Computer Organization & Architecture

Logic Designing and building Datapath

Introduction

- MIPS implementations
  - A simplified version
  - A more realistic pipelined version
- Simple subset, shows most aspects
  - Arithmetic/logical: `add`, `sub`, `and`, `or`, `slt`
  - Memory reference: `lw`, `sw`
  - Control transfer: `beq`, `j`
- Building the Processor incrementally
  - Building R-Type Datapath
  - Building I-Type Datapath
    - Building for Load/Store
    - Building for Addi
    - Building for BEQ
  - Building J-Type Datapath
  - Building Branch Datapath
  - Controlling operations in Processor

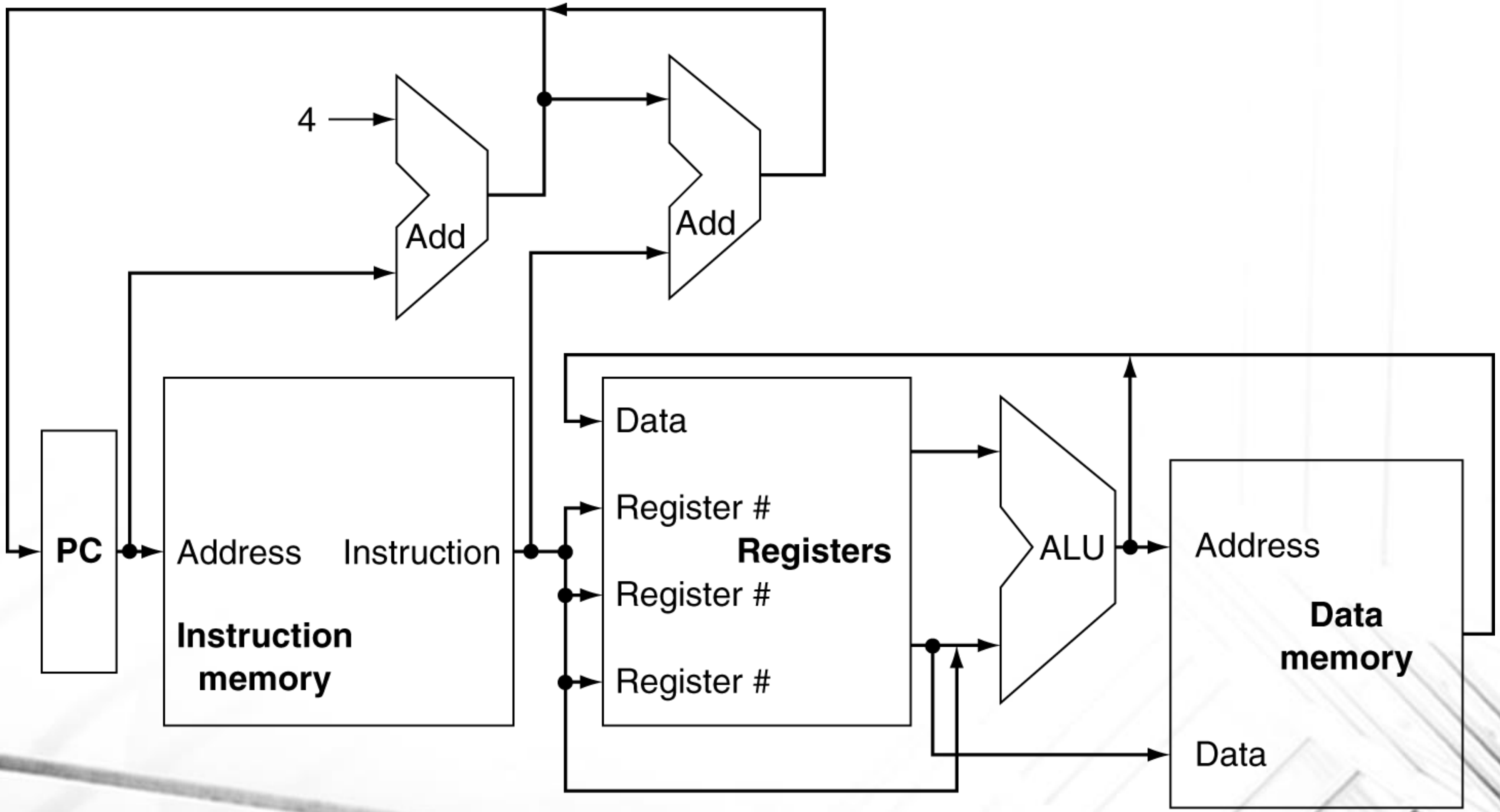
Logic Design Basics

- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Instruction Execution

1) Fetch Instruction → Based upon the address in PC from the Instruction Memory
2) Decode Instruction → Through Control Unit and read operand from the Register File
3) Execute Instruction → Depending on instruction class use ALU to calculate
   a) Arithmetic result
   b) Memory address for load/store
   c) Branch target address
4) Memory Access → Access Data Memory if the instruction is of memory reference
5) Write Back → Result is written back to the Register File

Datapath Overview



Register files

- Built using decoders, and flip flops
- For reading
  - Need only register number
- For writing
  - Need register number, data, clock

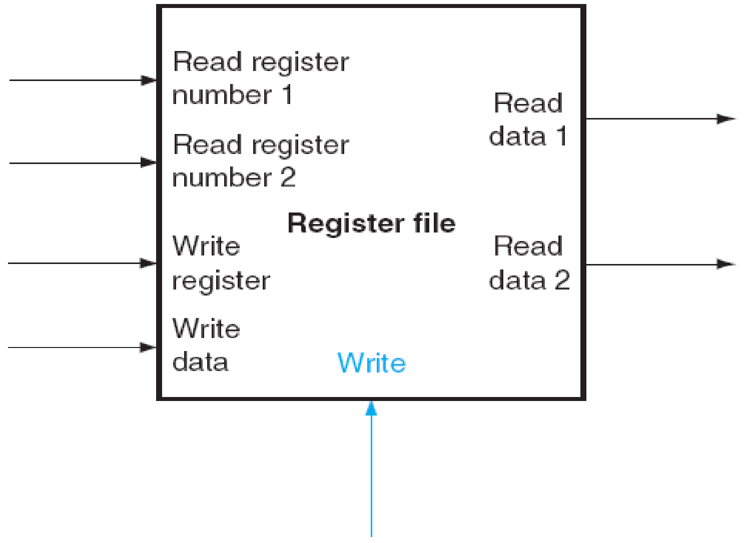


FIGURE C.8.7 A register file with two read ports and one write port has five inputs and two outputs. The control input Write is shown in color.

Implementing the read port

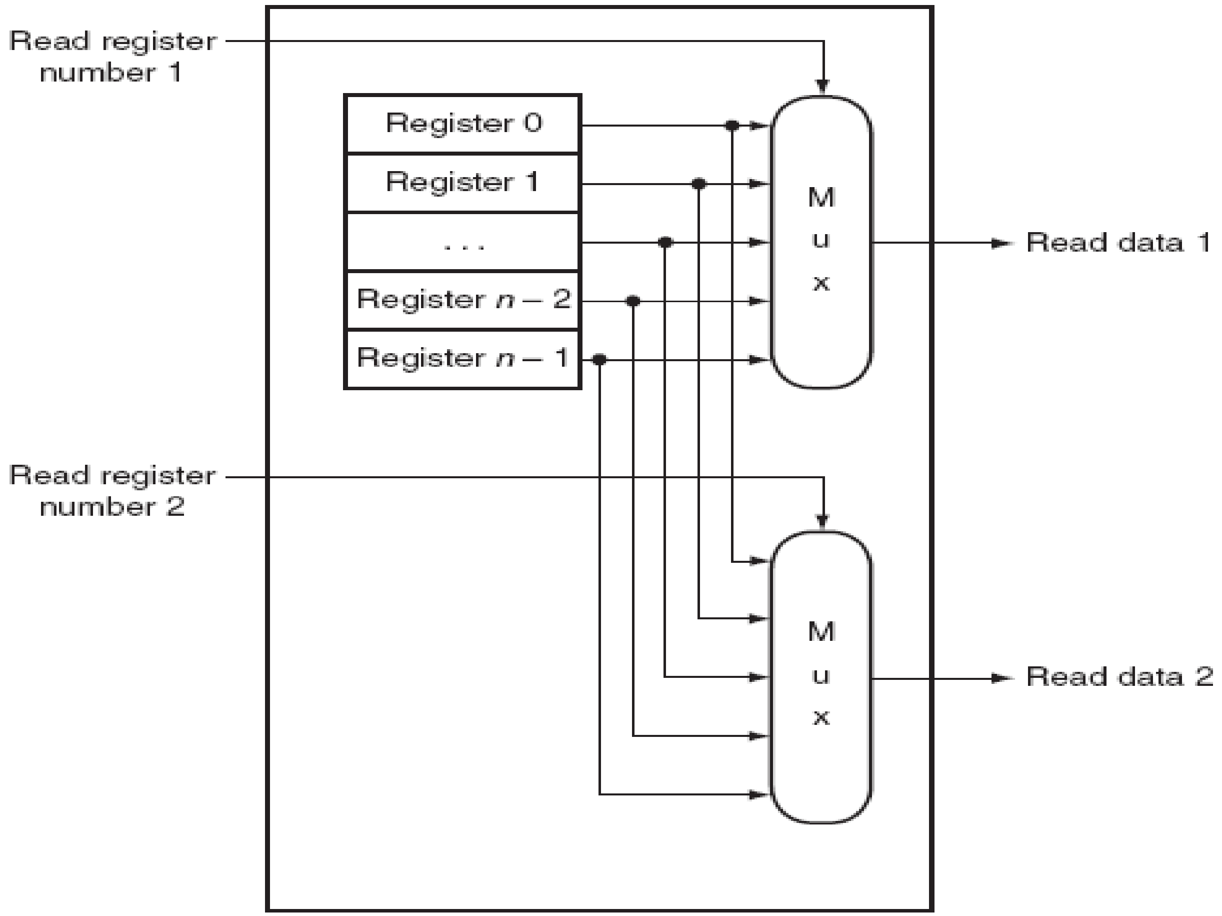


FIGURE C.8.8 The implementation of two read ports for a register file with *n* registers can be done with a pair of *n*-to-1 multiplexors, each 32 bits wide. The register read number signal is used as the multiplexor selector signal. Figure C.8.9 shows how the write port is implemented.

Implementing the write port

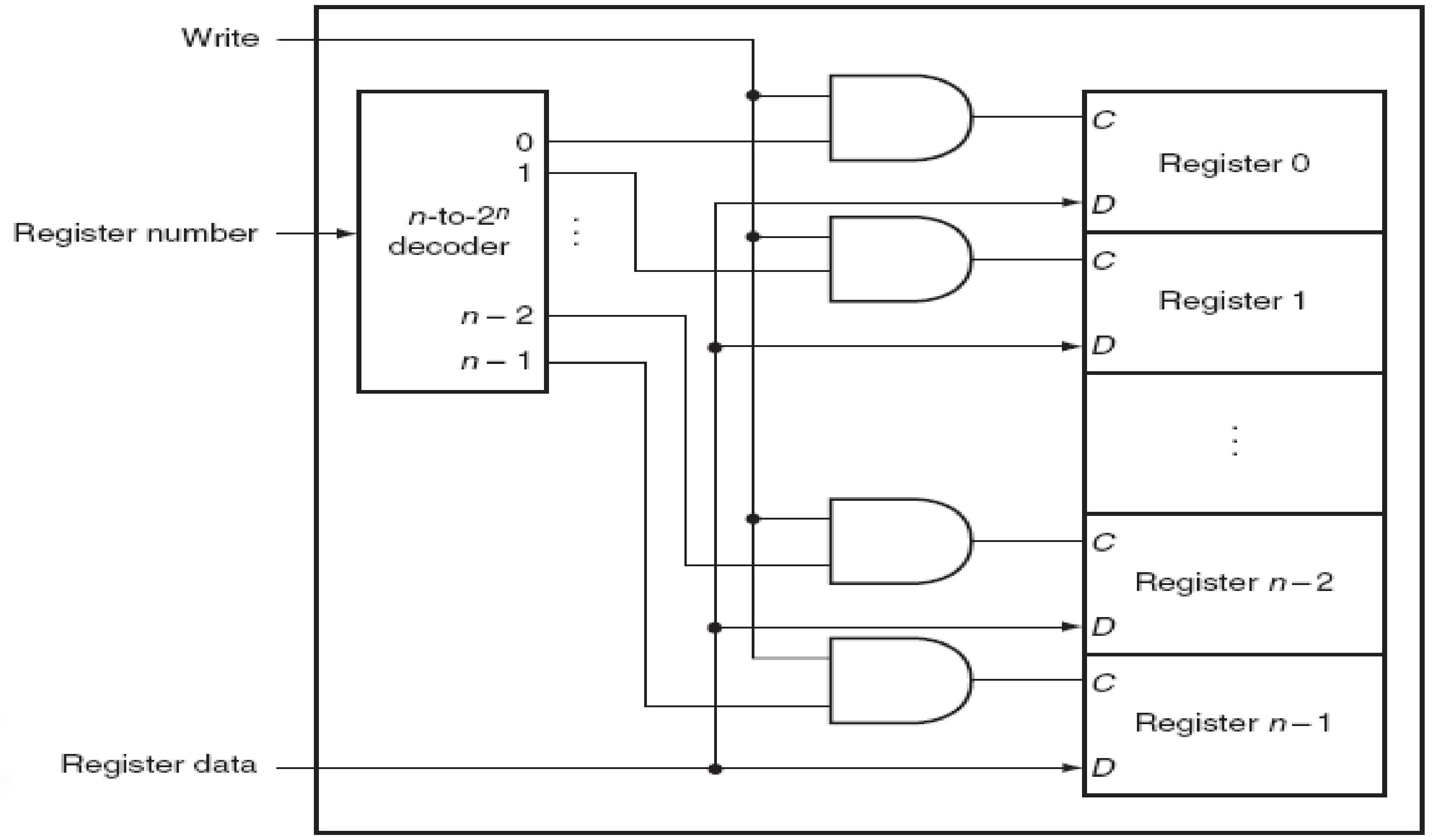


FIGURE C.8.9 The write port for a register file is implemented with a decoder that is used with the write signal to generate the *C* input to the registers. All three inputs (the register number, the data, and the write signal) will have setup and hold-time constraints that ensure that the correct data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

