CS M151B:
Computer Systems Architecture

Week 4

Retroactive Midterm Review

- Is a billion 10^9?
- Is GHz 10^9?

Retroactive Midterm Review

- Is a billion 10^9?
  - Yes
- Is GHz 10^9?
  - Yes

What's next?

- We have an adder.
- We can add things.
- Yay!
- Can we use the adder to do other things?

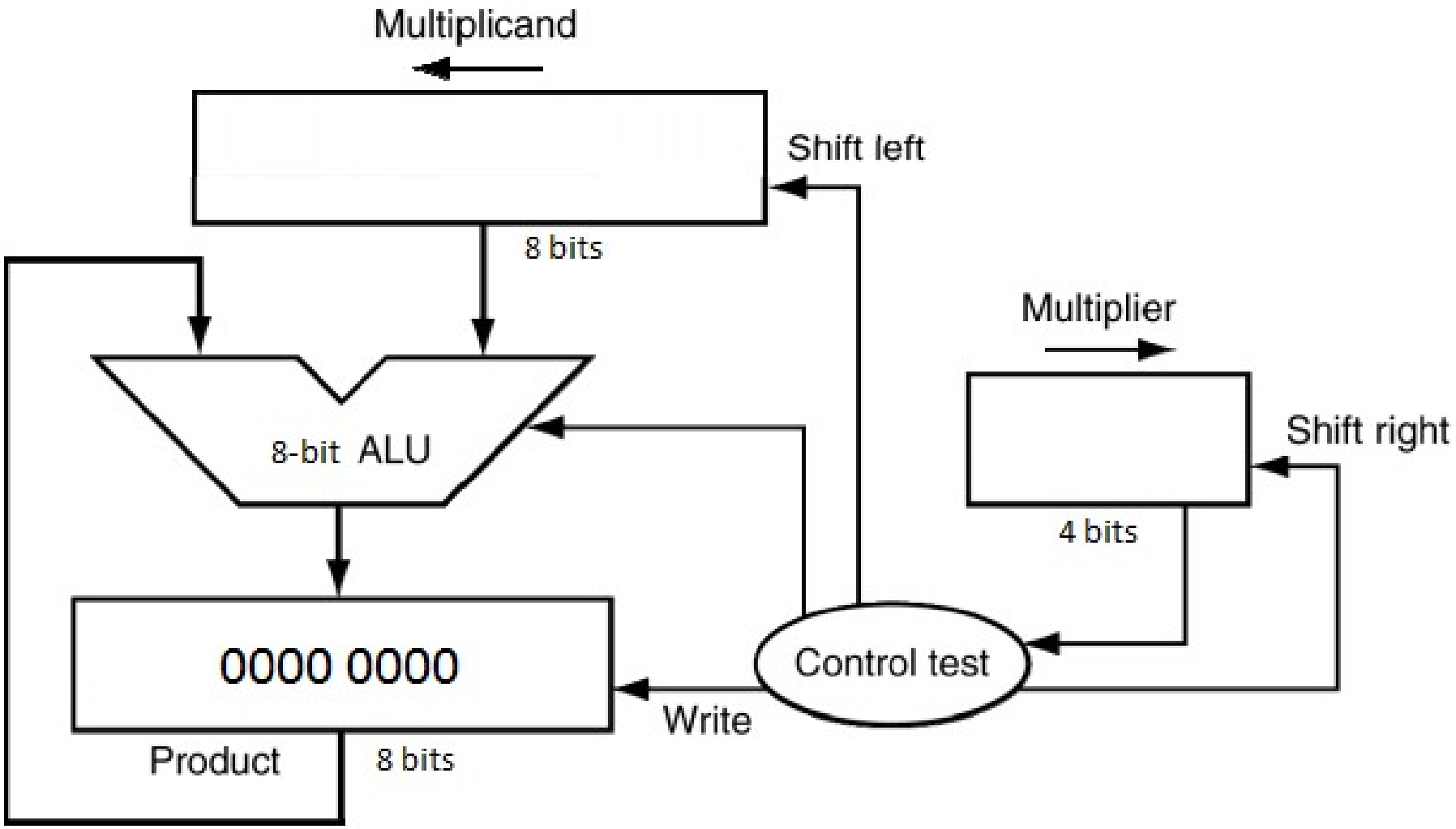
Multiplication

- Multiply via add:

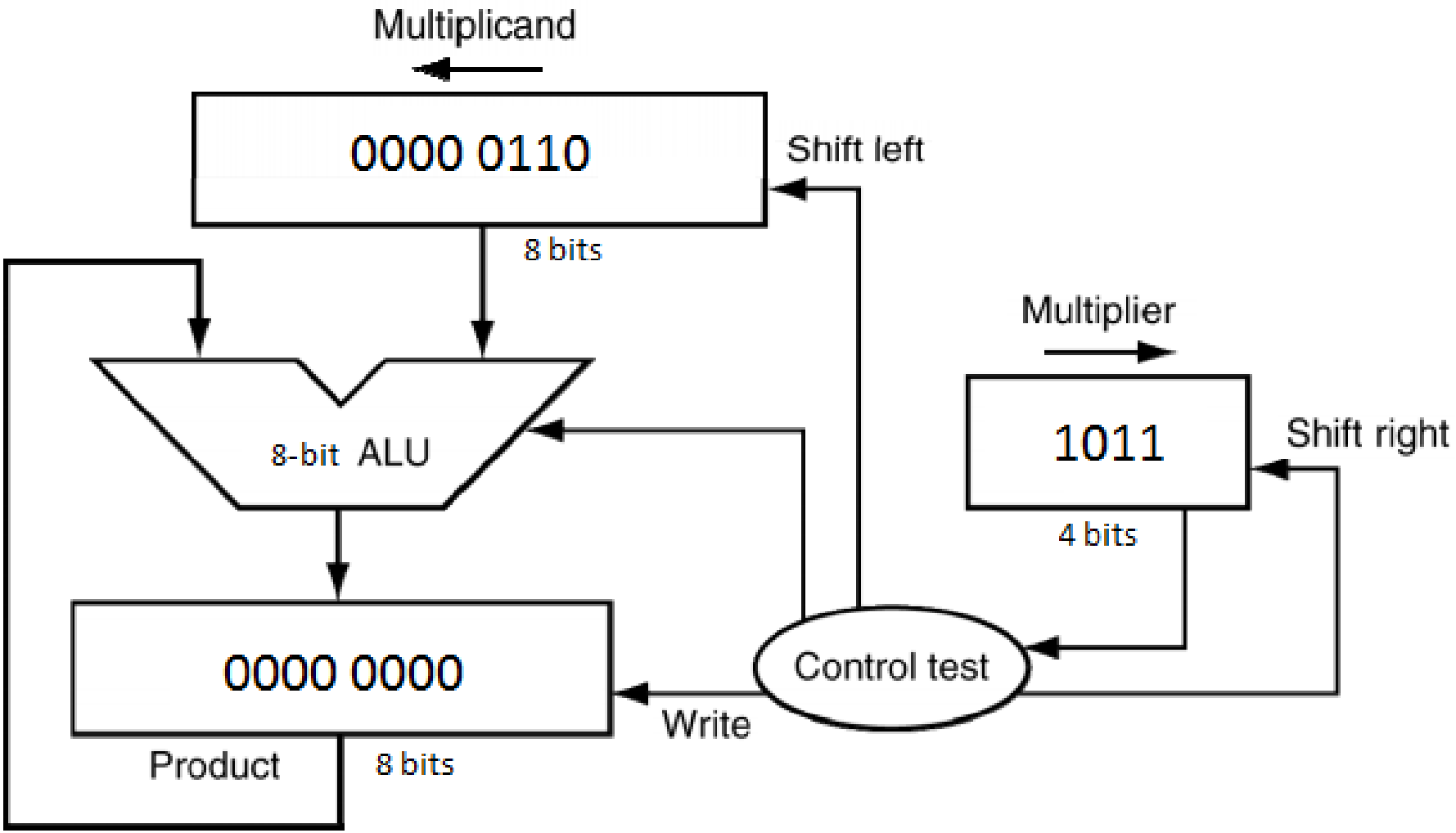
```
      0110 (6, multiplicand)
     x1011 (11, multiplier)
      ------
      0110
     0110
    0000
   0110
   -----------
   1000010 (66)
```

- For two n-bit numbers, we need to do n adds and the final value could be 2n-bits

