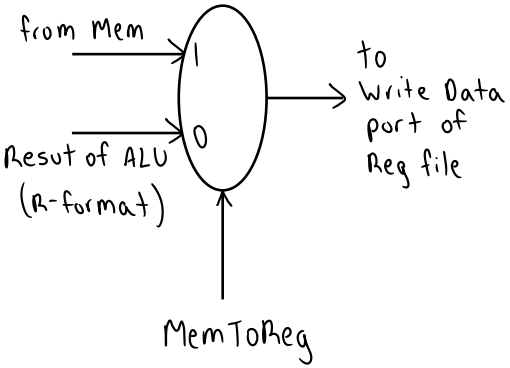
Write Reg

lw : [20-16]

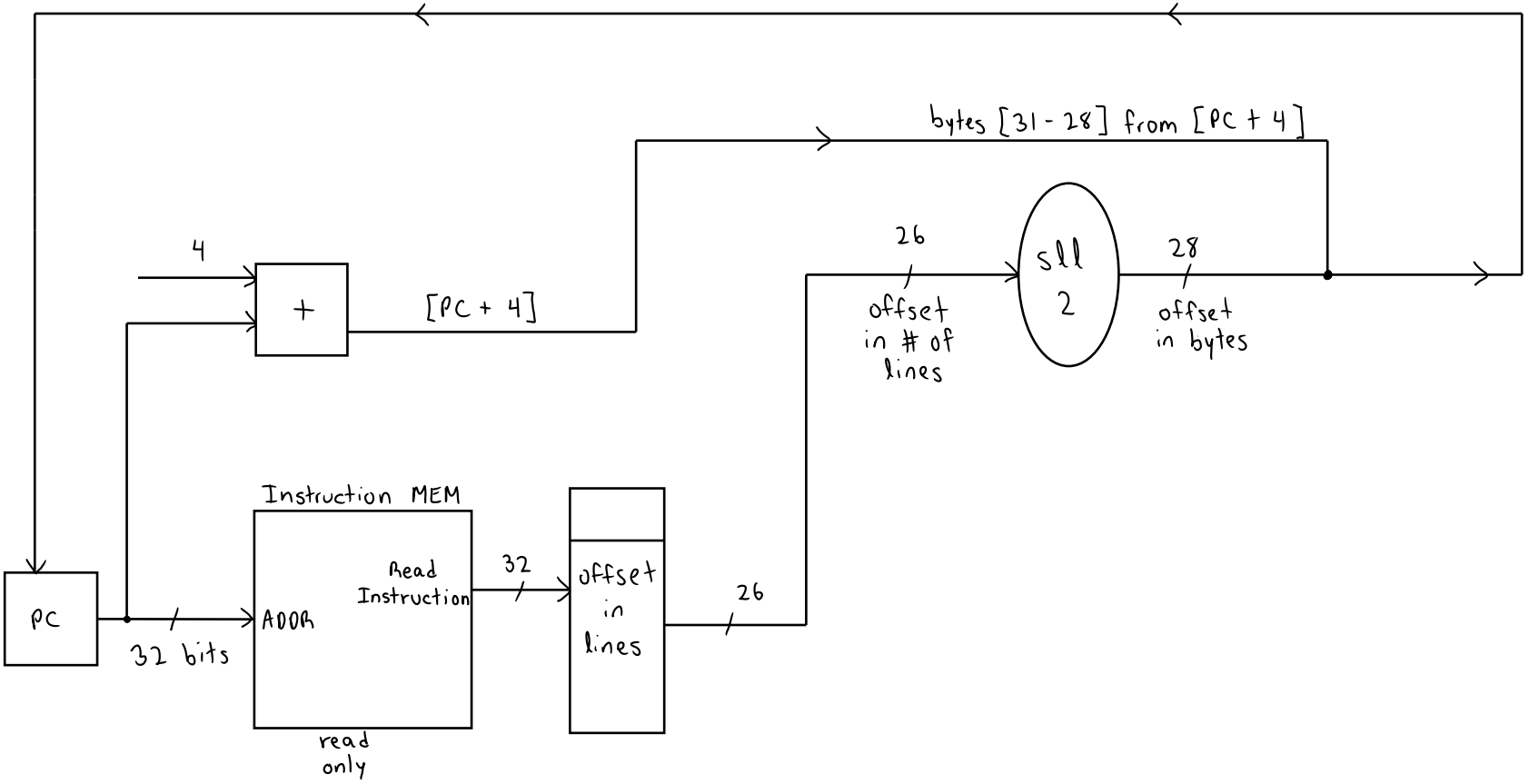
R-format : [15-11]