Memory elements

- Registers are used for small memories only.
- For large memories we need either SRAMs or DRAMs
- SRAM: Static Random Access Memory
- DRAM: Dynamic Random Access Memory

SRAM: Static Random Access Memory

- Fixed time to read any data
- Specific configuration in terms of number of addressable locations and width of each location
- Typical block diagram of 2Mx16 SRAM

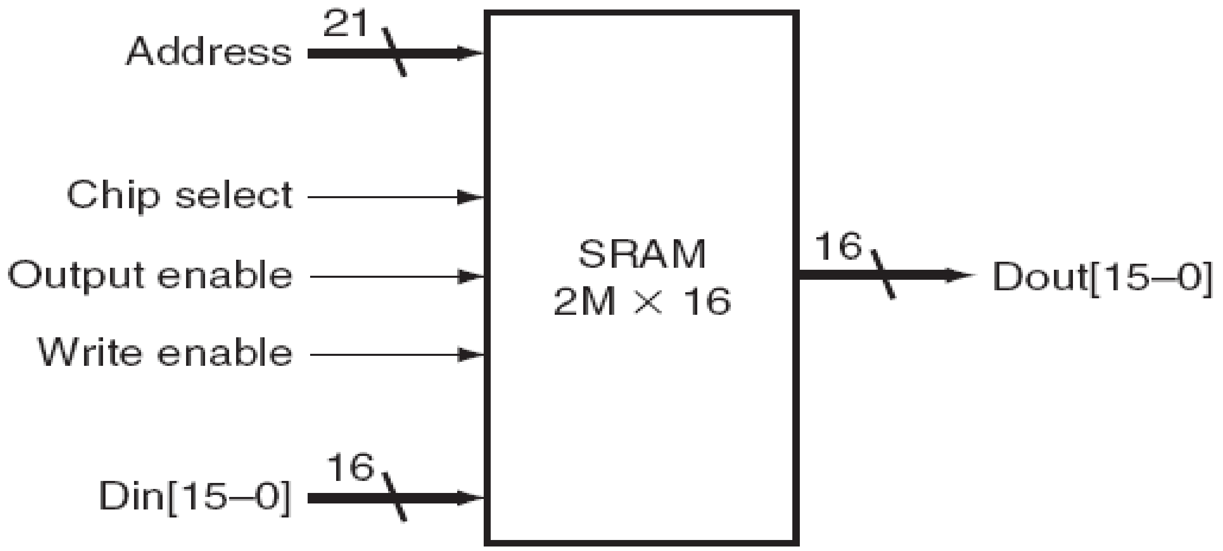


FIGURE C.9.1 **A 32K × 8 SRAM showing the 21 address lines (32K = 2^{15}) and 16 data inputs, the 3 control lines, and the 16 data outputs.**

Read and Write in SRAMs

- For read
  - Chip select
  - Output enable: useful to connect multiple memories to a single bus
  - Address
  - Typical read access time: 2 – 4 nS
- For write
  - Chip select
  - Data to be written
  - Write enable
  - Address
- Address selection is a problem in SRAM
- If we use register file strategy
  - For a 4Mx8 SRAM, we need a MUX with 4M inputs each 8bits wide and 22 selection lines
- How about using a decoder?
  - Single level decoding
  - Two level decoding

Addressing in SRAMs

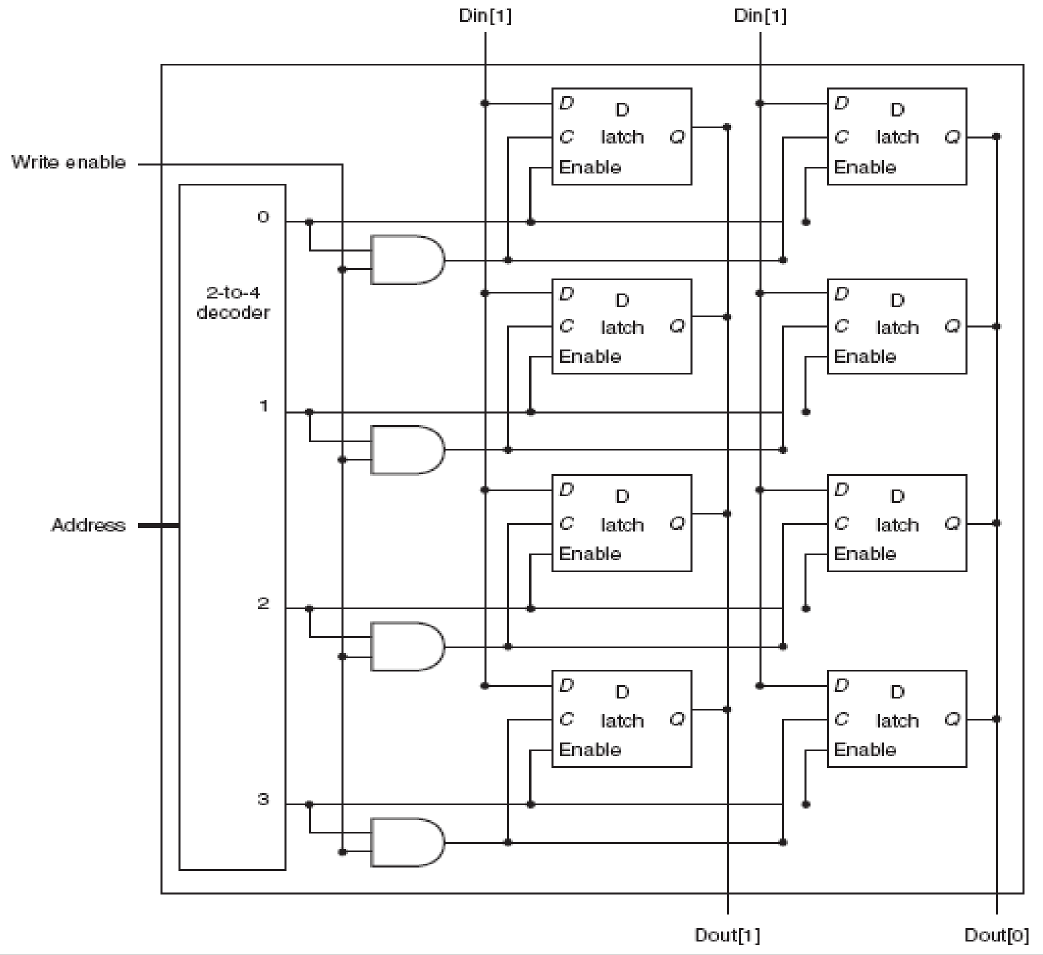


FIGURE C.9.3 The basic structure of a 4 × 2 SRAM consists of a decoder that selects which pair of cells to activate. The activated cells use a three-state output connected to the vertical bit lines that supply the requested data. The address that selects the cell is sent on one of a set of horizontal address lines, called word lines. For simplicity, the Output enable and Chip select signals have been omitted, but they could easily be added with a few AND gates.

Addressing in SRAMs

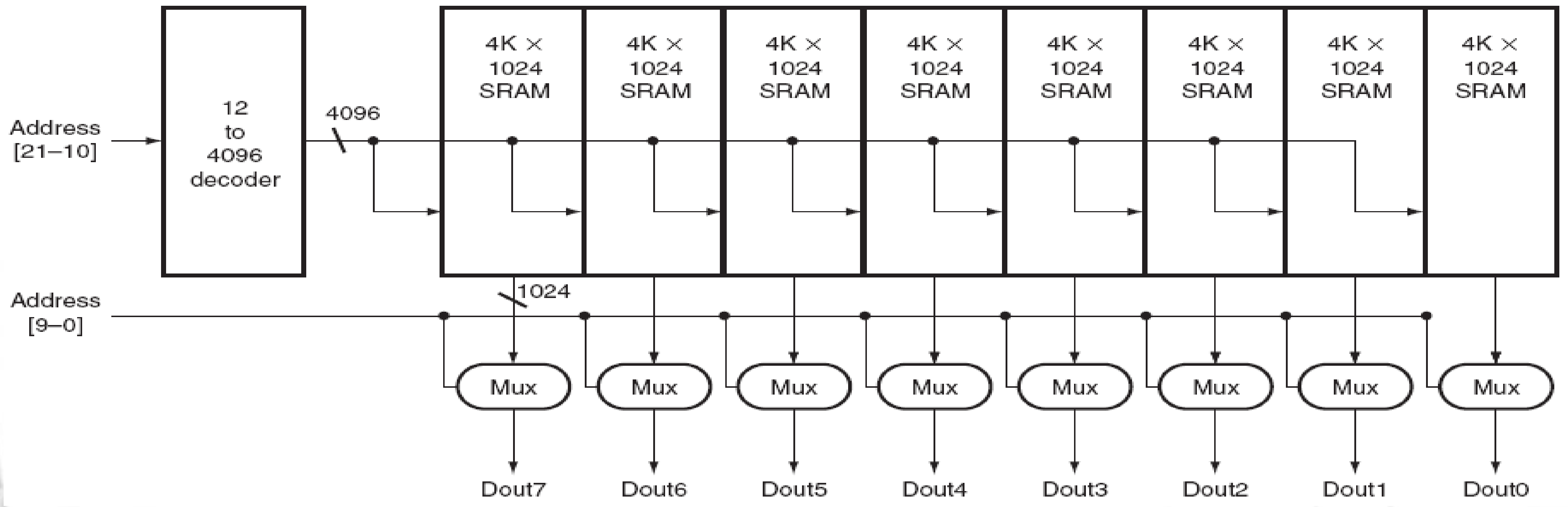


FIGURE C.9.4 Typical organization of a 4M × 8 SRAM as an array of 4K × 1024 arrays. The first decoder generates the addresses for eight 4K × 1024 arrays; then a set of multiplexors is used to select 1 bit from each 1024-bit-wide array. This is a much easier design than a single-level decode that would need either an enormous decoder or a gigantic multiplexor. In practice, a modern SRAM of this size would probably use an even larger number of blocks, each somewhat smaller.

DRAM: Dynamic Random Access Memory

- In SRAM value is stored using a pair of inverter gates which are constructed using around 6 transistors
- In DRAMs, value is stored using a capacitor and only a single transistor is needed
- DRAMs are much cheaper and denser as compared to SRAMs
- Value(charge) must be refreshed periodically in DRAMs
- Refreshing means first reading and then writing the value
- Refreshing rate in milliseconds
- Performed independently of processor

Read/Write in DRAMS

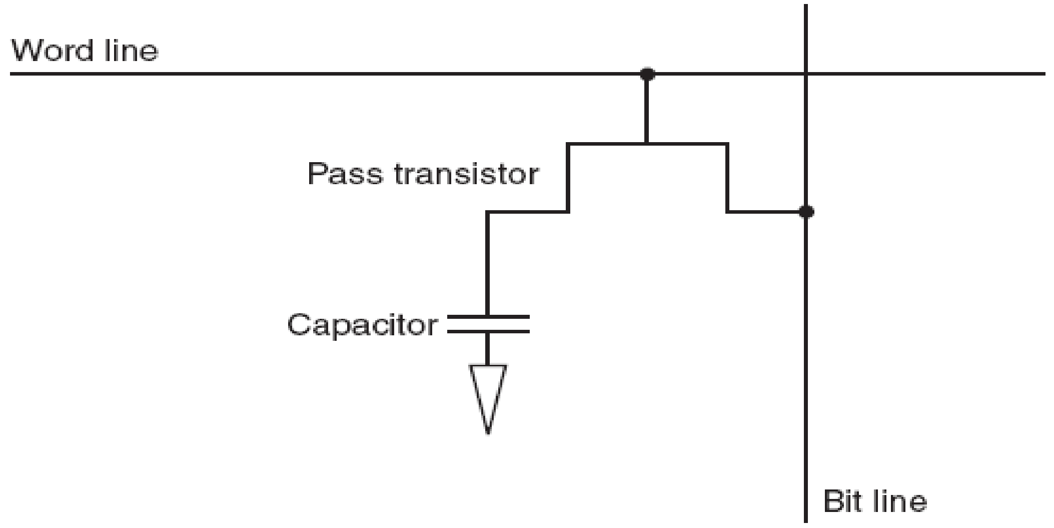


FIGURE C.9.5 A single-transistor DRAM cell contains a capacitor that stores the cell contents and a transistor used to access the cell.

- When the ‘Word line’ signal is asserted, switch is closed connecting bit line to capacitor
- For write place the value to be written on bit line
- For read, bit line is first charged to half way, word line is then asserted
- Based on the charge stored in capacitor bit line value either increases or decreases.