First Multiplier

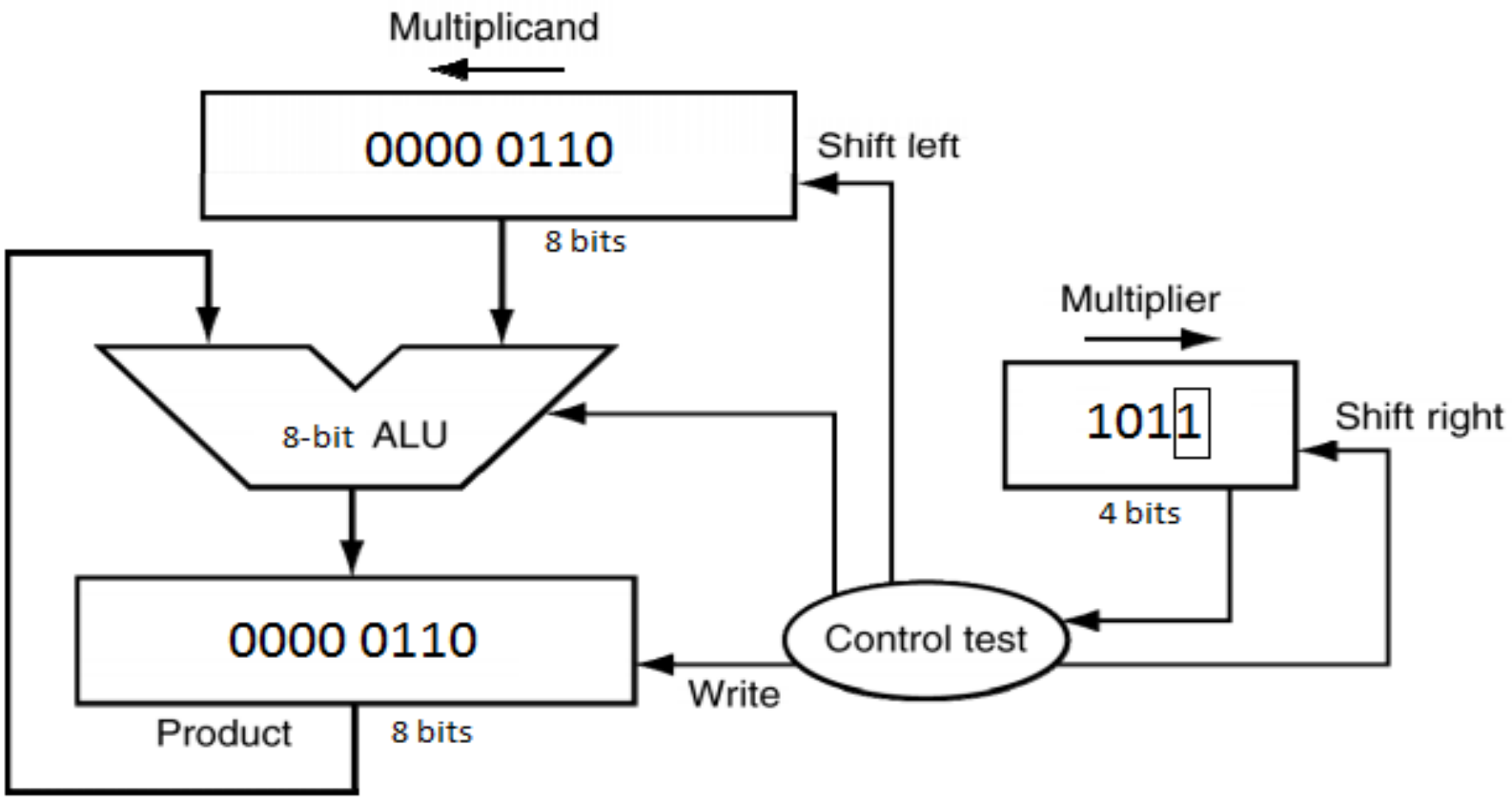


First Multiplier



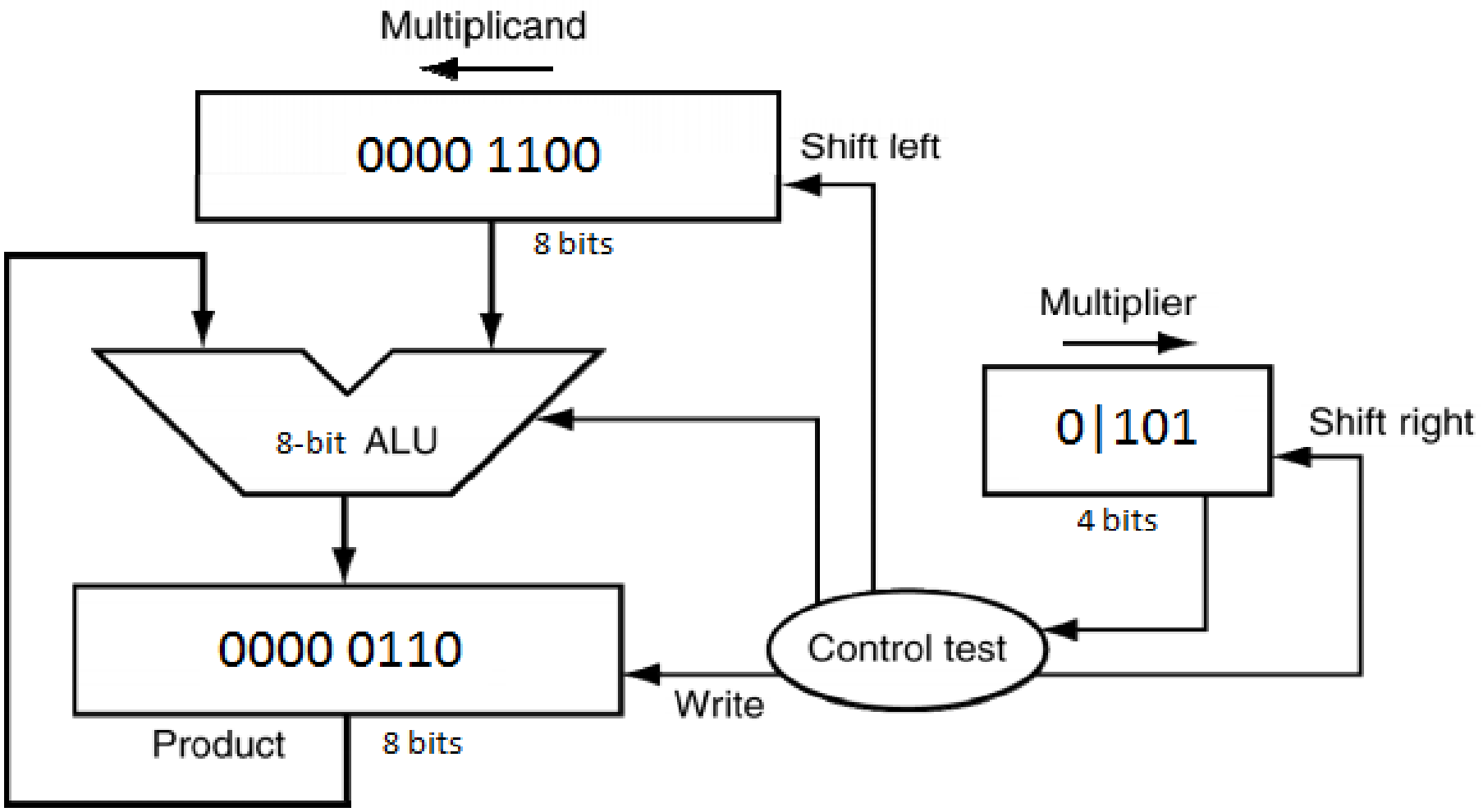
First Multiplier

- Check bit 0 of multiplier, add if 1



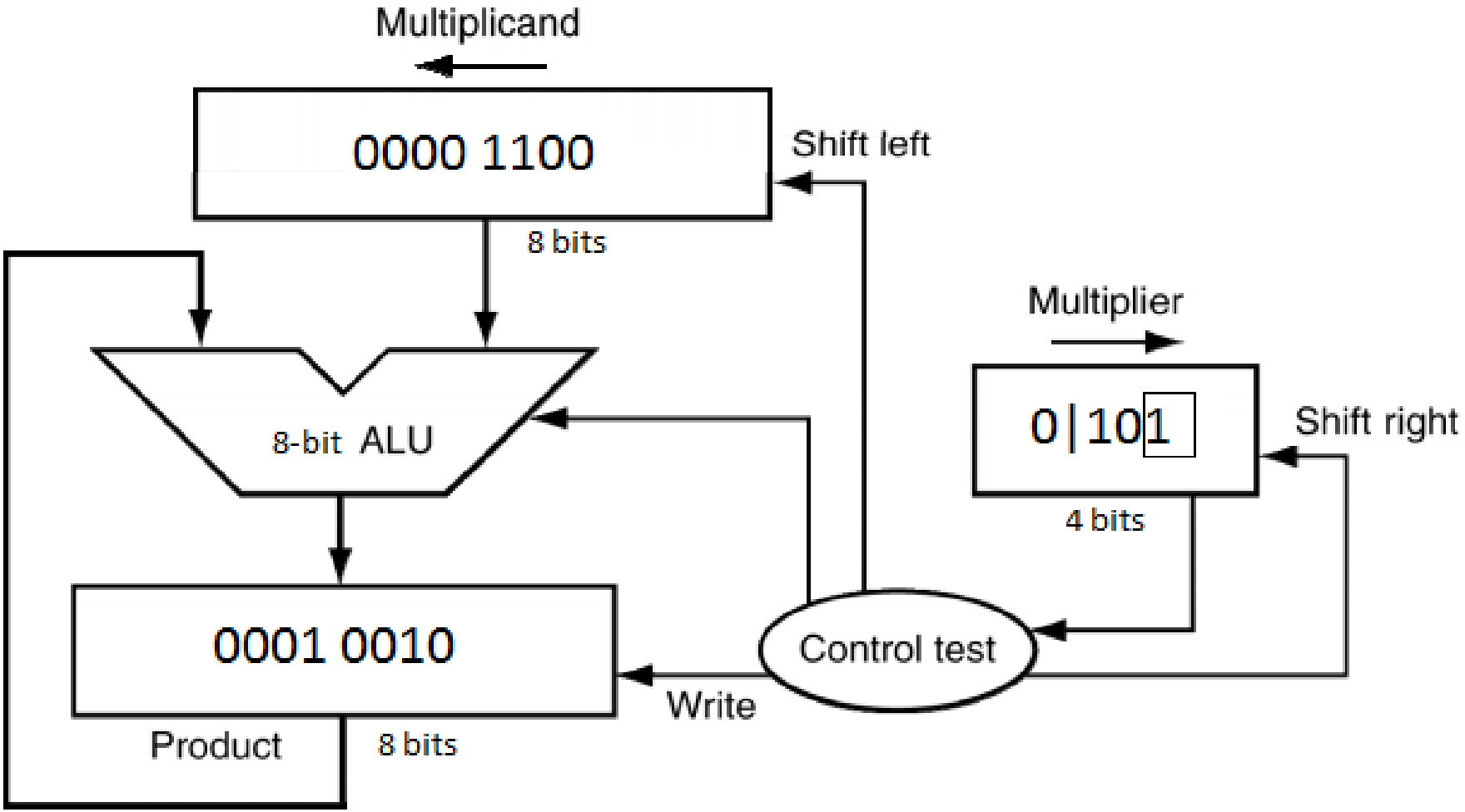
First Multiplier

- Shift



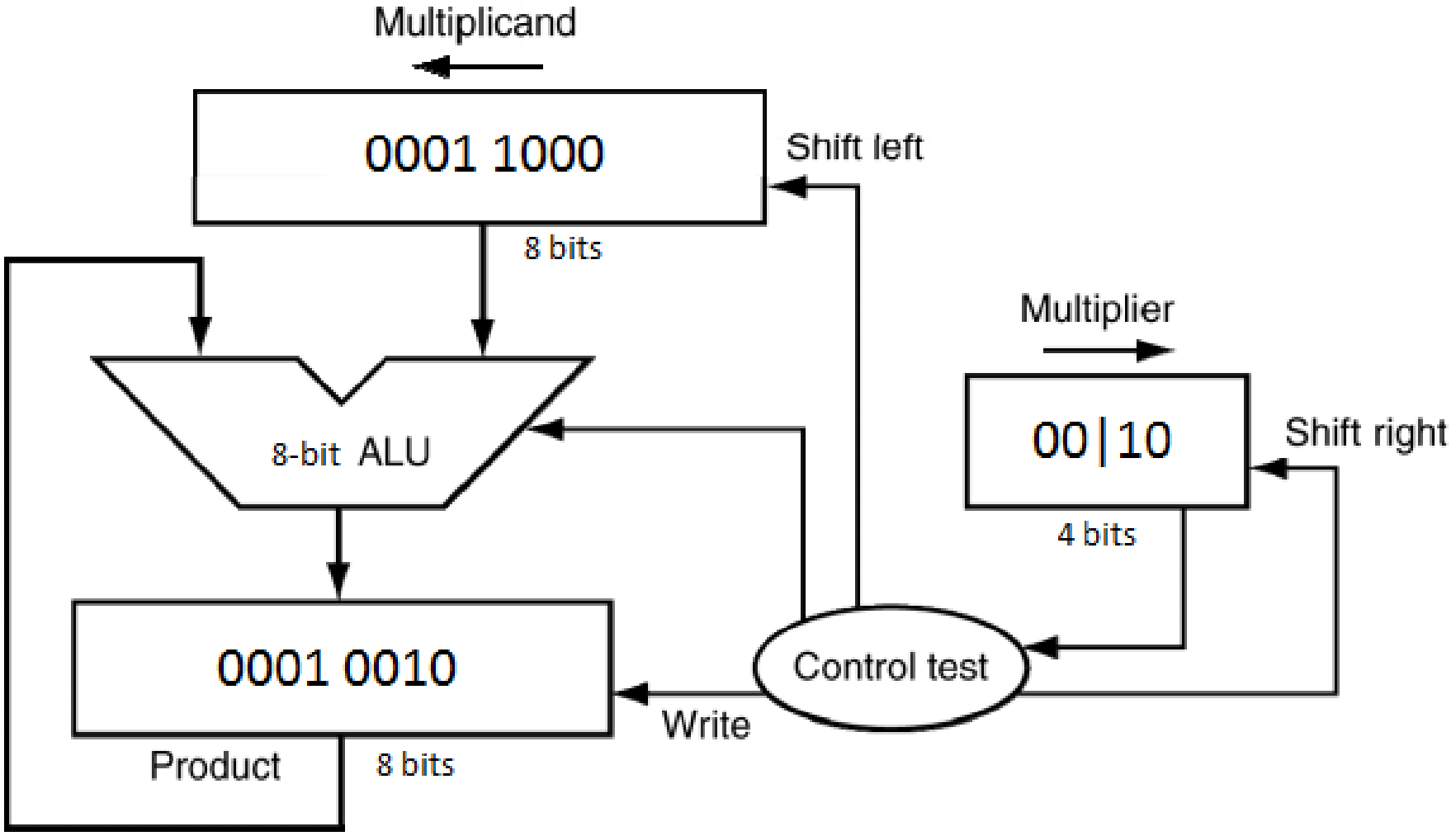
First Multiplier

- Check bit 0 of multipler, add if 1