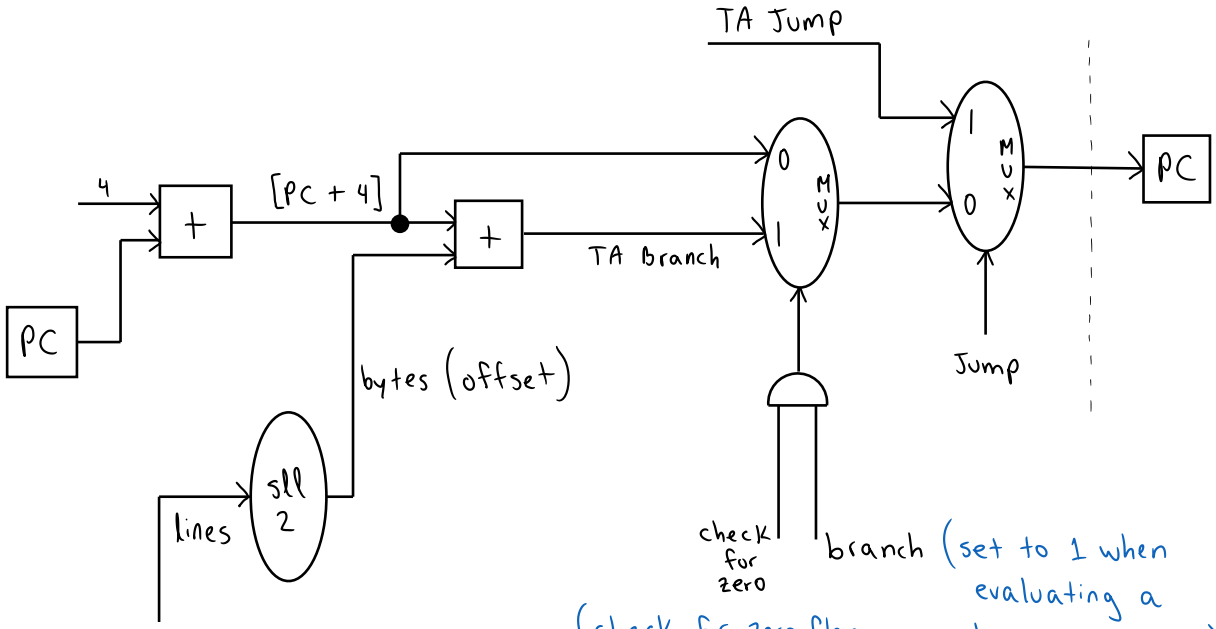
Write Data



Jump


bytes [31 - 28] from [PC + 4]
4
+
[PC + 4]
26
offset
in # of
lines
sll
2
28
offset
in bytes
Instruction MEM
PC
32 bits
ADDR
Read
Instruction
read
only
32
offset
in
lines
26

Control Logic for Updating PC Register



(check for zero flag is set whenever the ALU is used at all, not just when evaluating a Branch, so we need the AND gate to act as the control unit for the MUX)