Addressing in DRAMs

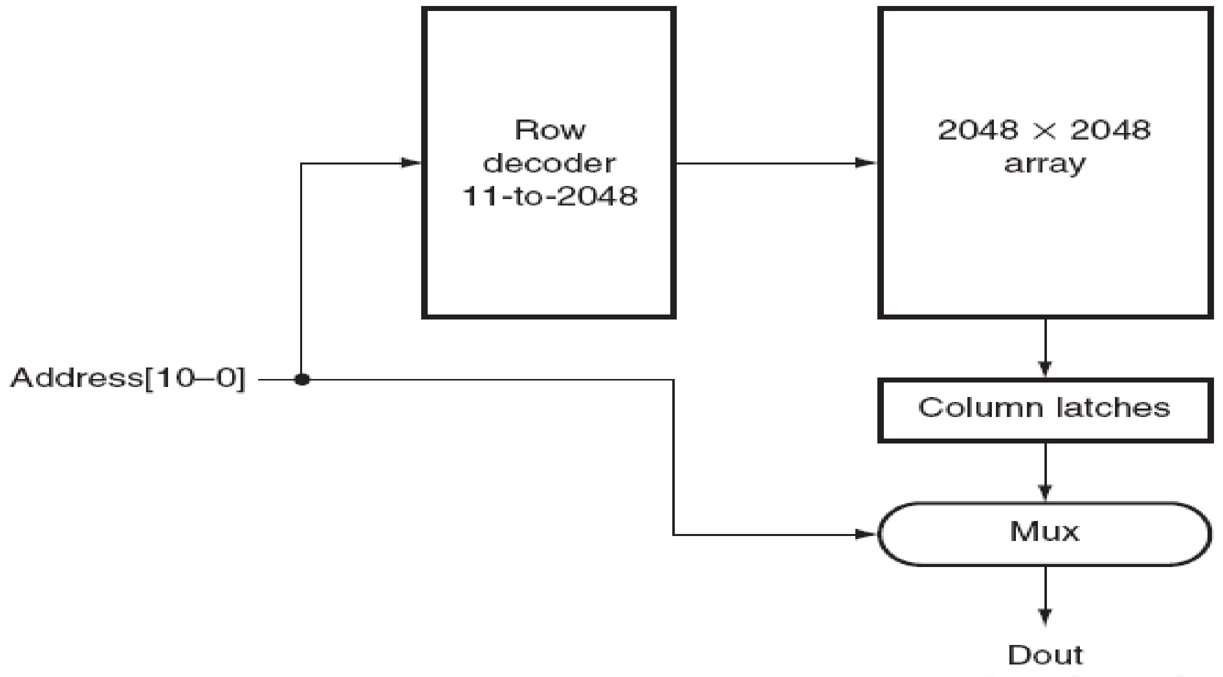
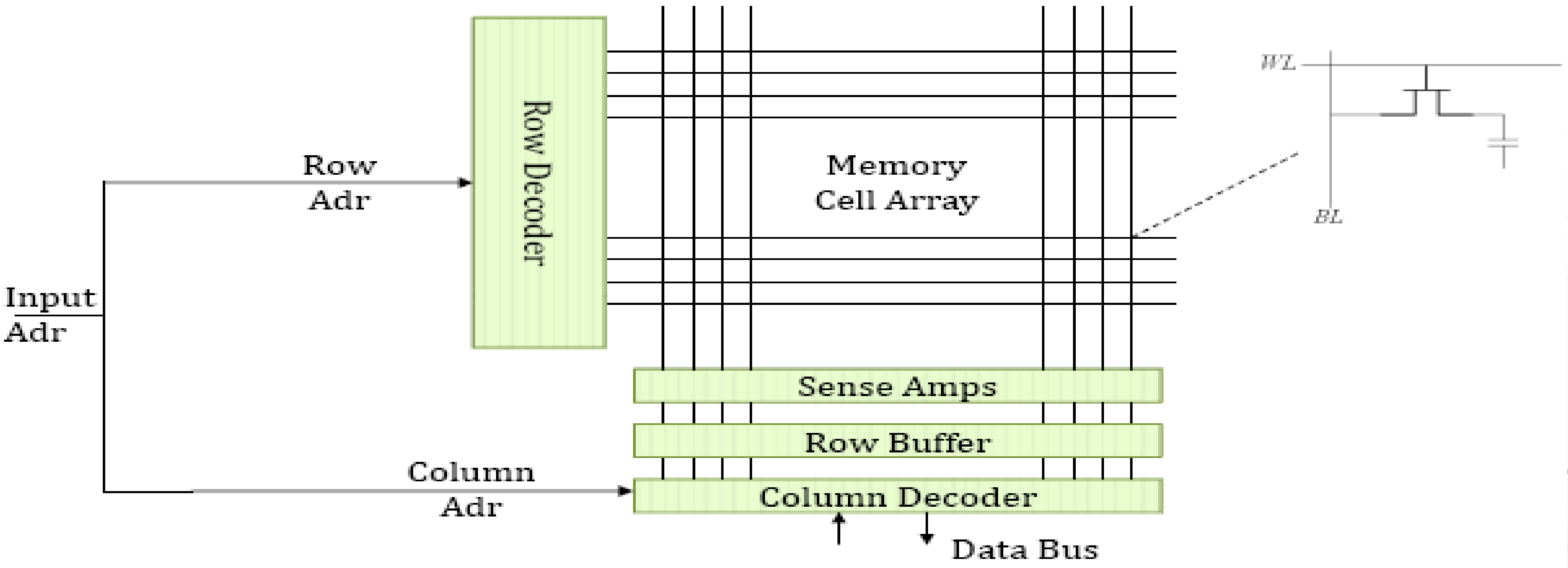


FIGURE C.9.6 A 4M × 1 DRAM is built with a 2048 × 2048 array. The row access uses 11 bits to select a row, which is then latched in 2048 1-bit latches. A multiplexor chooses the output bit from these 2048 latches. The RAS and CAS signals control whether the address lines are sent to the row decoder or column multiplexor.

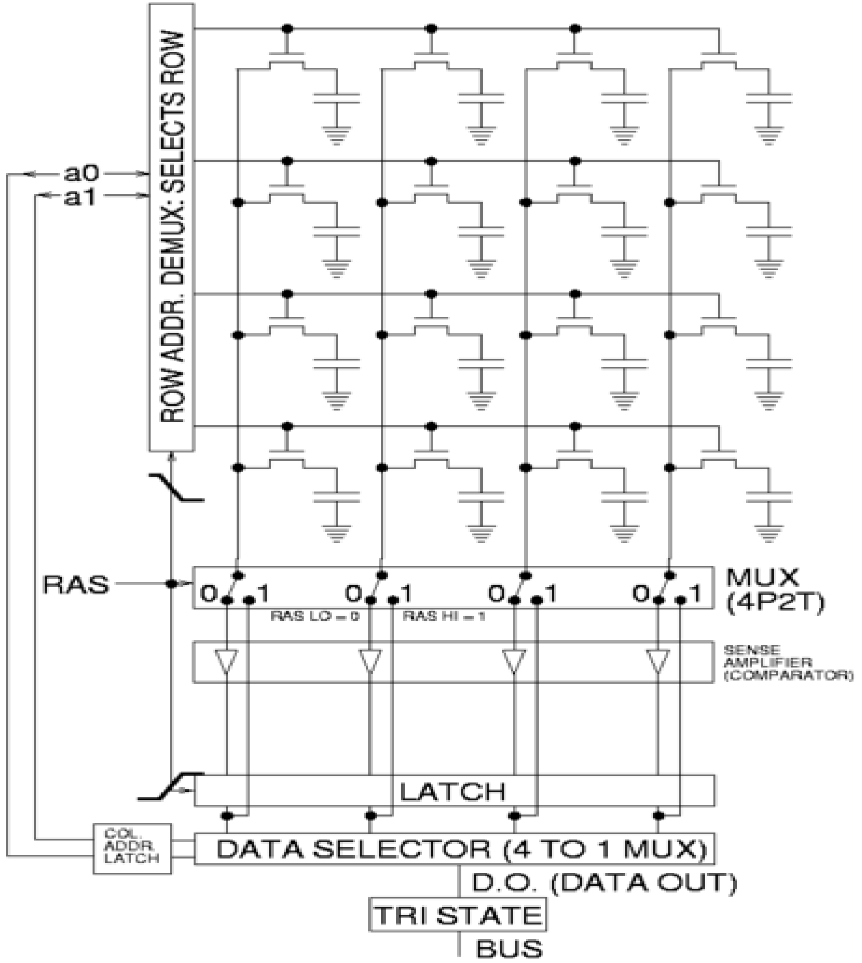
- DRAM uses two level decoding
- MSBs used for Row access
- LSBs used for column access
- Same line used for row access and column access to save resources

DRAM chip organization



26

DRAM chip organization



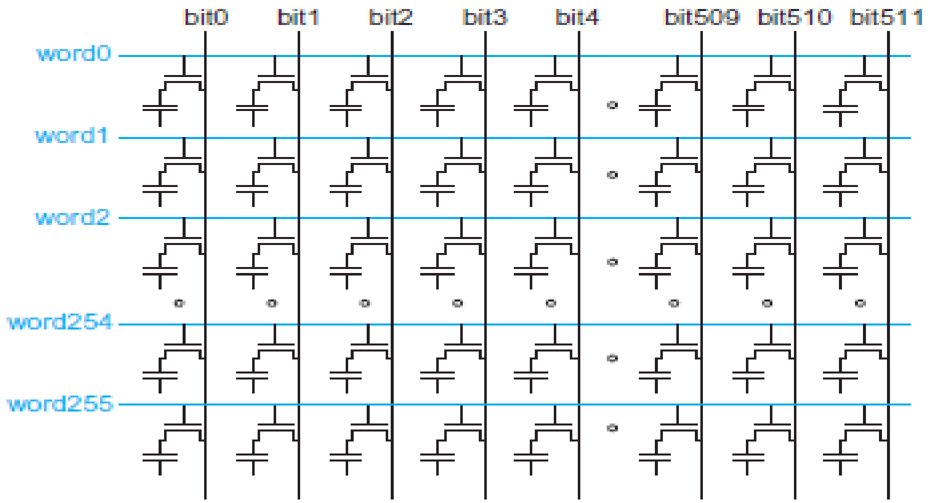
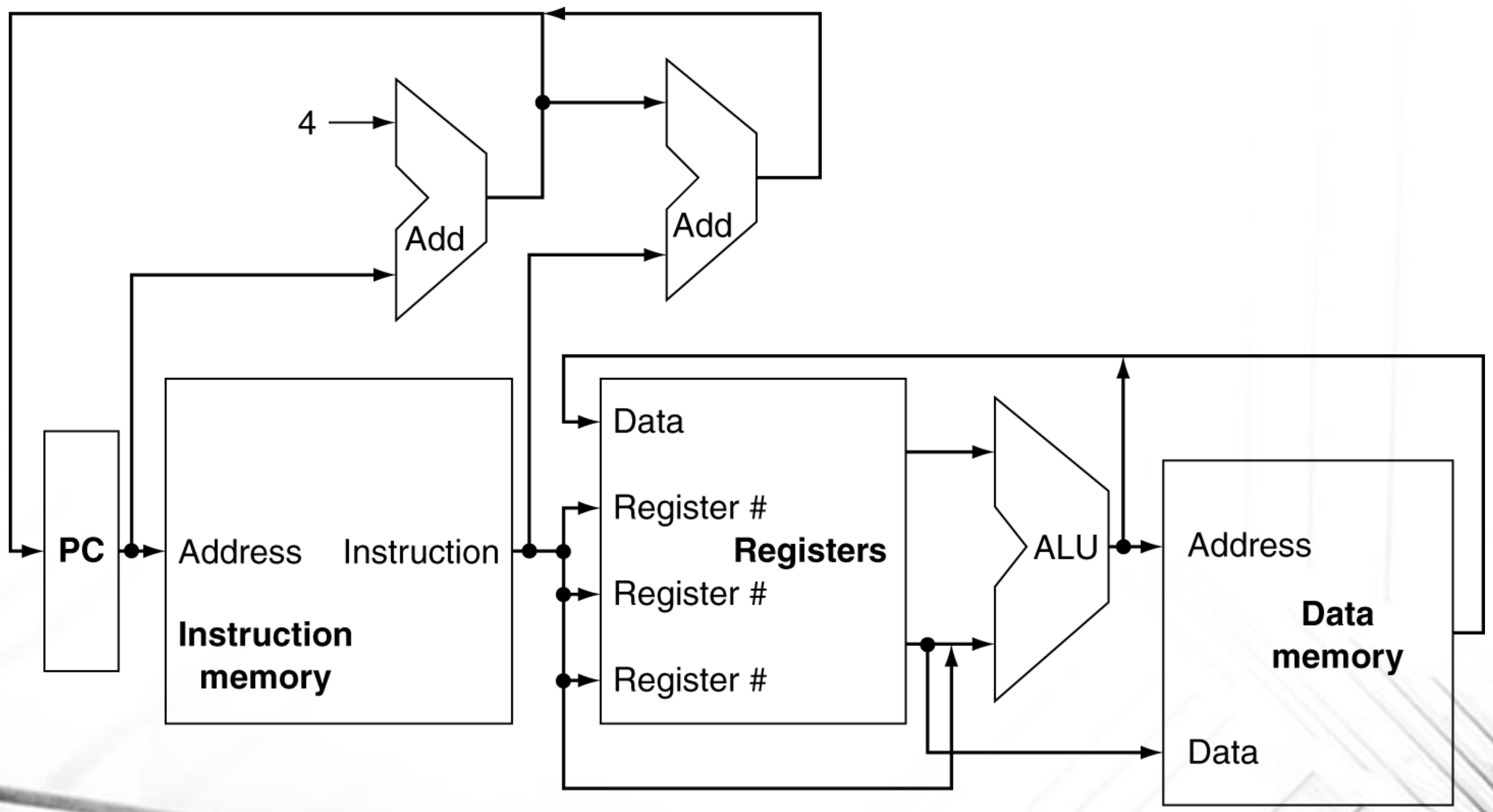