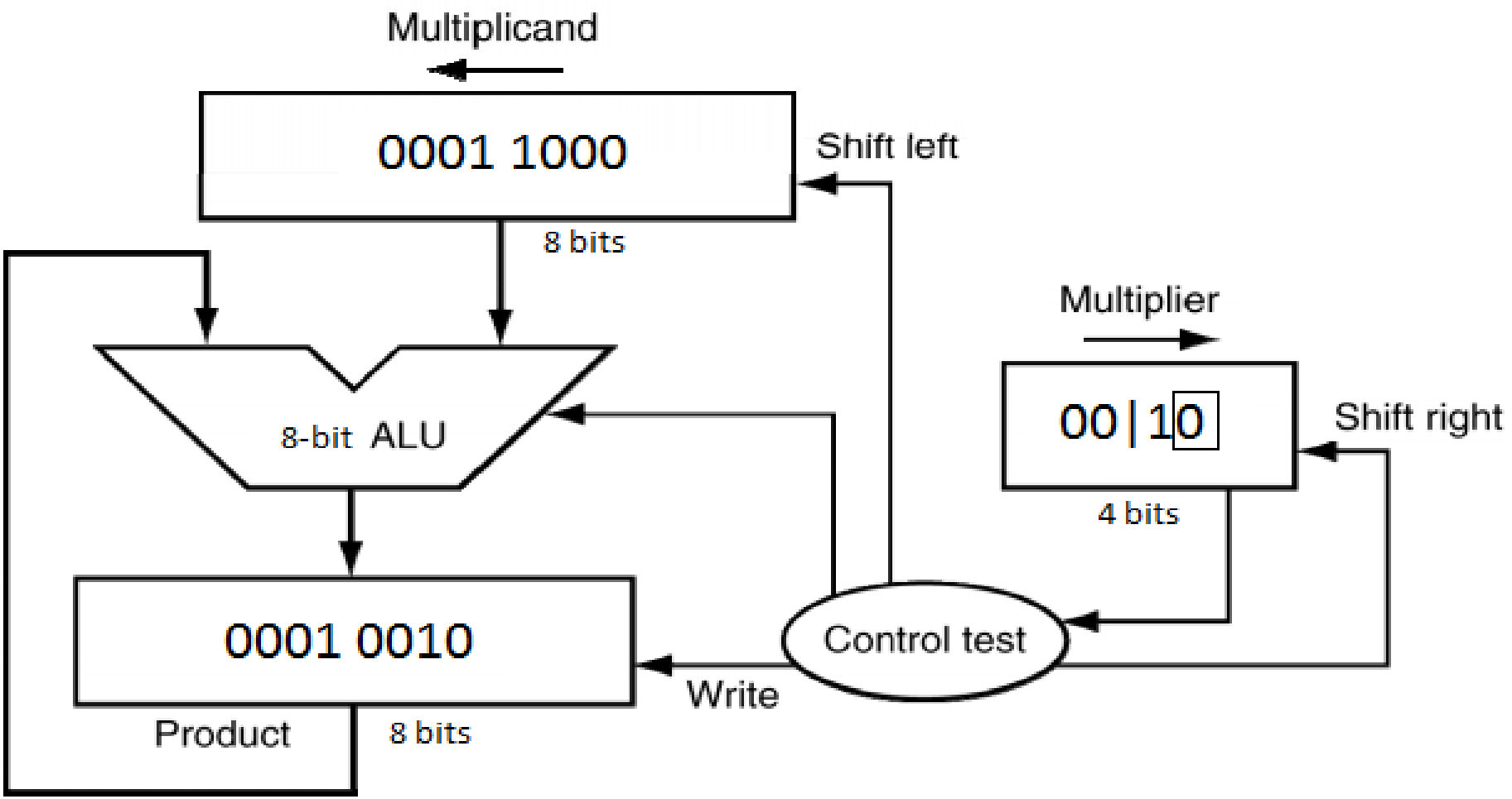
First Multiplier

- Shift



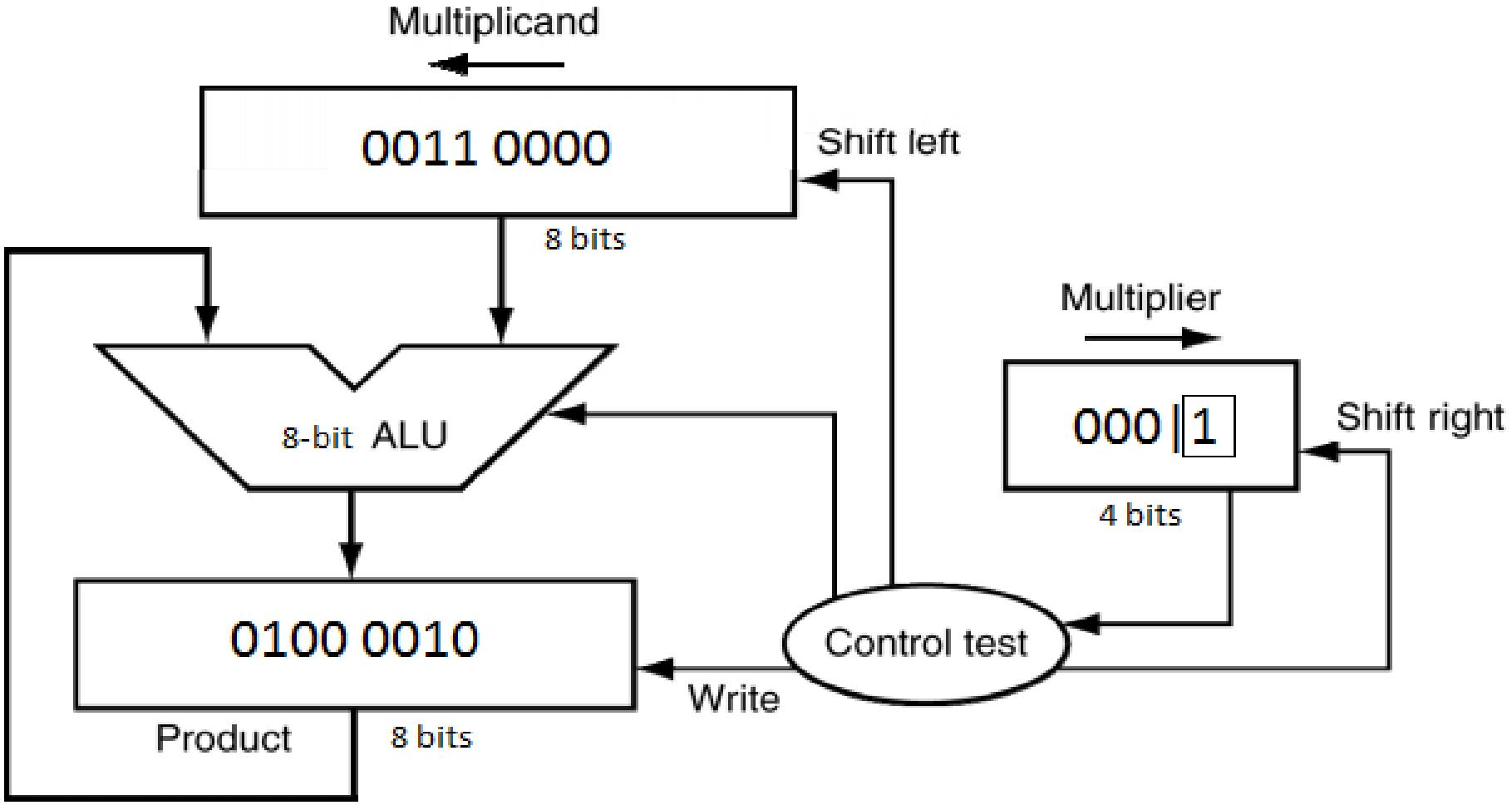
First Multiplier

- ...and so on



First Multiplier

- ...and so on



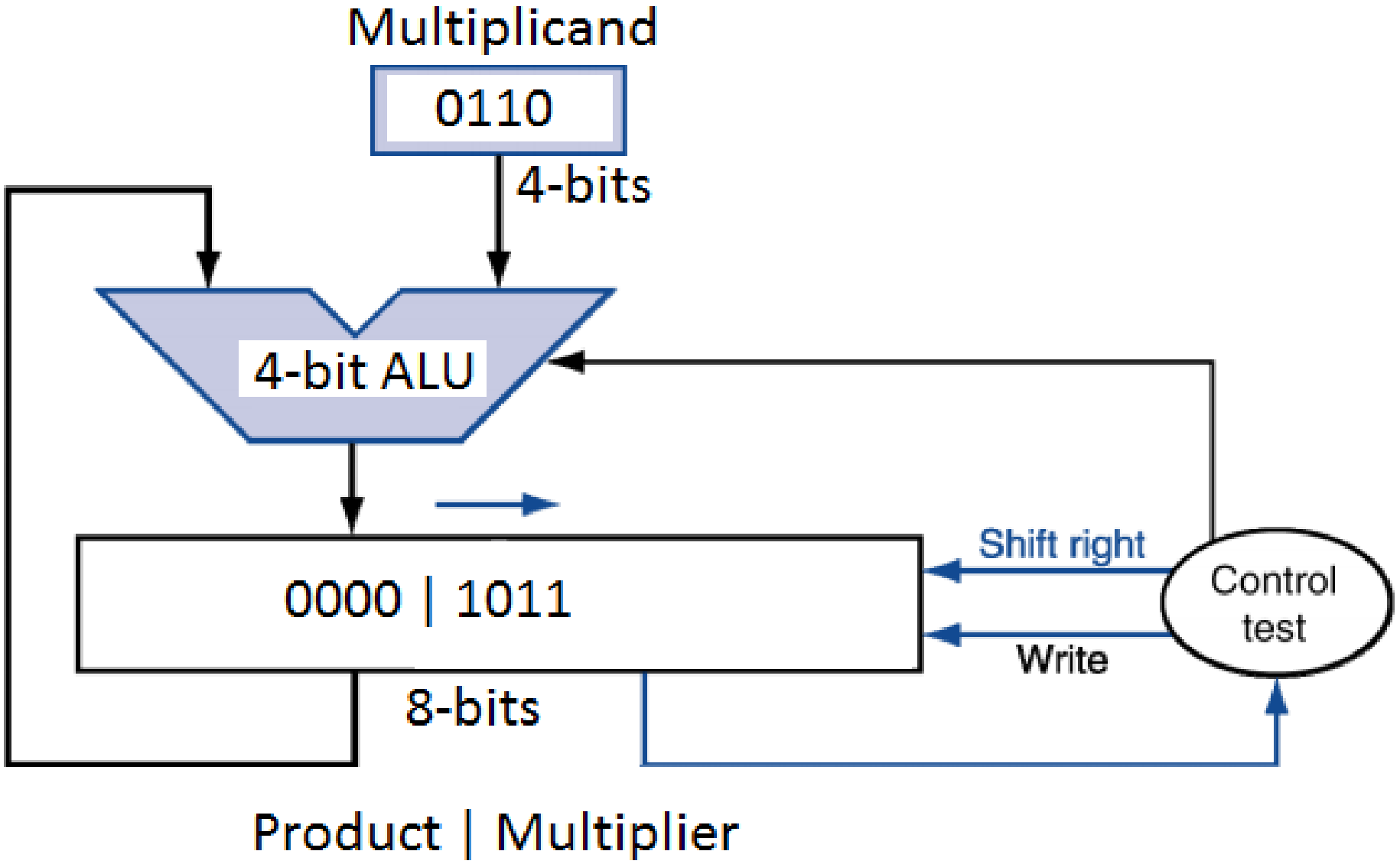
The Giga-dollar Question: Can we do better?

- Need to synchronize writes and shifts.
- Have one pass take a cycle.
- Doing a multiplication with 32-bit inputs takes 32 cycles!