- R-format, sw, lw:
  - Branch = 0
  - Jump = 0
  - We only update PC to [PC + 4]
- Branch Instruction:
  - Branch = 1
  - Jump = 0
  - We conditionally select either [PC + 4] or TA Branch depending on if check for zero is 0 or 1, respectively
- Jump Instruction:
  - Branch = 0
  - Jump = 1
  - We just update PC to TA Jump

From Patterson Appendix

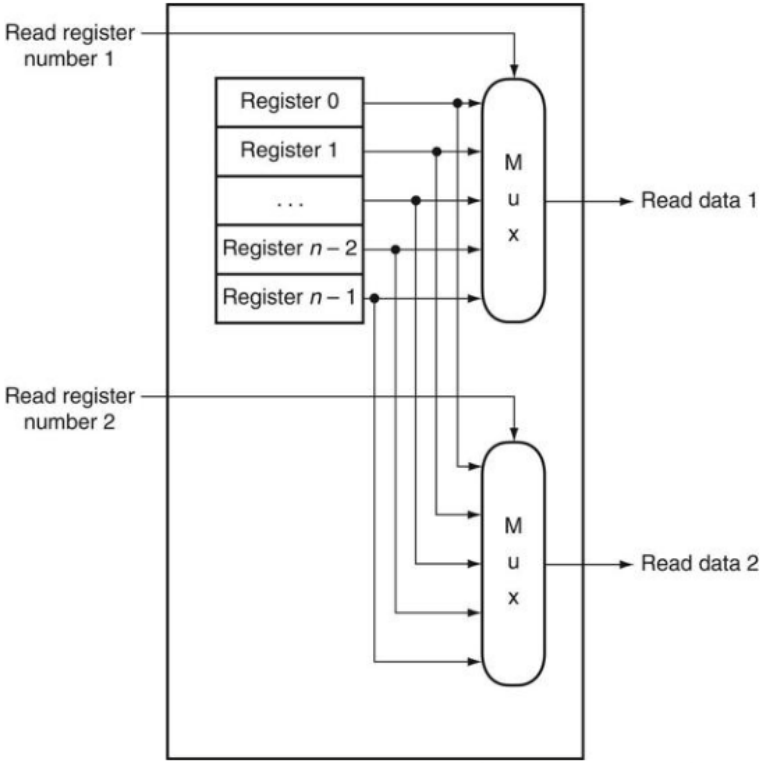


FIGURE B.8.8 The implementation of two read ports for a register file with *n* registers can be done with a pair of n-to-1 multiplexors, each 32 bits wide.
The register read number signal is used as the multiplexor selector signal. Figure B.8.9 shows how the write port is implemented.