FIGURE 12.43 DRAM subarray

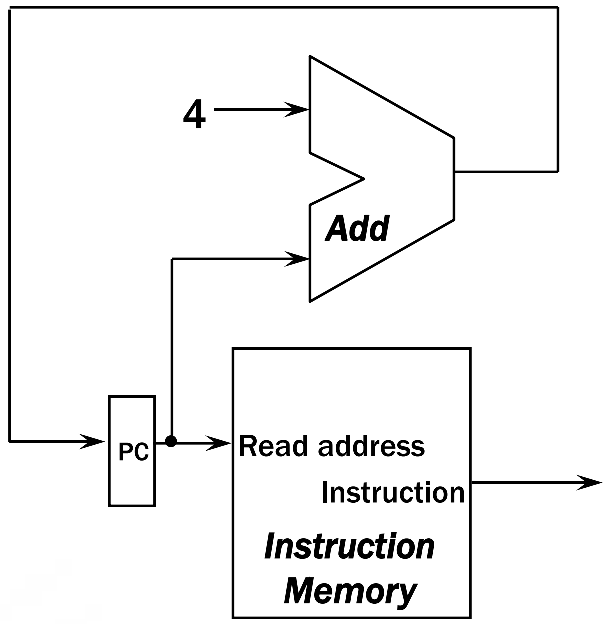
Building a Datapath

- Datapath
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, MUXes, memories, …
- We will build a MIPS datapath incrementally
  - Refining the overview design

CPU Overview... Single-Cycle Design



Instruction Datapath

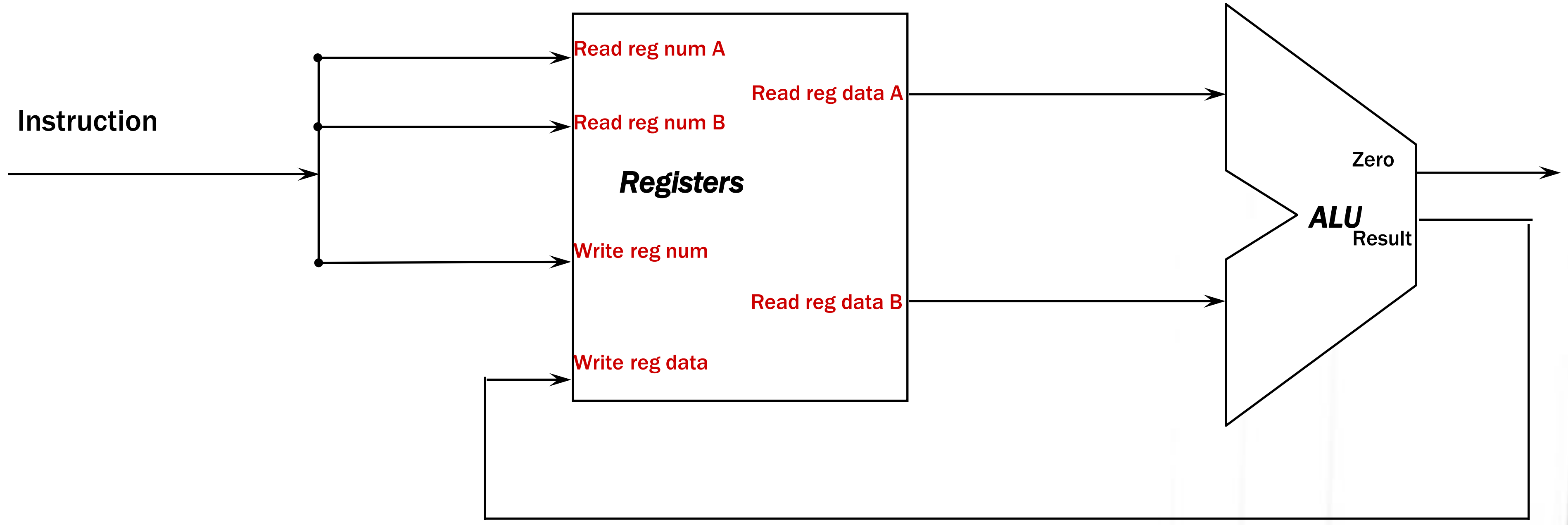


Note: Regular instruction width (32 for MIPS) makes this easy

- Instructions will be held in the instruction memory
- The instruction to fetch is at the location specified by the PC
  - Instr. = M[PC]

- **After we fetch one instruction, the PC must be incremented to the next instruction**
  - All instructions are 4 bytes
  - PC = PC + 4

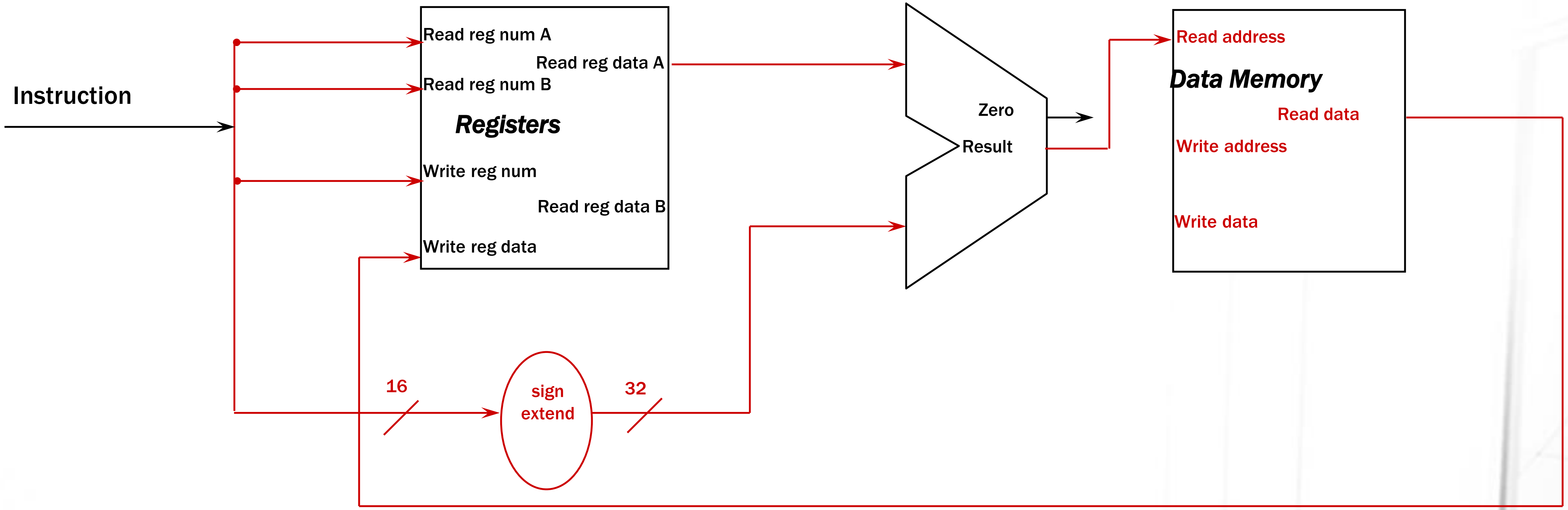
R-type Instruction Datapath



- R-type Instructions have three registers
  - Two read (Rs, Rt) to provide data to the ALU
  - One write (Rd) to receive data from the ALU

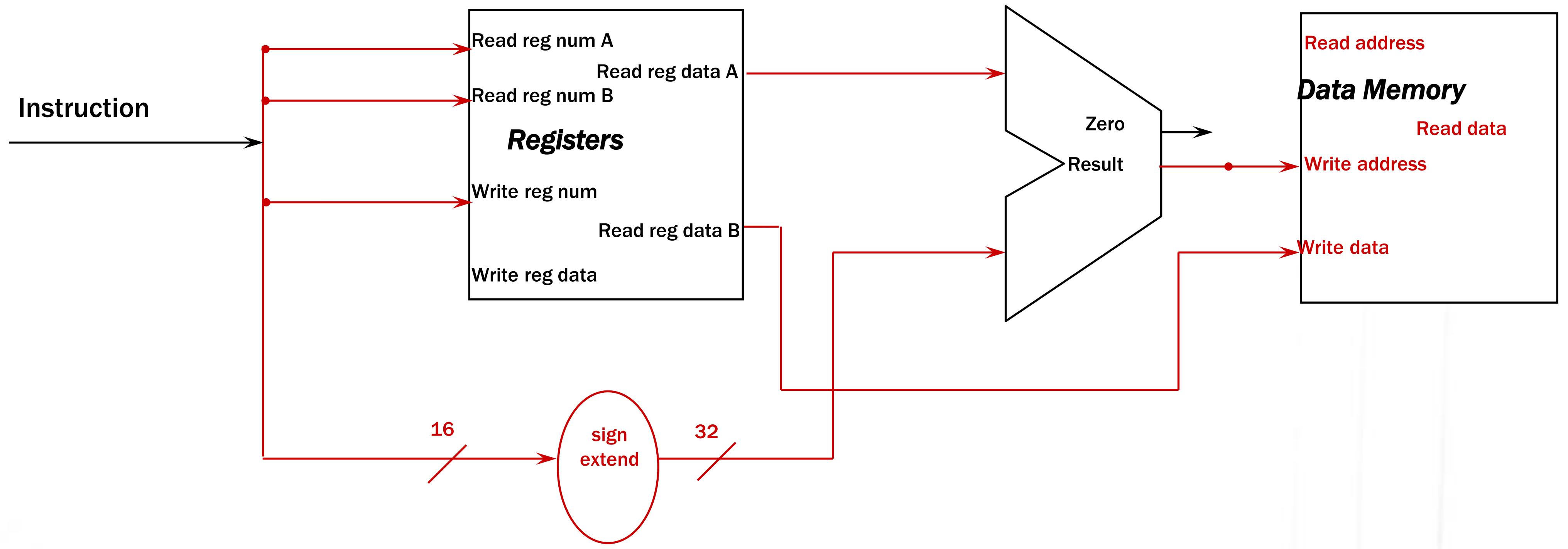
- We'll need to specify the operation to the ALU (later...)
- We might be interested if the result of the ALU is zero (later...)

Memory Operations (Load Word)



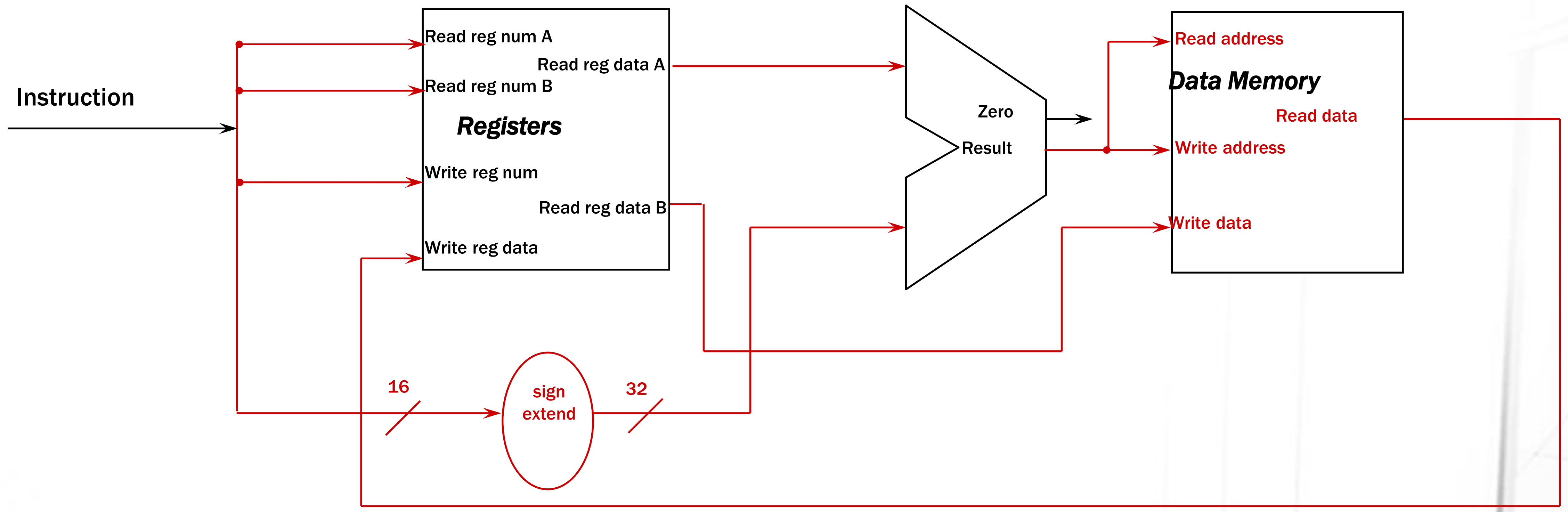
- Memory operations first need to compute the effective address
  - LW $t1, 450($s3) # E.A. = 450 + [$s3]
  - Add together one register and 16 bits of immediate data
  - Immediate data needs to be converted from 16-bit to 32-bit
- Memory then performs load or store using destination register