- Can we improve the microarchitecture?

The Original Multiplier

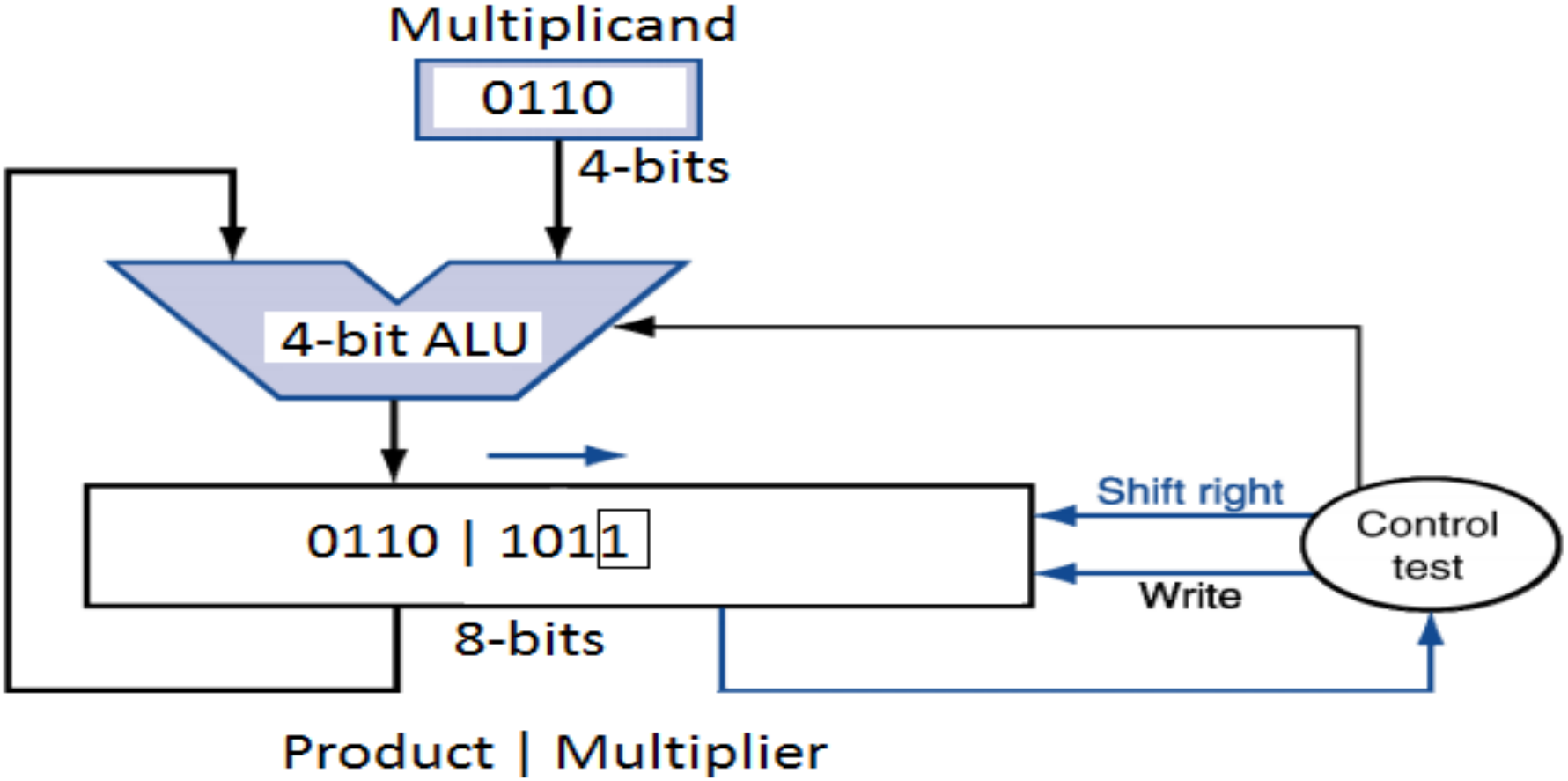
- For one thing, it seems we're wasting space/utilization
- The 2n-sized multiplicand register has at most n non-zero entries at a time.
- The n-size multiplier register shifts to zero.
- Maybe we can compress some things together?