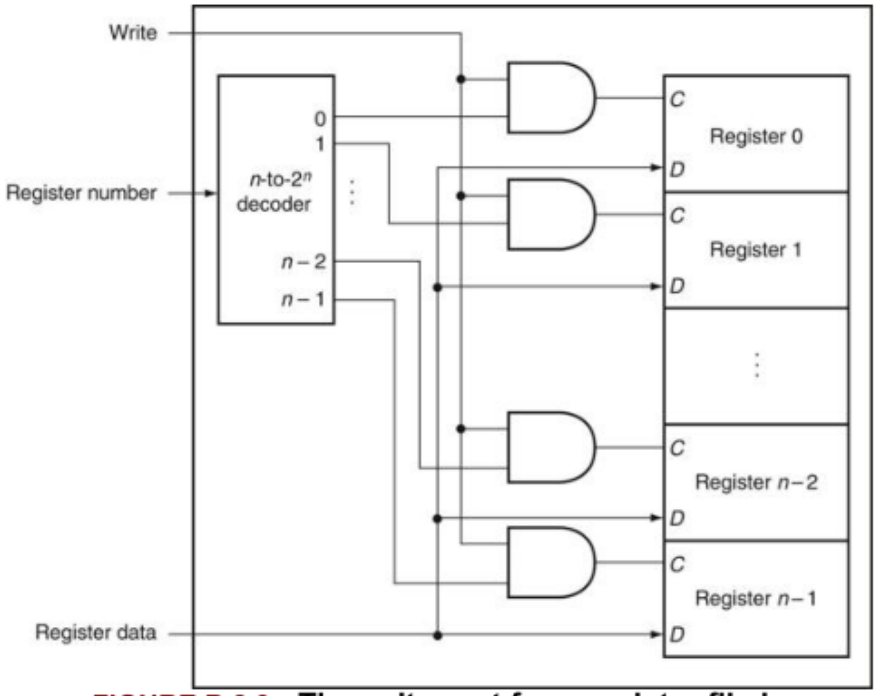


FIGURE B.8.9 The write port for a register file is implemented with a decoder that is used with the write signal to generate the *C* input to the registers.
All three inputs (the register number, the data, and the write signal) will have setup and hold-time constraints that ensure that the correct data is written into the register file.

Implementing the write port is slightly more complex, since we can only change the contents of the designated register. We can do this by using a decoder to generate a signal that can be used to determine which register to write. Figure B.8.9 shows how to implement the write port for a register file. It is important to remember that the flip-flop changes state only on the clock edge. In Chapter 4, we hooked up write signals for the register file explicitly and assumed the clock shown in Figure B.8.9 is attached implicitly.

What happens if the same register is read and written during a clock cycle? Because the write of the register file occurs on the clock edge, the register will be valid during the time it is read, as we saw earlier in Figure B.7.2. The value returned will be the value written in an earlier clock cycle. If we want a read to return the value currently being written, additional logic in the register file or outside of it is needed. Chapter 4 makes extensive use of such logic.

Datapath Control Lines

- Mux
  - RegDst
  - MemToReg
  - ALUSRC
  - Jump
  - Branch
- Access
  - RegWrite
- Memory
  - MemWrite
  - MemRead

	R-format	Load	Store	Branch	Jump
RegDst	1	0	X	X	X
MemToReg	0	1	X	X	X
ALUSRC	0	1	1	0	X
Branch	0	0	0	1	0
Jump	0	0	0	0	1
RegWrite	1	1	0	0	0
MemWrite	0	0	1	0	0
MemRead	0	1	0	0	0

Single Cycle Implementation: Performance

The clock cycle has the same length for every instruction

CPI = 1

The clock cycle is determined by the longest possible path. Let's find out what is the instruction that takes the longest time:

* The Adder for PC is not considered to take any time.
* Access times:
  - Memory Unit: 200 ps
  - ALU, Adders: 100 ps
  - Register file: 50 ps

R-format

fetch: Read Instruction memory	⇒	200 ps
Decode: Read Reg file (read 2 source reg in parallel)	⇒	50 ps
Compute: Access ALU	⇒	100 ps
Write Back: Write into the Reg file	⇒	50 ps
		400 ps

Store

fetch: Read Instruction memory	⇒	200 ps
Decode: Read Reg file (read 2 source reg in parallel)	⇒	50 ps
Compute: Access ALU (compute target Address)	⇒	100 ps
Memory Access (write data to memory)	⇒	200 ps
		550 ps

Load

fetch: Read Instruction memory	⇒	200 ps
Decode: Read Reg file (reads only 1 source Address)	⇒	50 ps
Compute: Access ALU (compute target Address)	⇒	100 ps
Memory Access (read data from memory)	⇒	200 ps
Write Back (write to Reg file)	⇒	50 ps
		600 ps

Beq

fetch: Read Instruction memory	⇒	200 ps
Decode: Read Reg file (read 2 source reg in parallel)	⇒	50 ps
Compute: Access ALU (check for zero) ↖ Done in Parallel	⇒	100 ps
Compute: Target Address (use an Adder)		350 ps

* A program with 50 instructions: 50 × length of lw (since lw is the longest instruction)