Memory Operations (Store Word)

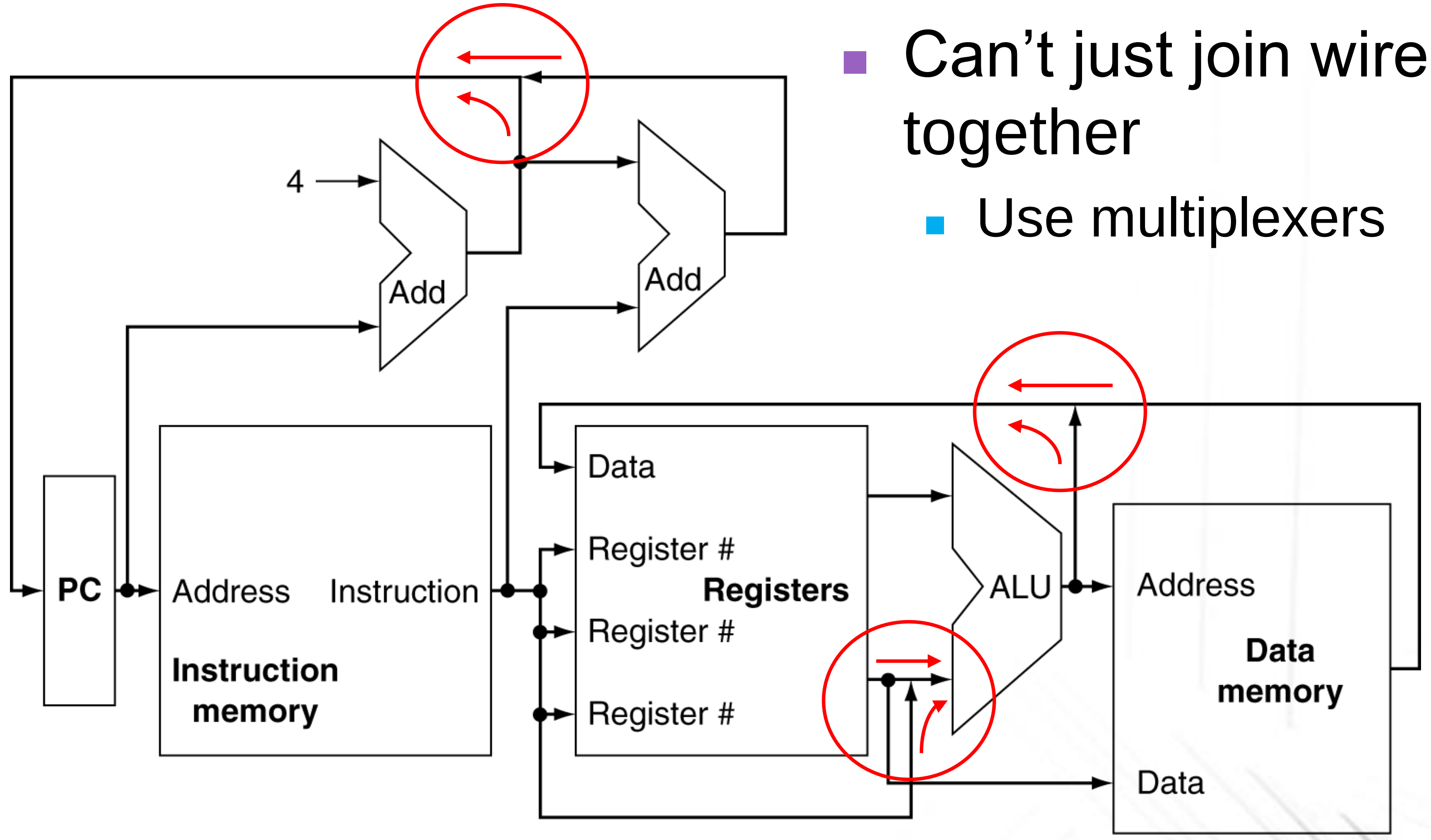


- Memory operations first need to compute the effective address
  - SW $t1, 450($s3)   # E.A. = 450 + [$s3]
  - Add together one register and 16 bits of immediate data
  - Immediate data needs to be converted from 16-bit to 32-bit
- Memory then performs load or store using destination register

Memory Operations (Load n Store)

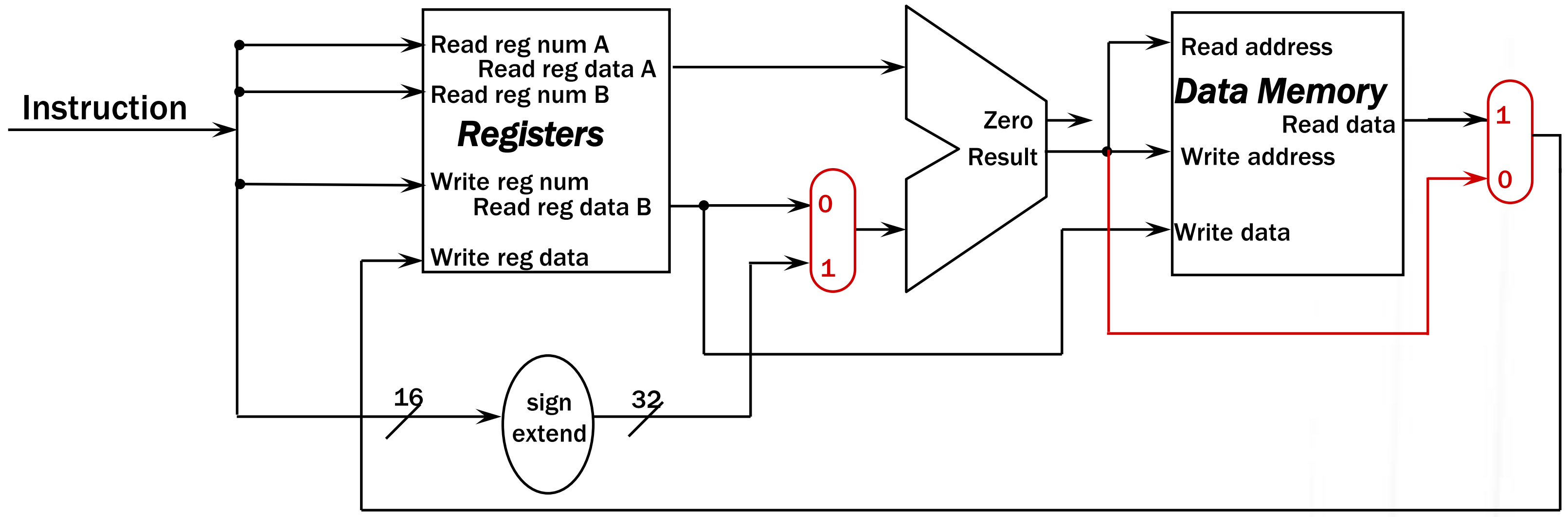


Towards Integrating Datapaths



- Can’t just join wires together
  - Use multiplexers

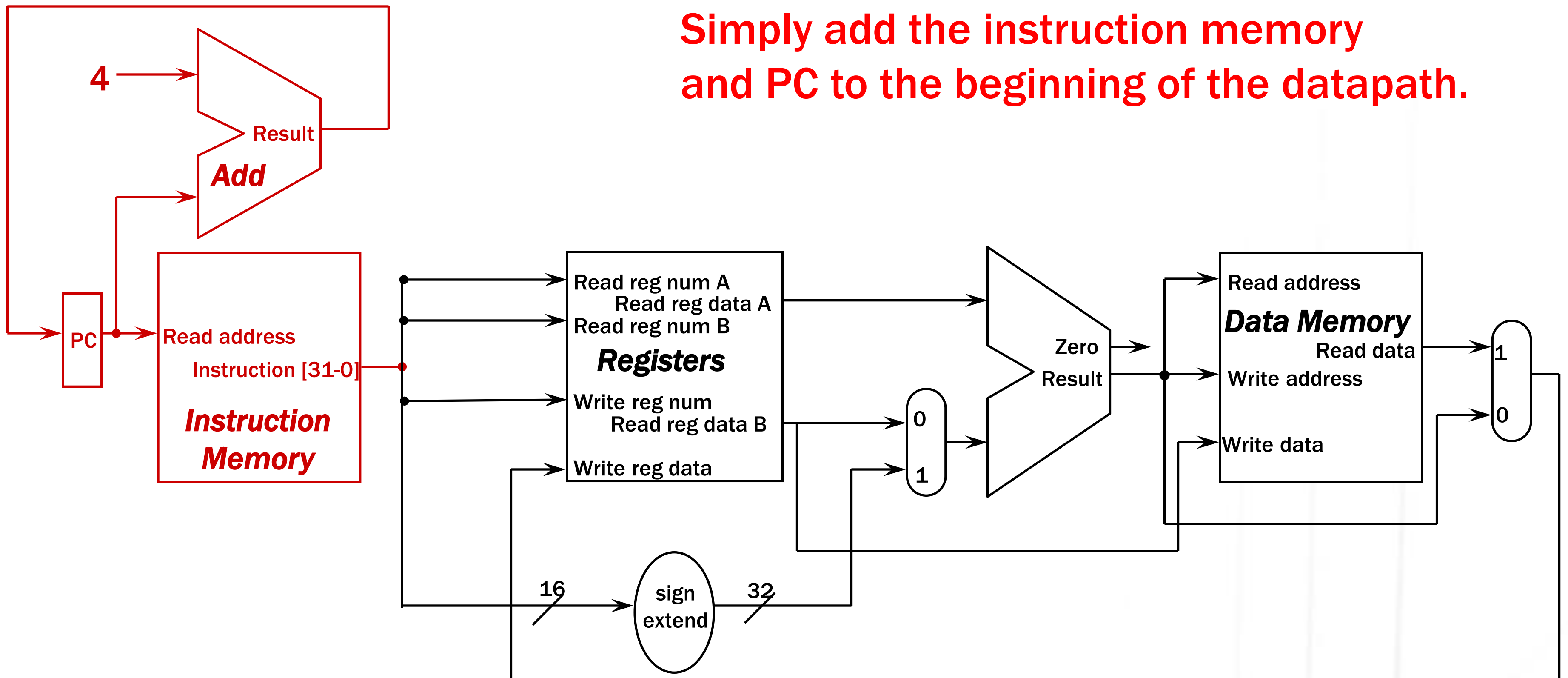
Integrating the R-types and Memory



- R-types and Load/Stores are similar in many respects
- Differences:
  - 2nd ALU source: R-types use register, I-types use Immediate
  - Write Data: R-types use ALU result, I-types use memory
- Mux the conflicting datapaths together
  - Put on the control logic for now

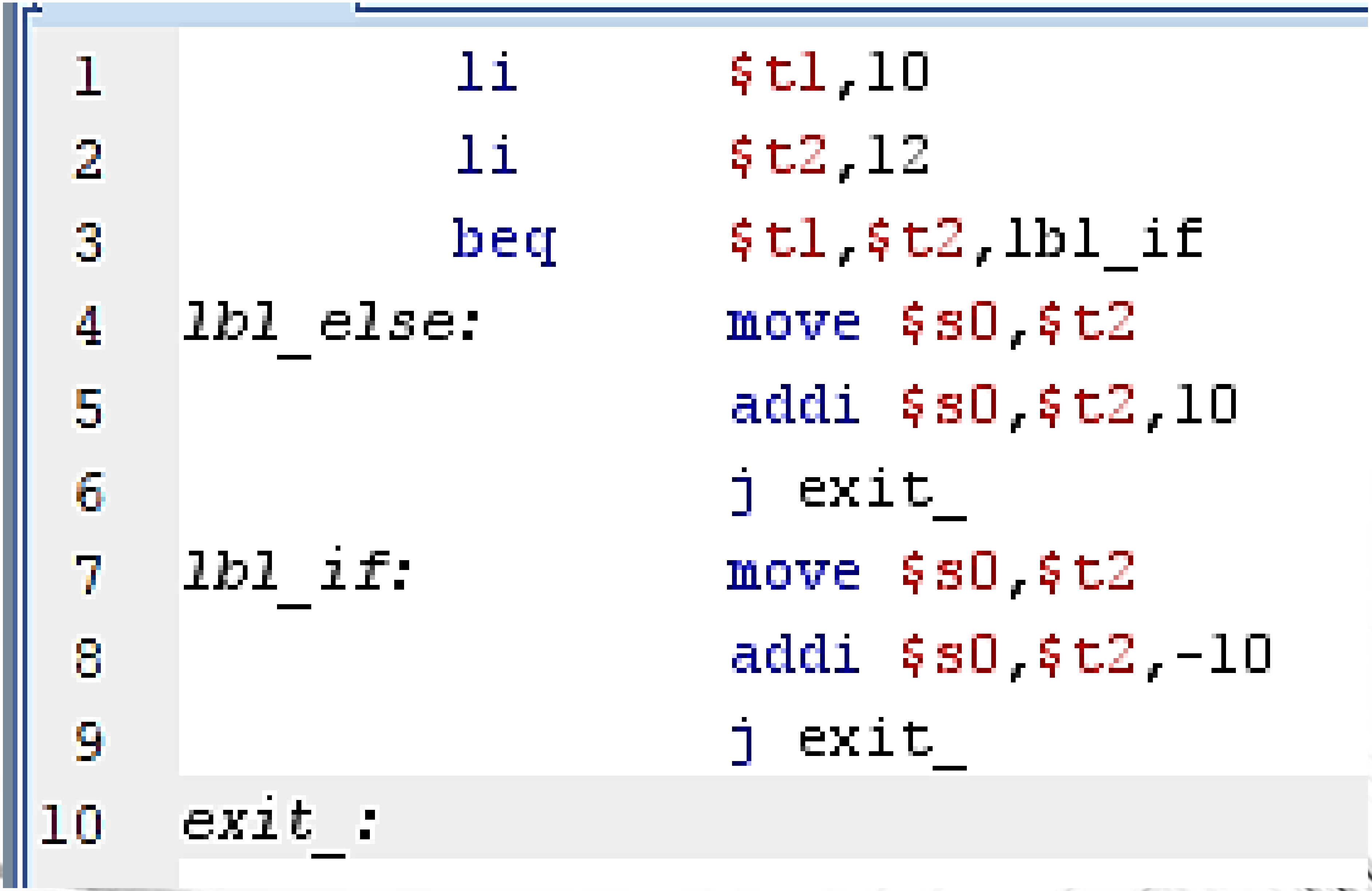
Adding the instruction memory

Simply add the instruction memory and PC to the beginning of the datapath.