The Refined Multiplier



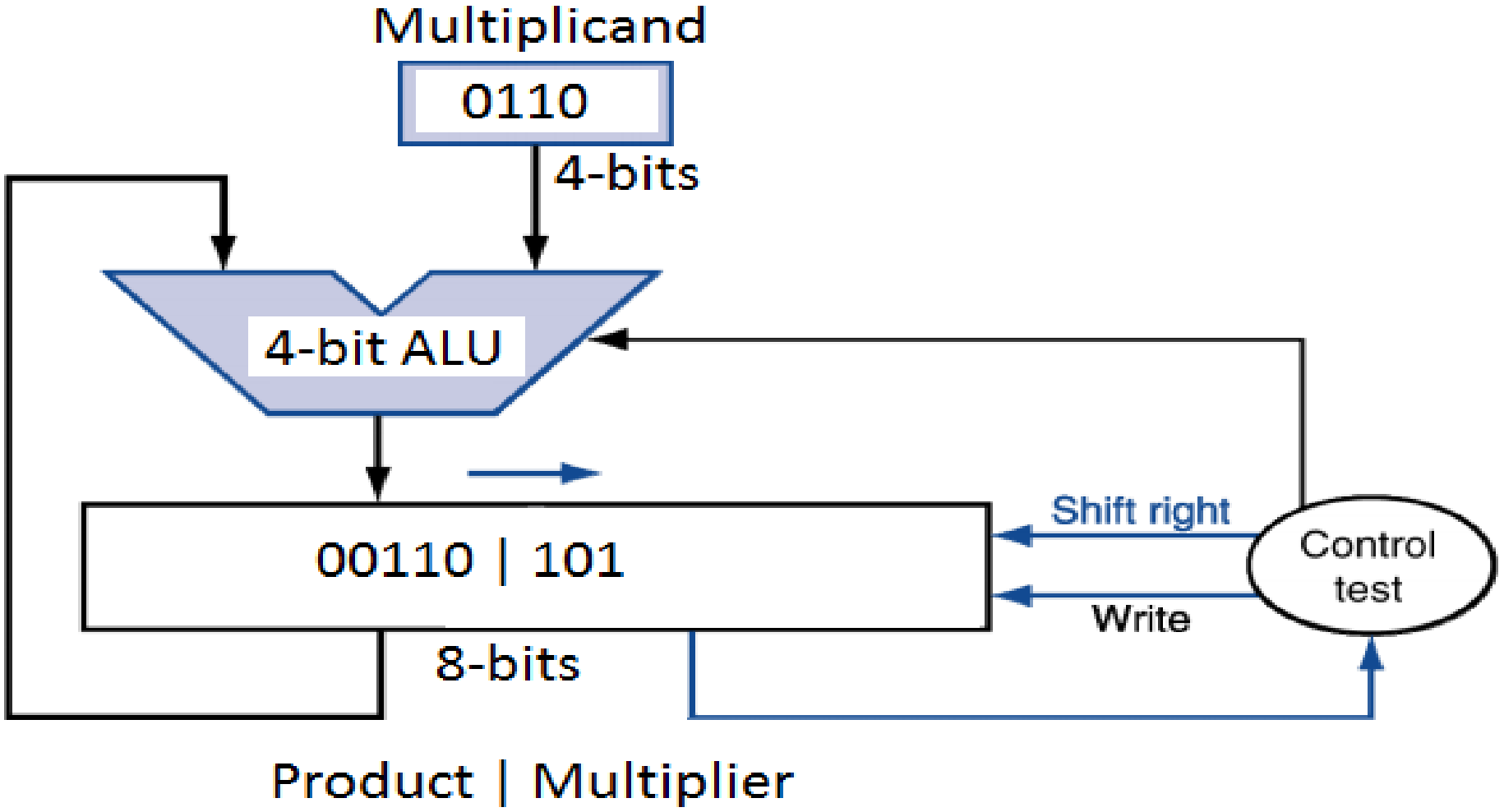
The Refined Multiplier

- Check bit 0 of product (multiplier), add if 1
- Note: You're adding the 4-bit multiplicand to the 4 most significant bits of the product.



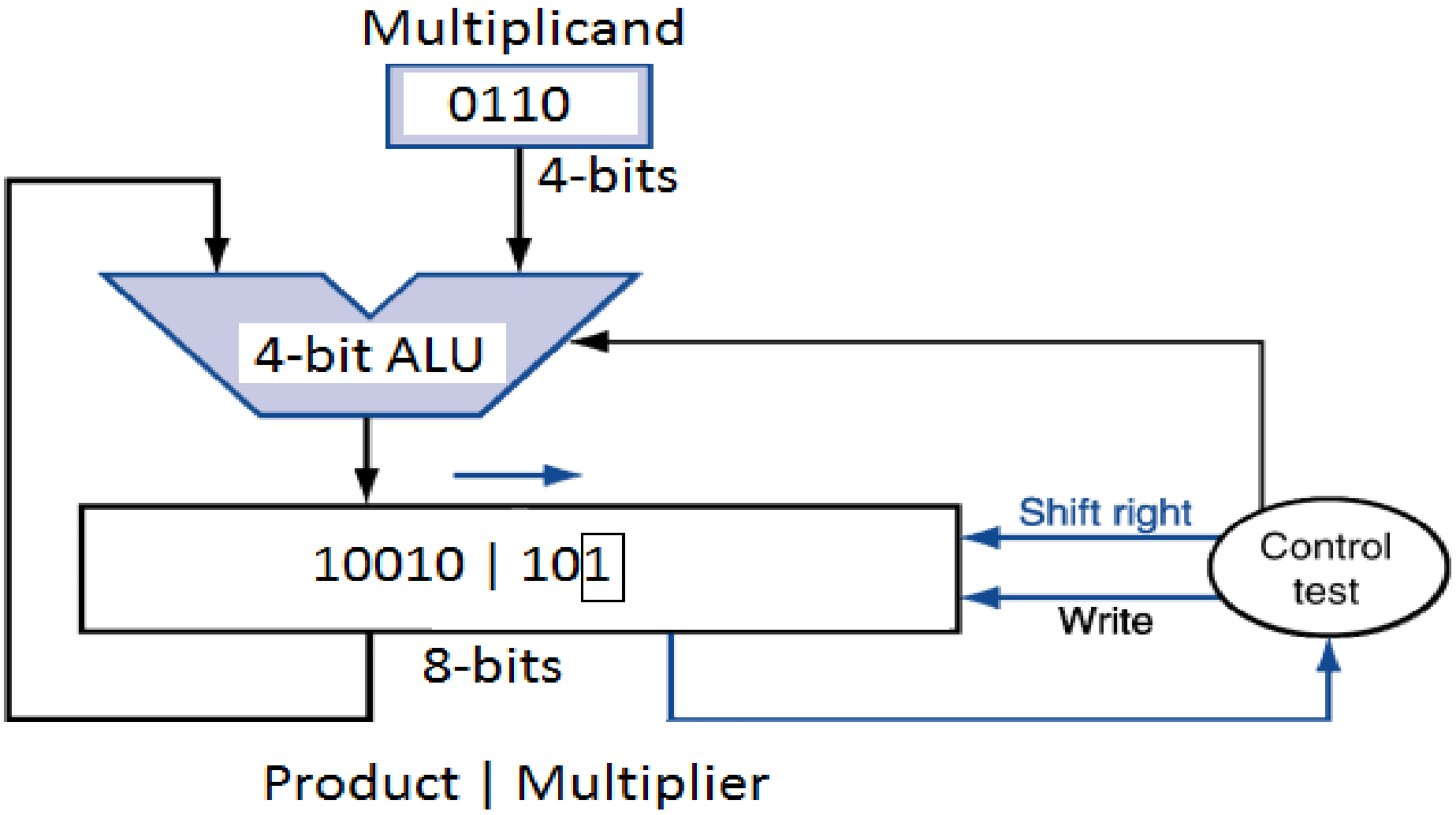
The Refined Multiplier

- Shift product



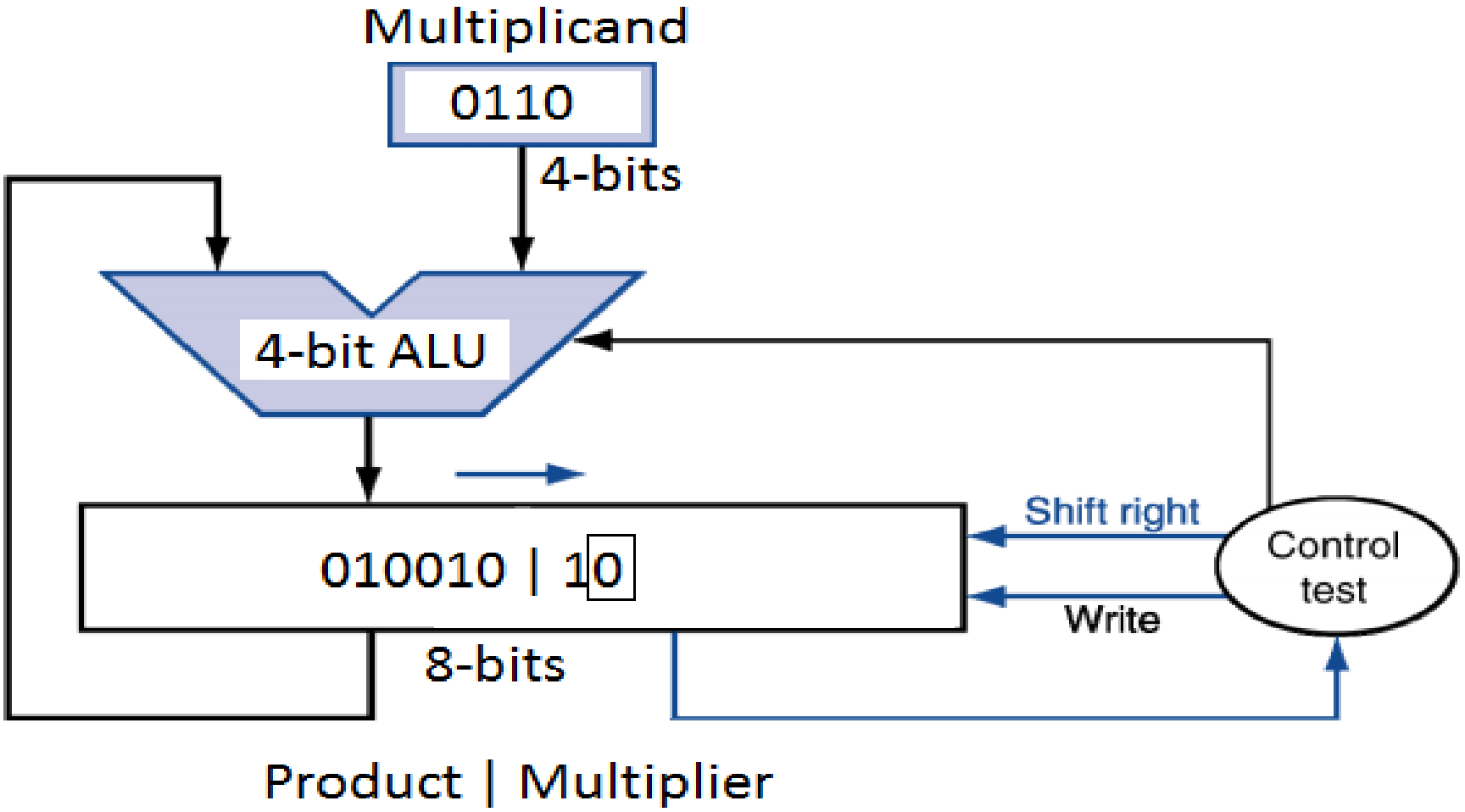
The Refined Multiplier

- Check bit 0 of product (multiplier), add if 1



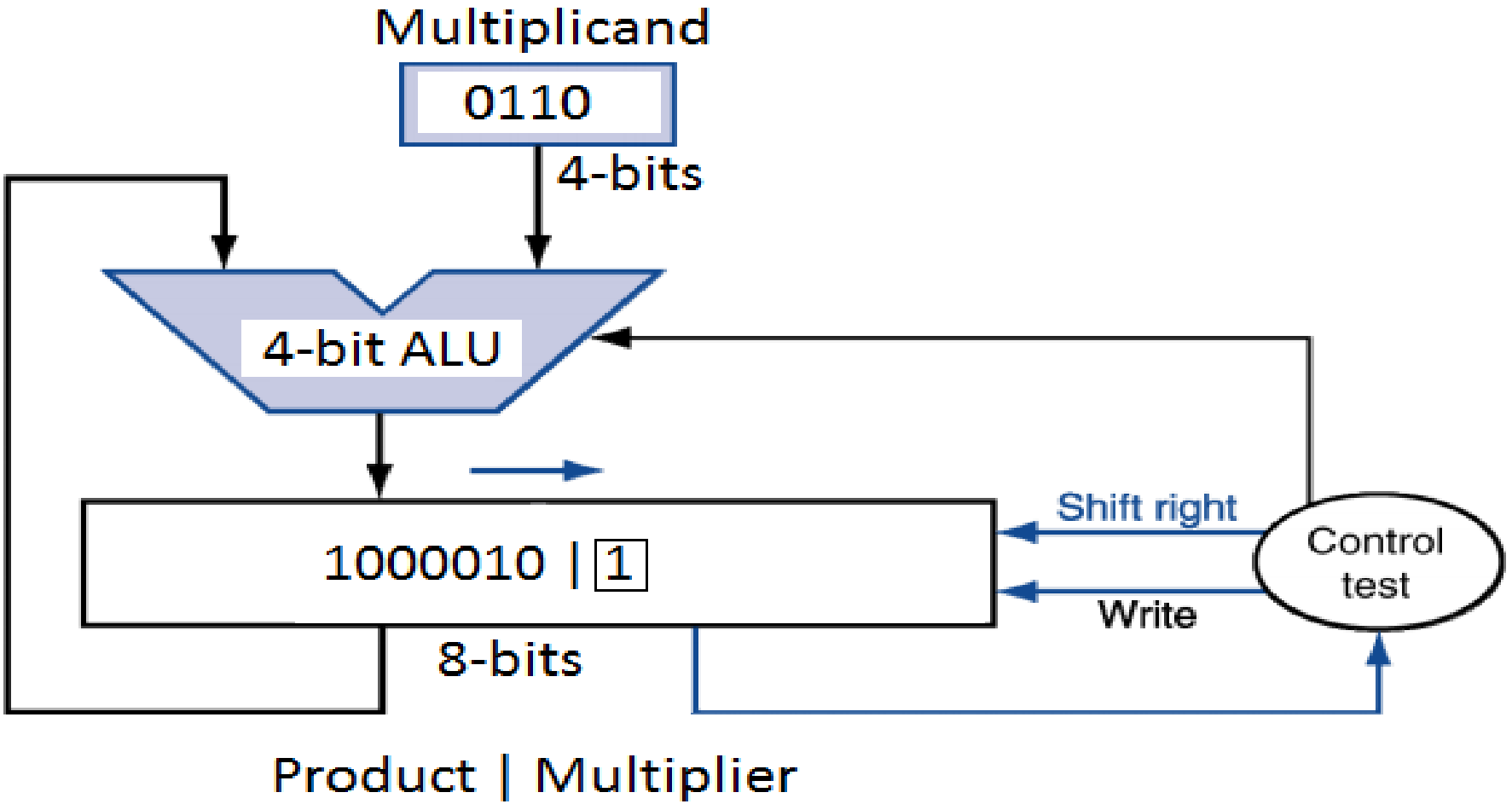
The Refined Multiplier

- ...etc.



The Refined Multiplier

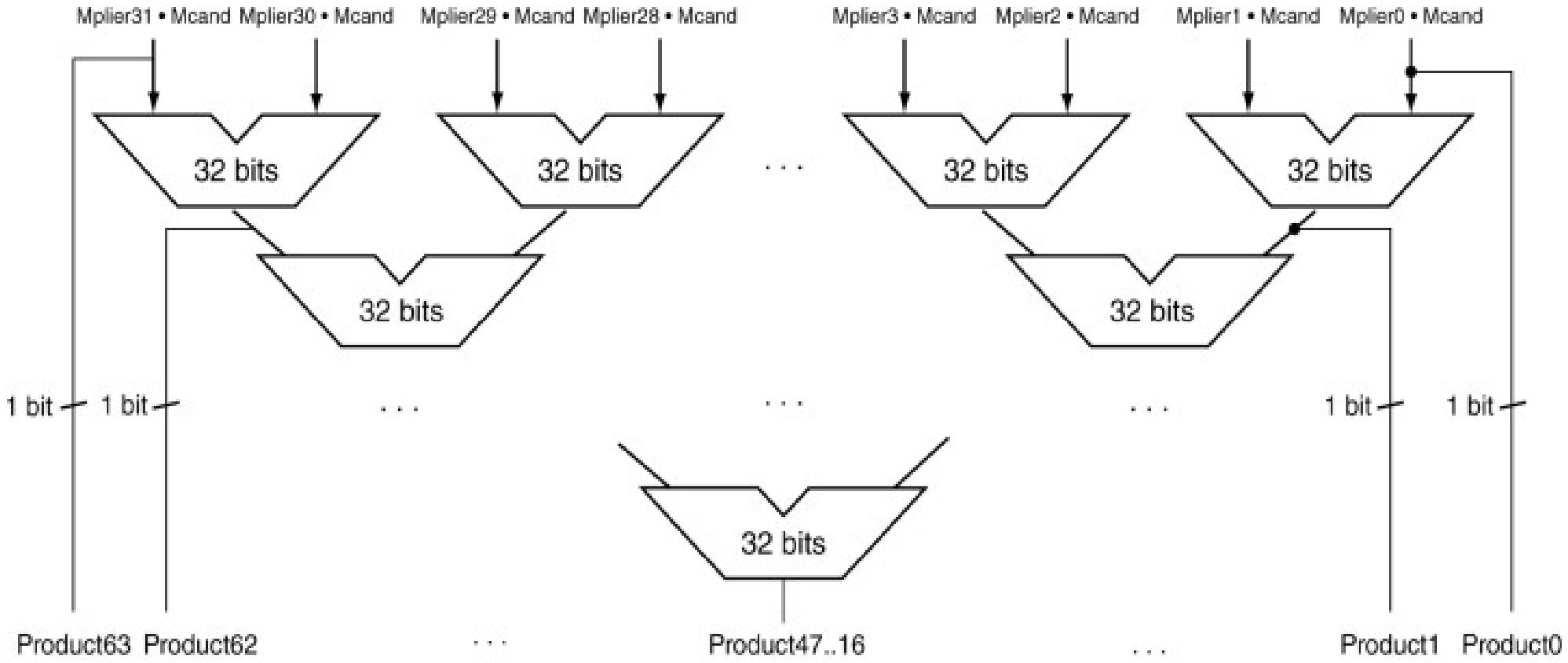
- ...etc.



The Refined Multiplier

- This is a more compact design, but it still takes a n cycles for multiplication of n-bit values.
- Why is this design so costly?
- Fundamentally, we're implementing multiplication of two n-bit numbers as n additions in sequence.
- This is a great use of space (only one ALU).
- Can we learn from our adders and do some of these operations in parallel?

The More Refined Multiplier



- Tradeoffs?

The More Refined Multiplier

- Two interpretations for the organization of this adder.
  - A single pass through the adder is 1 cycle.
  - A single pass through a row of ALUs is 1 cycle, making the entire operation take $\log_2 n$ cycles.
- Consider if we made this a one cycle operation.

The More Refined Multiplier

- Single clock cycle tradeoffs
  - A fairly ridiculous amount of ALUs

    = 16 + 8 + 4 + 2 + 1 = 31 ALUs
  - Clock time must be tuned to account for the latency through $\log_2 n$ ALUs where n is the number of bits we're dealing with. With 32 bits, this will take 5 stages.
  - Can this work better if we split it into multiple cycles?