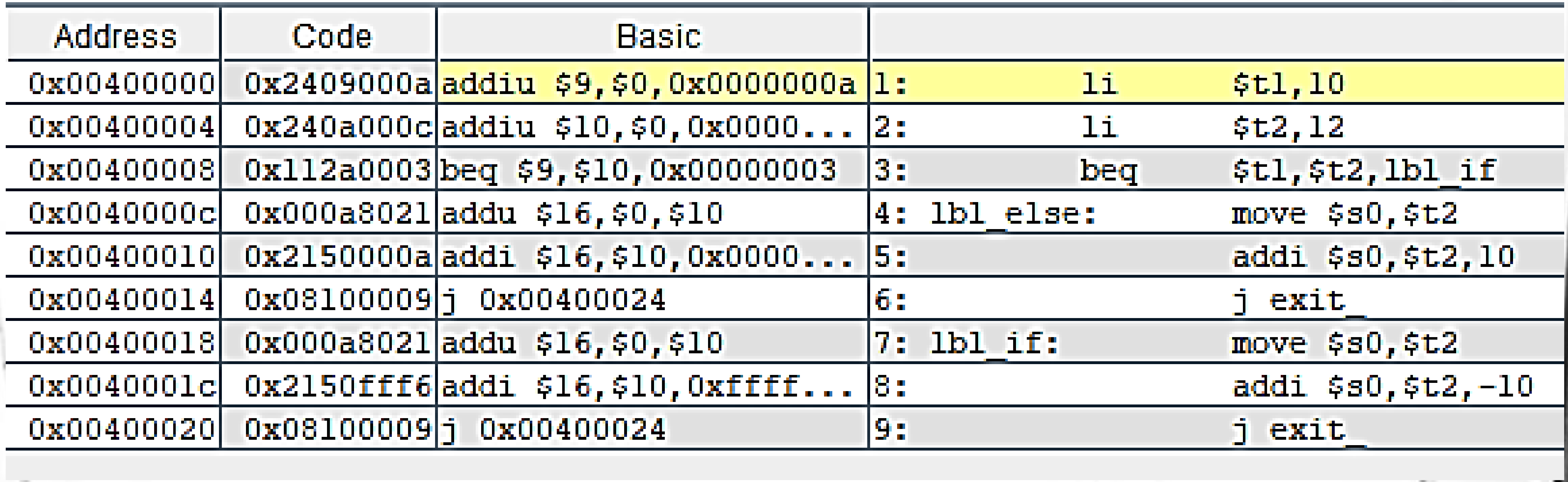
Separate Instruction and Data memories are needed in order to allow the entire datapath to complete its job in a single clock cycle.

Branch



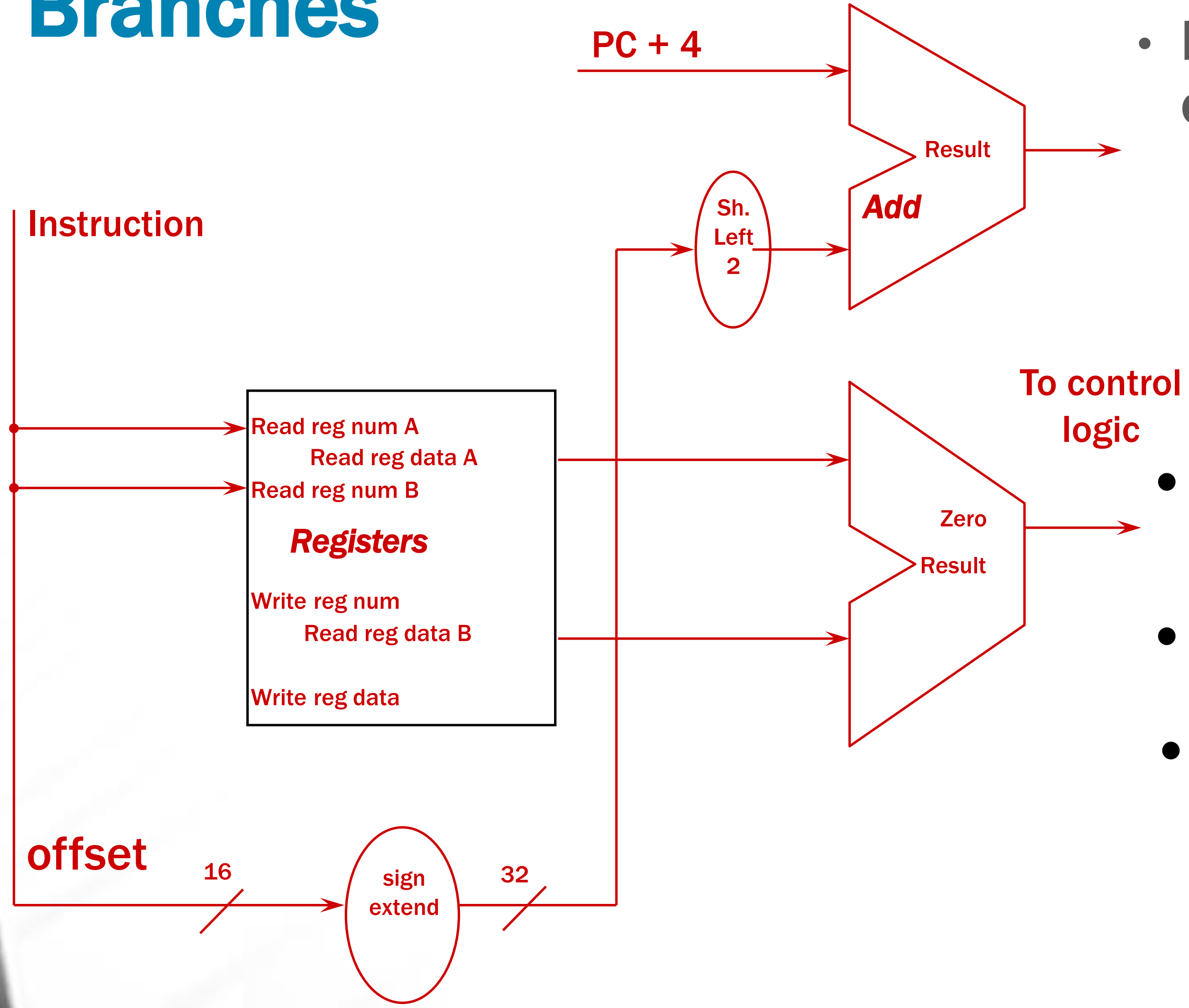
```
            li      $t1,10
            li      $t2,12
            beq     $t1,$t2,lbl_if
lbl_else:           move $s0,$t2
                    addi $s0,$t2,10
                    j exit_
lbl_if:             move $s0,$t2
                    addi $s0,$t2,-10
                    j exit_
exit_:
```

Branch



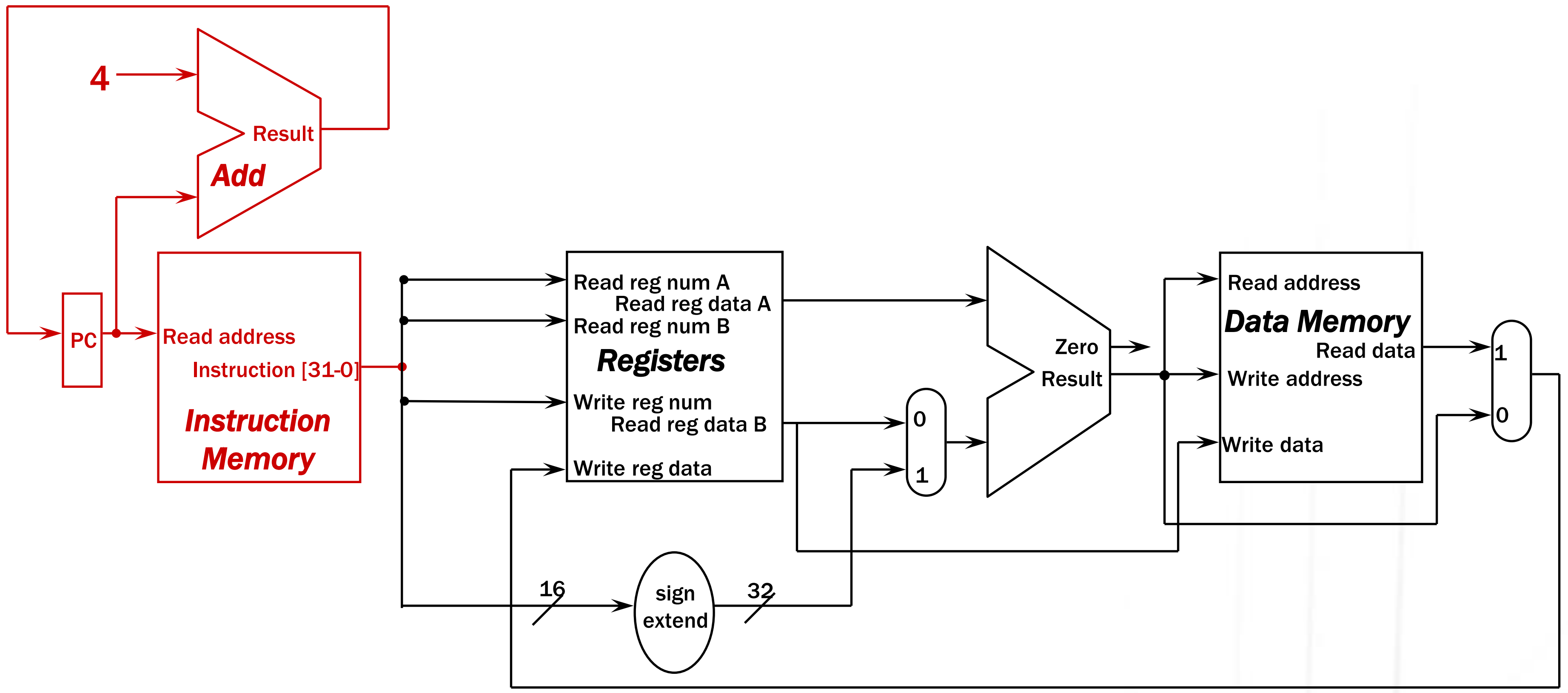
Address	Code	Basic	
0x00400000	0x2409000a	addiu $9,$0,0x0000000a	1: li $t1,10
0x00400004	0x240a000c	addiu $10,$0,0x0000...	2: li $t2,12
0x00400008	0x112a0003	beq $9,$10,0x00000003	3: beq $t1,$t2,lbl_if
0x0040000c	0x000a8021	addu $16,$0,$10	4: lbl_else: move $s0,$t2
0x00400010	0x2150000a	addi $16,$10,0x0000...	5: addi $s0,$t2,10
0x00400014	0x08100009	j 0x00400024	6: j exit_
0x00400018	0x000a8021	addu $16,$0,$10	7: lbl_if: move $s0,$t2
0x0040001c	0x2150fff6	addi $16,$10,0xffff...	8: addi $s0,$t2,-10
0x00400020	0x08100009	j 0x00400024	9: j exit_