The More Refined Multiplier

- $\log_2 n$ clock cycle tradeoffs
  - Based on this, each clock cycle only needs to account for the latency of 1 ALU.
  - Still fairly ridiculous amount of ALUs

    = 16 + 8 + 4 + 2 + 1 = 31 ALUs
  - Or is it?!

The More Refined Multiplier

- $\log_2 n$ clock cycle tradeoffs
  - Perhaps you only need one row of n/2 ALUs. Then, for each cycle, you reuse this row of ALUs to do the addition.
  - Allows us to use 16, rather than 31 ALUs. Pretty good.
  - However...

The More Refined Multiplier

- $\log_2 n$ clock cycle tradeoffs
  - Consider the 32-bit case where we need to do 5 stages of additions.
  - Having one row of ALUs means that there can be no pipelining. Each multiplication will take 5 cycles.
  - If we have five rows however, each row can be used concurrently for different multiplications.
  - By the time you read this, you'll to know about pipelining, which means that you'll recognize that in an ideal case, this will allow us to complete a multiplication every cycle if we have a series of multiplications to do!

How do we do signed multiplication?

- The simple method:
  - Check the signs of the operands
  - Convert negative numbers to positive
  - Negate result where appropriate.
- This requires complexity (more hardware) and latency (possible negation of inputs and output)

How do we do signed multiplication?

- A method closer to the existing multiplier:
  - If negative number is in multiplier, do all of the same sequence of adds except when you reach the most significant bit, add the negative version.
- Adds complexity and latency. Is there a more elegant solution for signed?

Some observations

- Shifting is (was?) faster than adding.
- $2^n + 2^{n+1} + \ldots + 2^{n+m} = -2^n + 2^{n+m+1}$
- The sum of any number of consecutive powers of 2 can be reduced to the sum of two powers of two!
- Let's apply these to a method for multiplication

Booth's Algorithm

- Start with multiplicand (called A) and multiplier (called B).
- When the multiplier is shifted right, the bit that is shifted out becomes the “bit to the right” (BttR).
- The BttR starts off as being 0.
- A : 0110 (6)
- -A: 1010 (-6)
- B : 1011 (-5)
- Product:

  0000 | 1011 BttR: 0

Booth's Algorithm

LOOP:

If LSB == 0 and BttR == 0

No-op

else if LSB == 1 and BttR == 0

Add -A to most significant bits of product

else if LSB == 1 and BttR == 1

No-op

else if LSB == 0 and BttR == 1

Add A to most significant bits of product

shift product right

if already shifted n-1 times, done, else goto LOOP

Booth's Algorithm

- A : 0110 = 6
- -A: 1010 = -6
- B : 1011 = -5
- Product:

  0000 | 1011 BttR: 0
- 1.
  - 0000 | 1011 BttR: 0 - Start
  - 1010 | 1011 BttR: 0 - Subtract A
  - 11010 | 101 BttR: 1 - Shift

Booth's Algorithm

2.

- 11010 | 101 BttR: 1 - Init
- 111010 | 10 BttR: 1 - Shift

3.

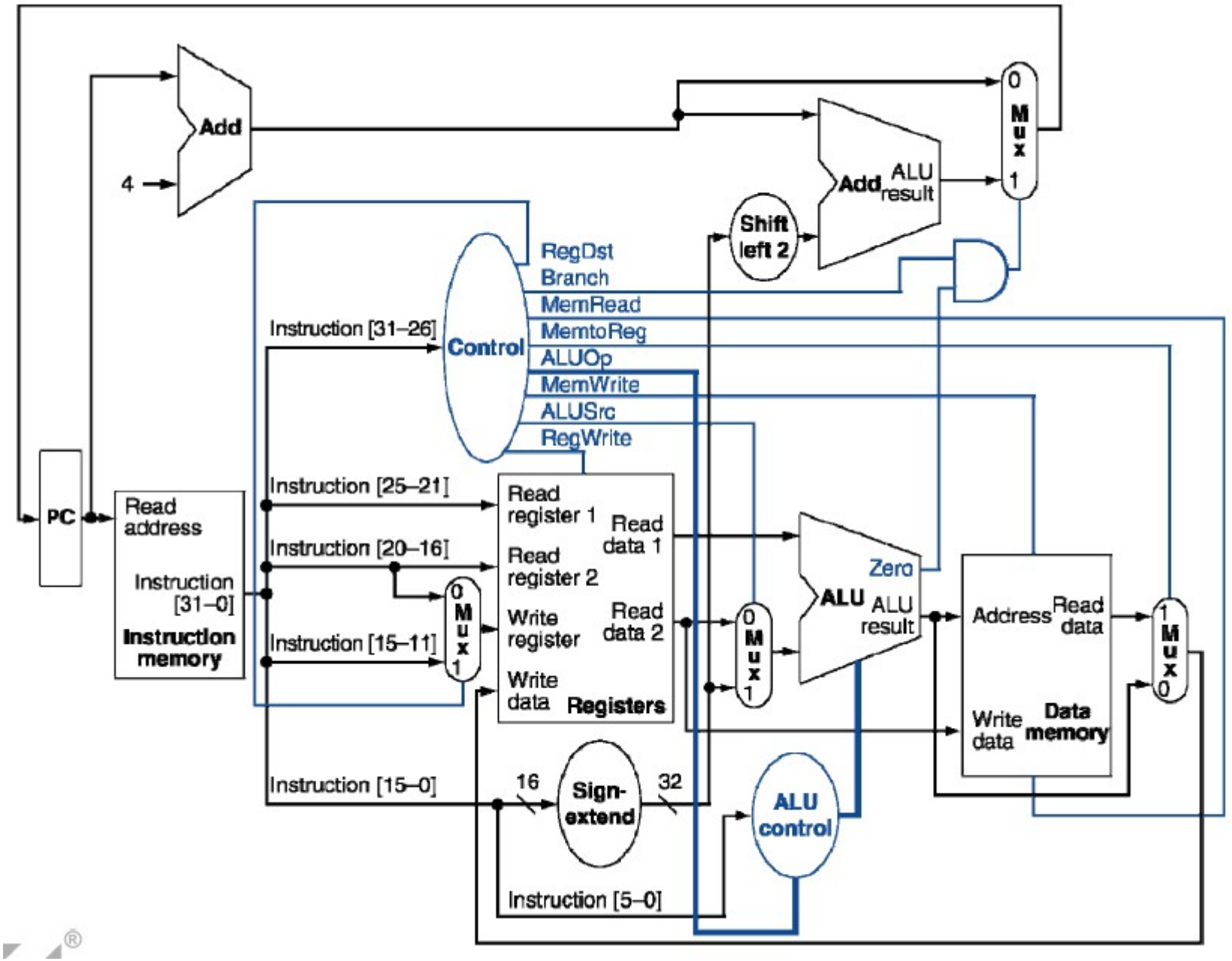
- 111010 | 10 BttR : 1 - Init
- 010010 | 10 BttR : 1 – Add A
- 0010010 | 1 BttR : 0 - Shift

Booth's Algorithm

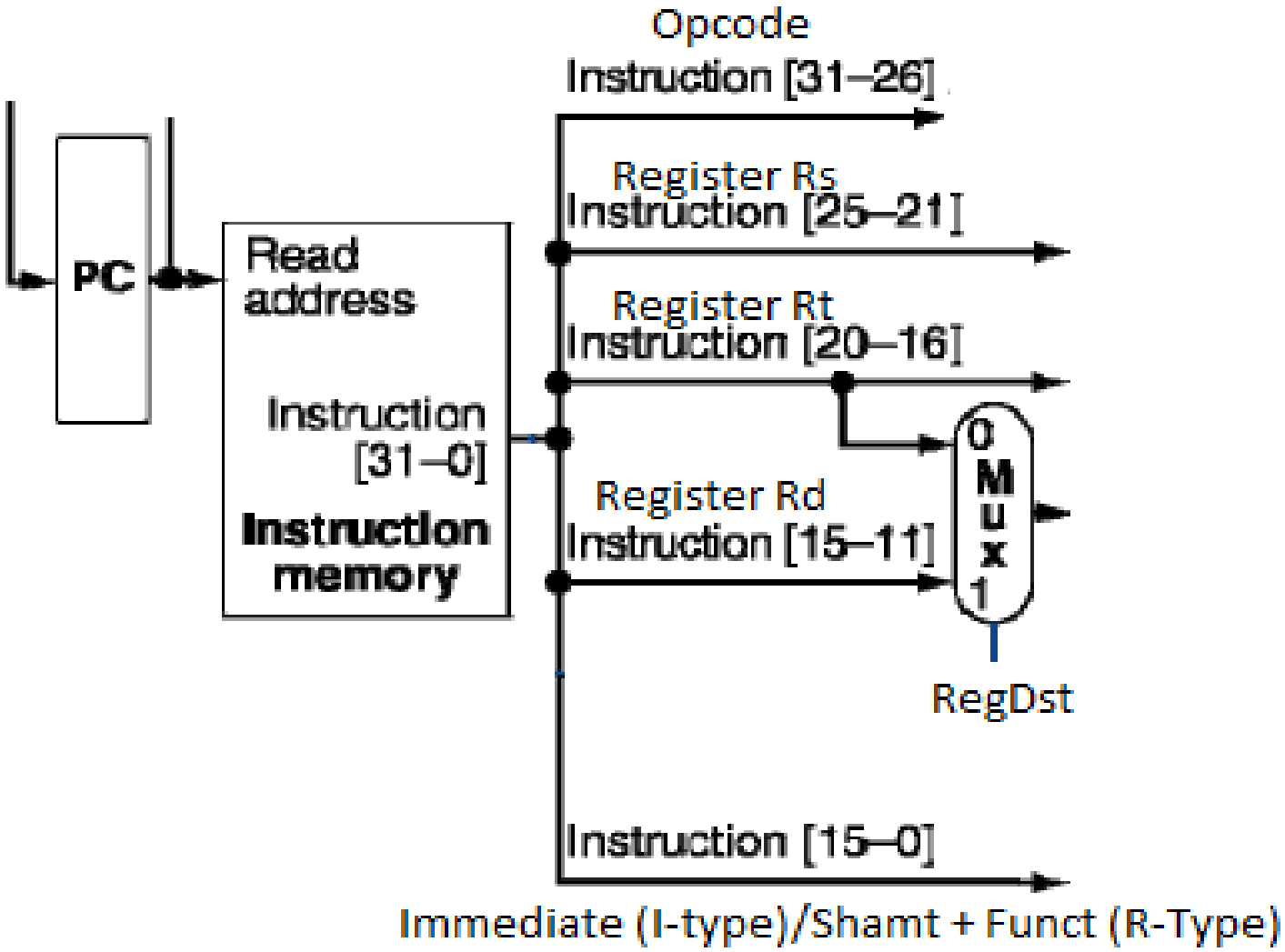
4.