Branches

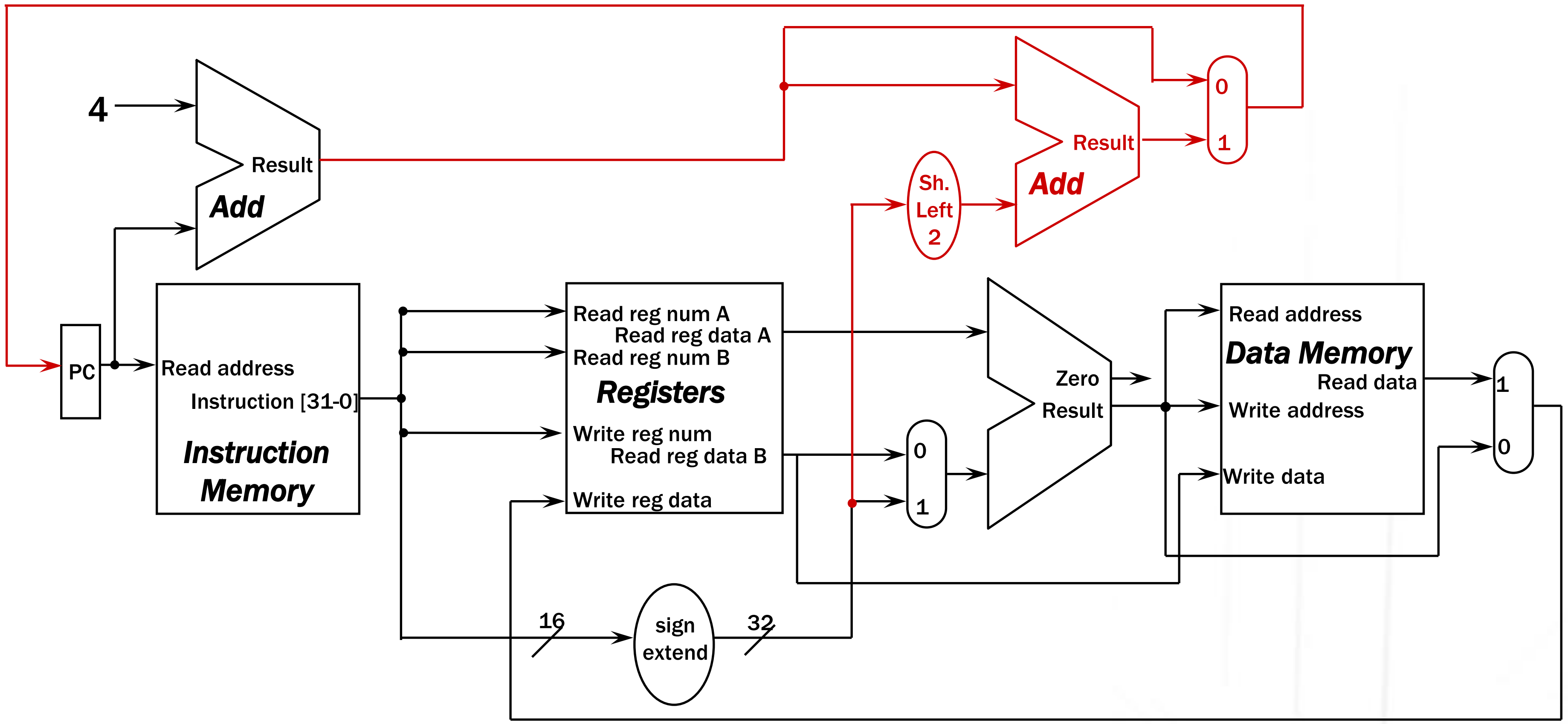


- Branches conditionally change the next instruction
  - BEQ $2, $1, 42
  - The offset is specified as the number of words to be added to the next instruction (PC+4)
- Take offset, multiply by 4
  - Shift left two
- Add this to PC+4 (from PC logic)
- Control logic has to decide if the branch is taken
  - Uses ‘zero’ output of ALU

Datapath without Branch Instruction

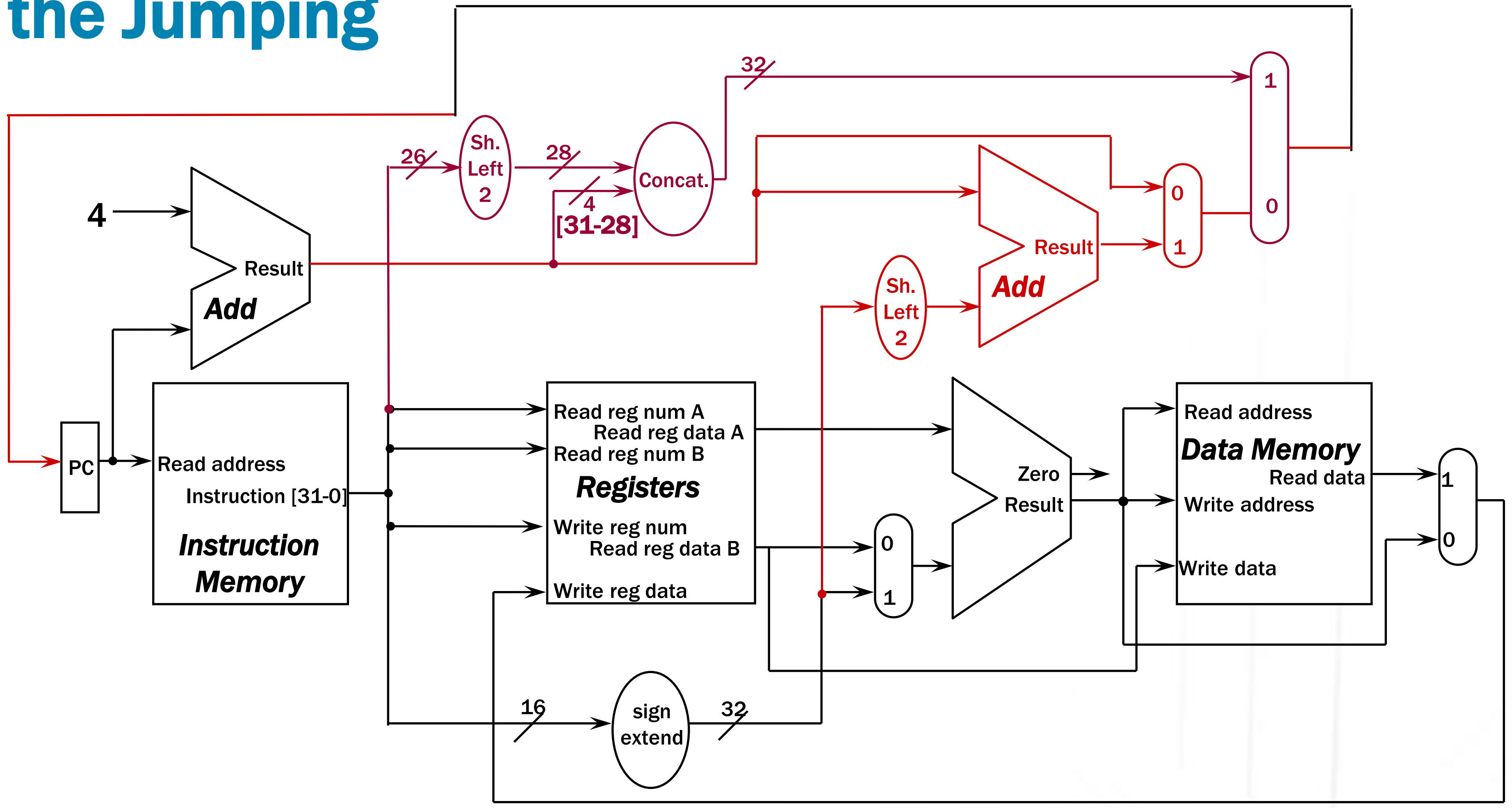


Adding the Branch Datapath



Now we have the datapath for R-type, I-type, and branch instructions.

Adding the Jumping

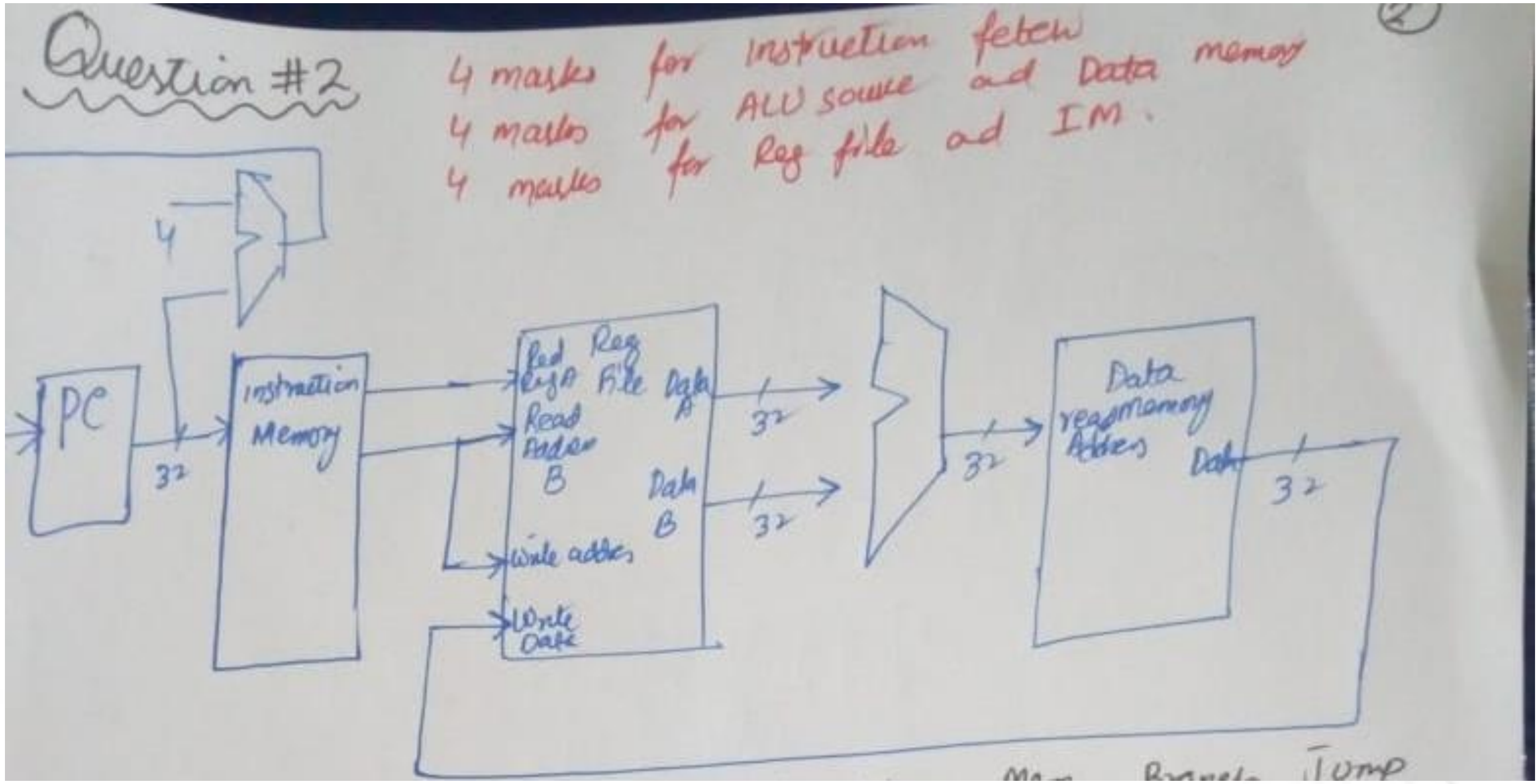


Now we have the datapath for R-type, I-type, branch (conditional branch) and jump (unconditional branch) instructions.

On to the control logic!

Previous Evaluation from Datapath Design

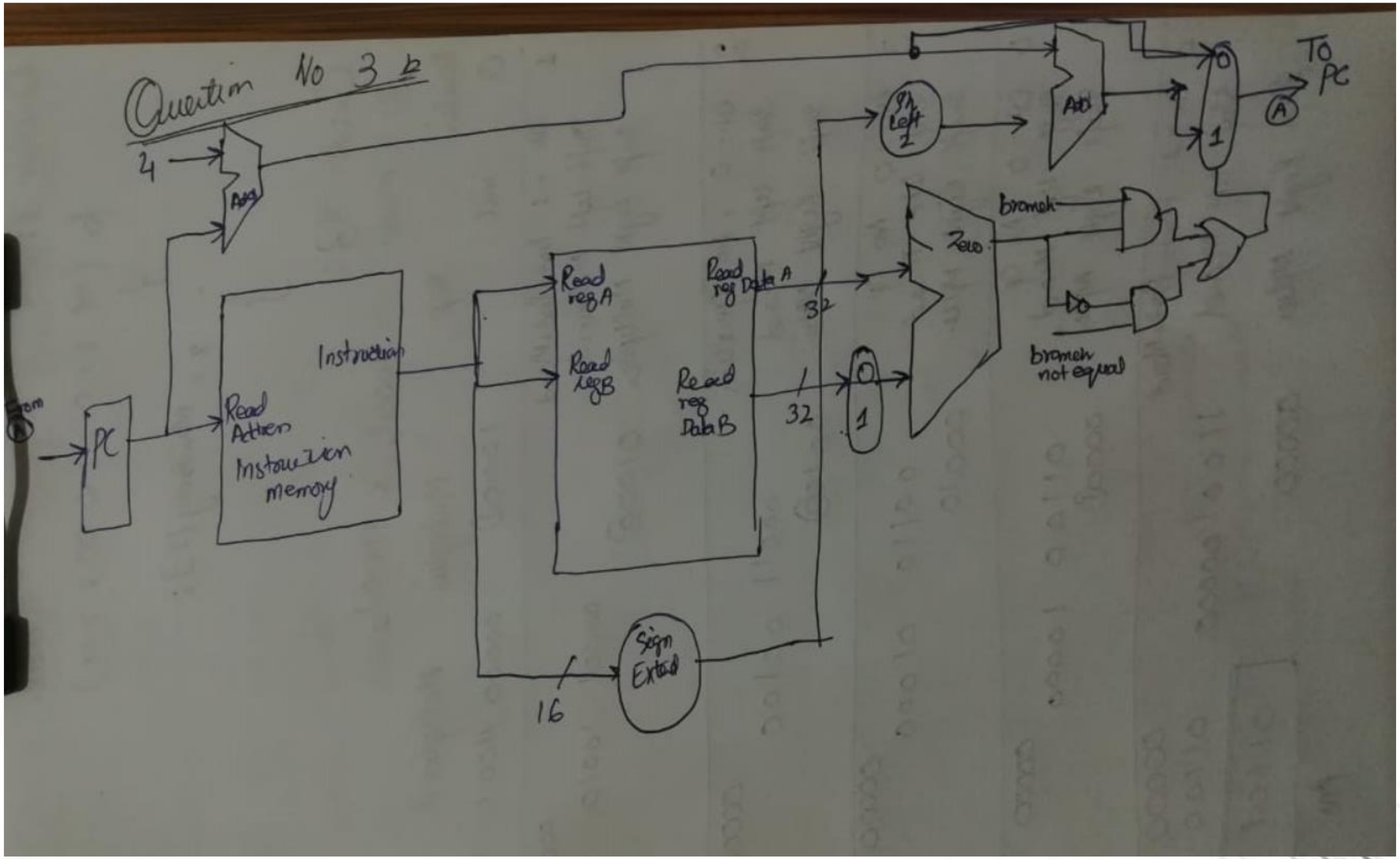
- ***Reproduce*** a 32bit Single cycle MIPS Datapath that can support a new instruction called **FRR**. FRR is very similar to our basic LW instruction, but instead of using an offset in an immediate, we store it in a register.
- ***Reg[$rt] = Mem[$rs + $rt]***
- This means we need to add two registers together and use the result to address memory for a read. As a design Engineer your task is to decide the components and control signals needed to get this to work.
- You are required to:
  - Make a clean Datapath for **FRR** instruction only, using Datapath components
  - Fill the control signal table given below
  - Decide which instruction type (R, I, J) can support this instruction, draw its template indicating number of bits for each field


Question #2

4 marks for instruction fetch
4 marks for ALU source and Data memory
4 marks for Reg file and IM.

Previous Evaluation from Datapath Design

- For the MIPS assembly instructions **BEQ & BNE**:
    - ***Construct*** a single cycle Datapath that executes these two instructions only.
    - Describe how these instructions would be executed in the Datapath you've drawn.
    - Write down in tabular form showing the values of ALL control signals needed for the implementation of these instructions.
    - You may use multiple instances of a component and can increase/decrease the dimensions of the below mentioned components. Not all components are mandatory to use.


Question No 3 b
4
Add
From
A
PC
Read Address
Instruction memory
Instruction
Read reg A
Read reg B
Read reg Data A
Read reg Data B
32
32
16
Sign Extend
0
1
Zero
branch
branch not equal
Sl Left 2
Add
0
1
A
To PC

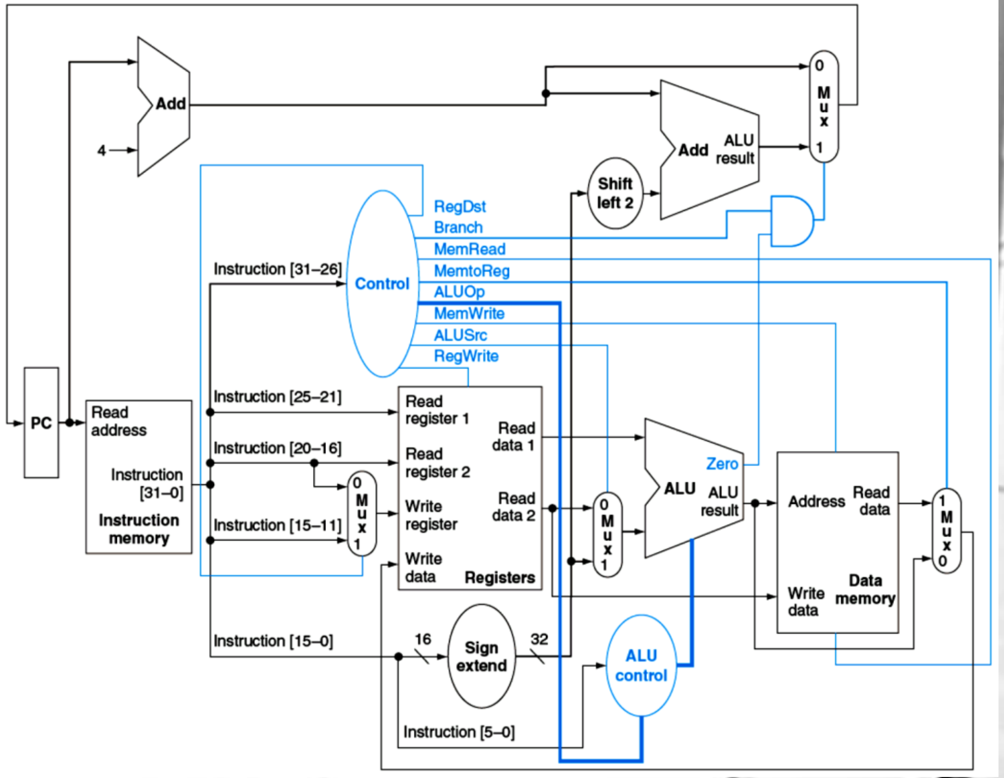
Previous Evaluation from Datapath Design

- The simple data path with control for MIPS is given in the following diagram:
- ***Extend*** the given 32-bit single cycle MIPS Datapath to support the instruction "*jr* jump register" instruction. The instruction format is given below:

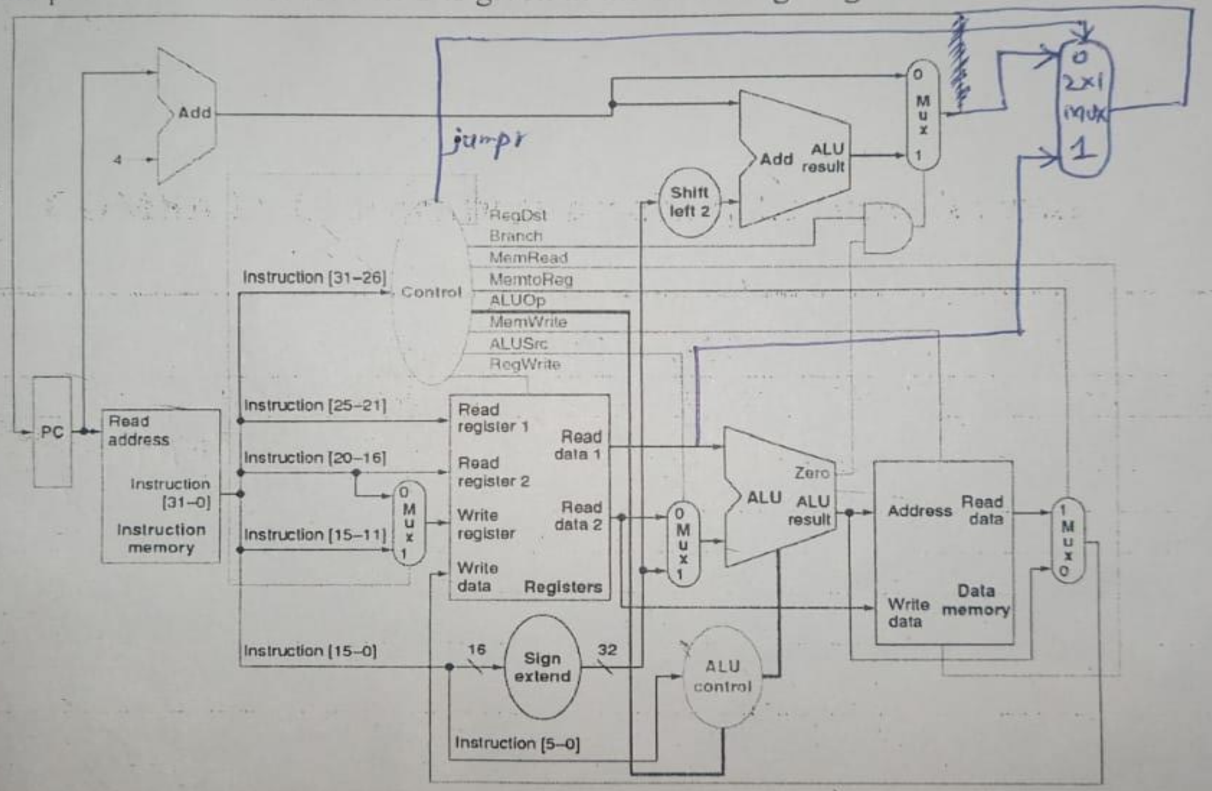
Jump Register	jr	R	PC=R[rs]	0 / 08_{hex}

- Fill the control signal table given below:

Instruction	Opcode	Reg Write	ALU Src	Mem To Reg	Reg Dest	Mem Read	Mem Write	Branch	Jump
JR									



The simple data path with control for MIPS is given in the following diagram:



a) ***Extend*** the given 32-bit single cycle MIPS Datapath to support the instruction "*jr* jump register" instruction. The instruction format is given below:

Jump Register jr R PC=R[rs] 0 / 08$_{hex}$

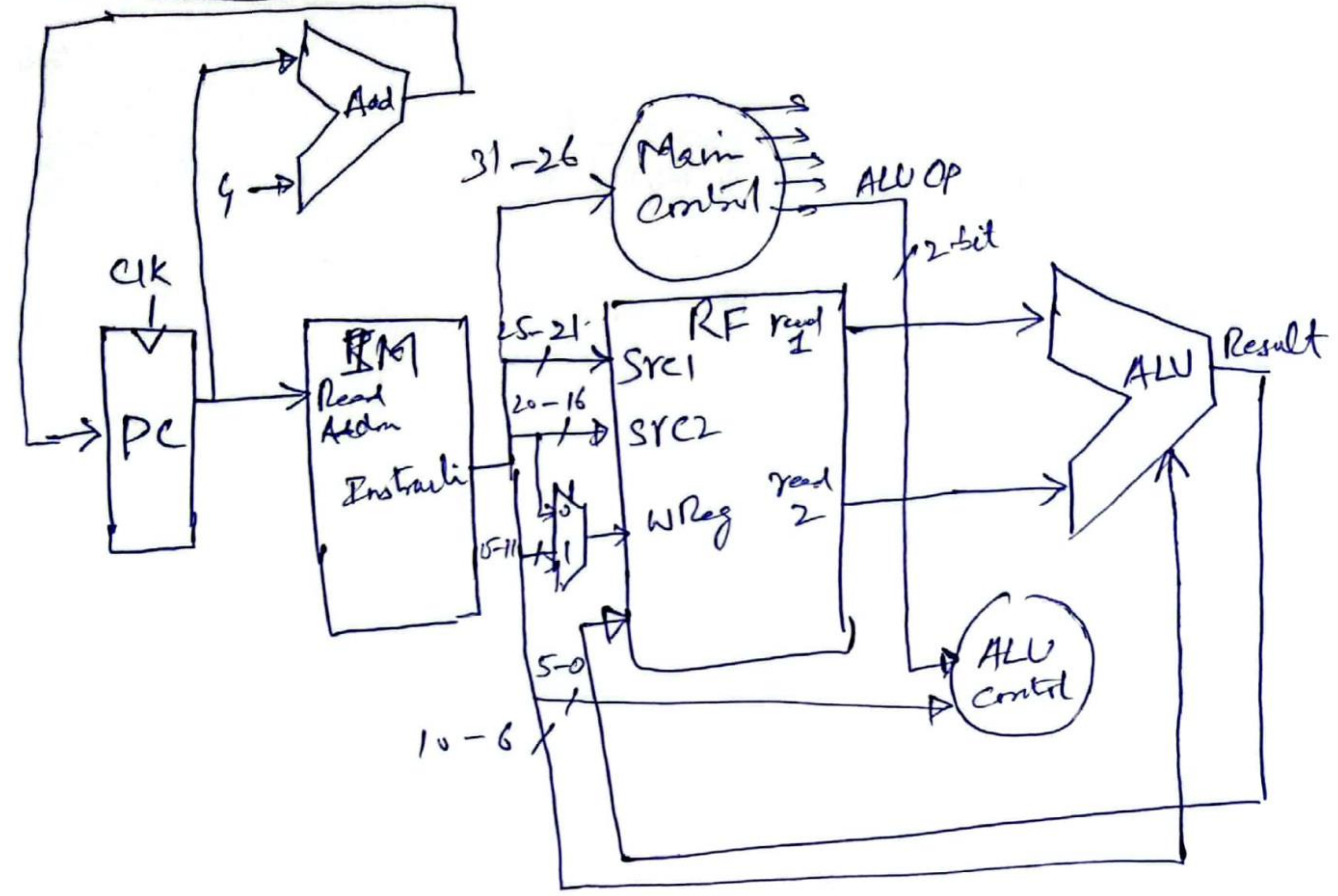
b) Fill the control signal table given below:

Instruction	Opcode	Reg Write	ALU Src	Mem To Reg	Reg Dest	Mem Read	Mem Write	Branch	Jumpr
JR	000000	X	X	X	X	X	X	X	1

Previous Evaluation from Datapath Design

- For the MIPS assembly instructions SLL & SRL, *construct* a single cycle datapath from the datapath components mentioned below that executes these two instructions only. You may use multiple instances of a component and can increase/decrease the dimensions of these components. Not all components are mandatory to use.

Question #3:-



Add
4
Clk
PC
IM
Read Addr
Instruction
31-26
Main Control
ALU OP
2-bit
25-21
20-16
15-11
5-0
10-6
RF
Src1
Src2
WReg
read 1
read 2
0
1
ALU
Result
ALU Control

Previous Evaluation from Datapath Design

- **Construct** a single cycle data path that implements the MIPS I-type instruction. Modify it so that it implements a **StorePCValue** instruction so it may store the value of PC in the destination register of register file.

- ***Construct*** a single cycle datapath from the datapath components mentioned below that executes a new I-type MIPS instruction getpc \$rt which sets register \$rt to the PC value. You may use multiple instances of a component and can increase/decrease the dimensions of these components. Not all components are mandatory to use.