- 0010010 | 1 BttR : 0 - Init
- 1100010 | 1 BttR : 0 – Subtract A
- 11100010 = -30 - Success!

Single Cycle Datapath



Instruction Fetching



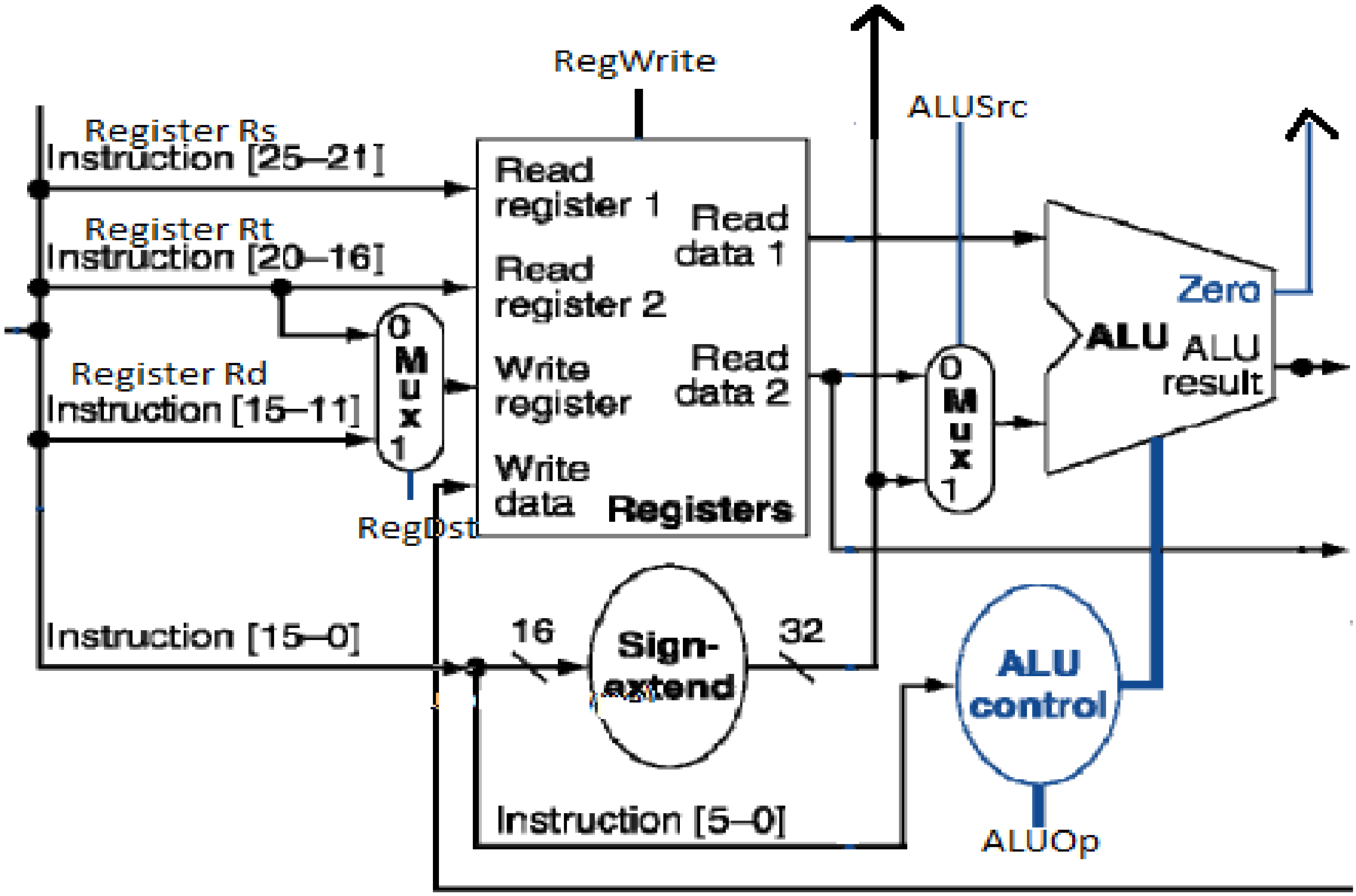
Instruction Fetching

- Opcode is fed into control unit to determine which operation/operation type to use.
- Rs: Address/identifier of the source register. Always read from.
- Rt: Address/identifier of the tsource register. Can be read from (R-Type, store) or can be written to (Load word)
- Rd: Address/identifier of destination register. Can be written to (R-Type), but it can also be the upper 5 bits of the immediate field (I-Type)

Instruction Fetching

- Note: some of the fields are not valid depending on the instruction type.
- For branch, Rd is the upper bits of Imm.
- For jump, Rs, Rt, and Rd are the upper bits of Imm/Addr
- The values are still fed into the register file however so must take care in assigning control.

Read from reg/Arith



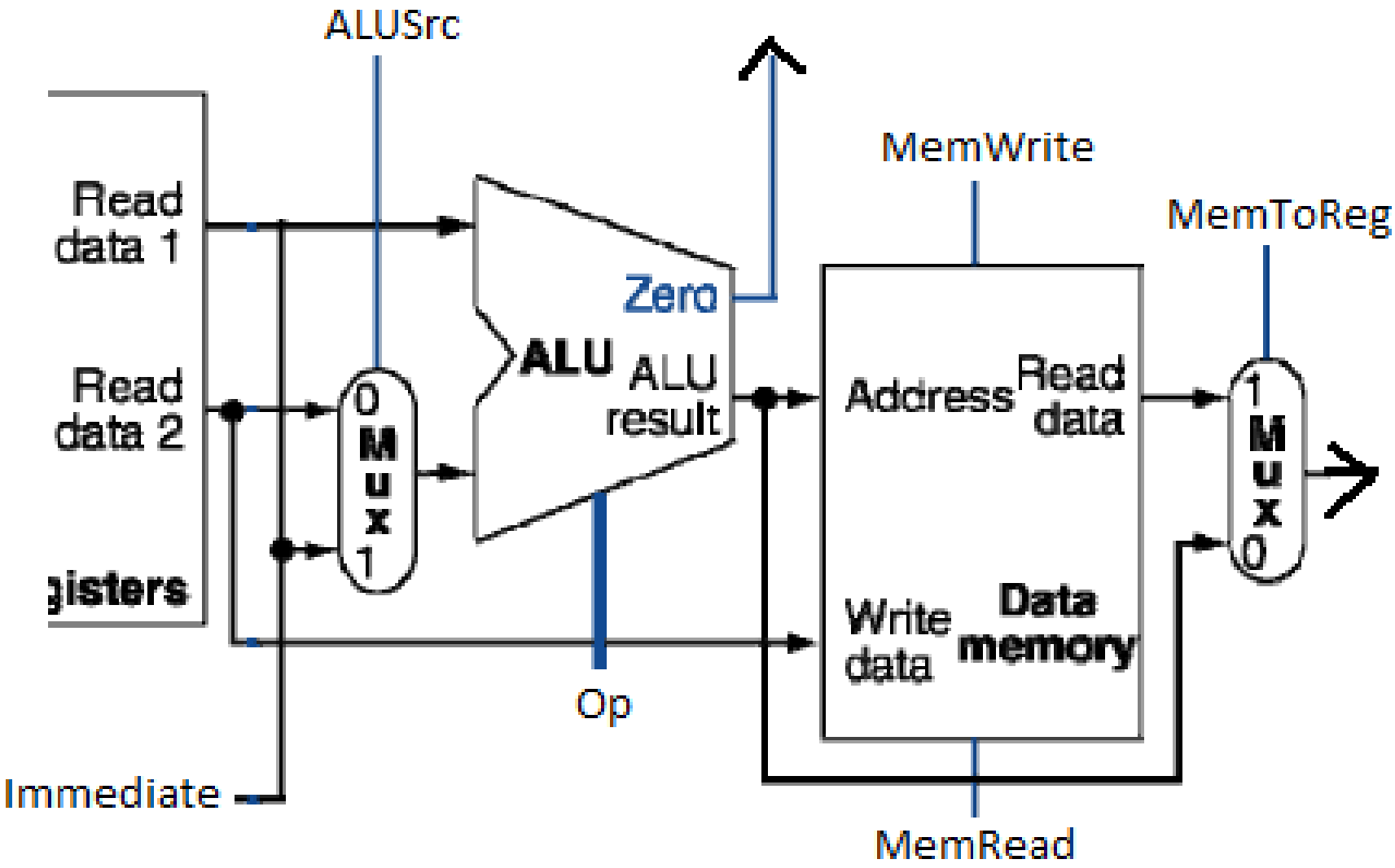
Read from reg/Arith

- Register file allows the reading of two registers and the writing to one register.
- Register written to is either Rt or Rd depending on RegDst
- Rs is always an input to ALU.
- Second input to ALU is either Rt (for R-Type or branch) or sign extended immediate field (load/store) depending on ALUSrc.
- Note: Something is always read from register file and passed to the ALU.

Read from reg/Arith

- Opcode determines ALUOp.
- If opcode corresponds to load/store, ALU control ignores the bits that would be the funct of an R-Type and always tells ALU to add.
  - lw $1, 4($2) <=> must add 4 with R[$2]
- If opcode corresponds to branch, ALU control always tells ALU to sub.
- If opcode corresponds to R-Type (000000), ALU control considers the funct field to decide which operation to do.

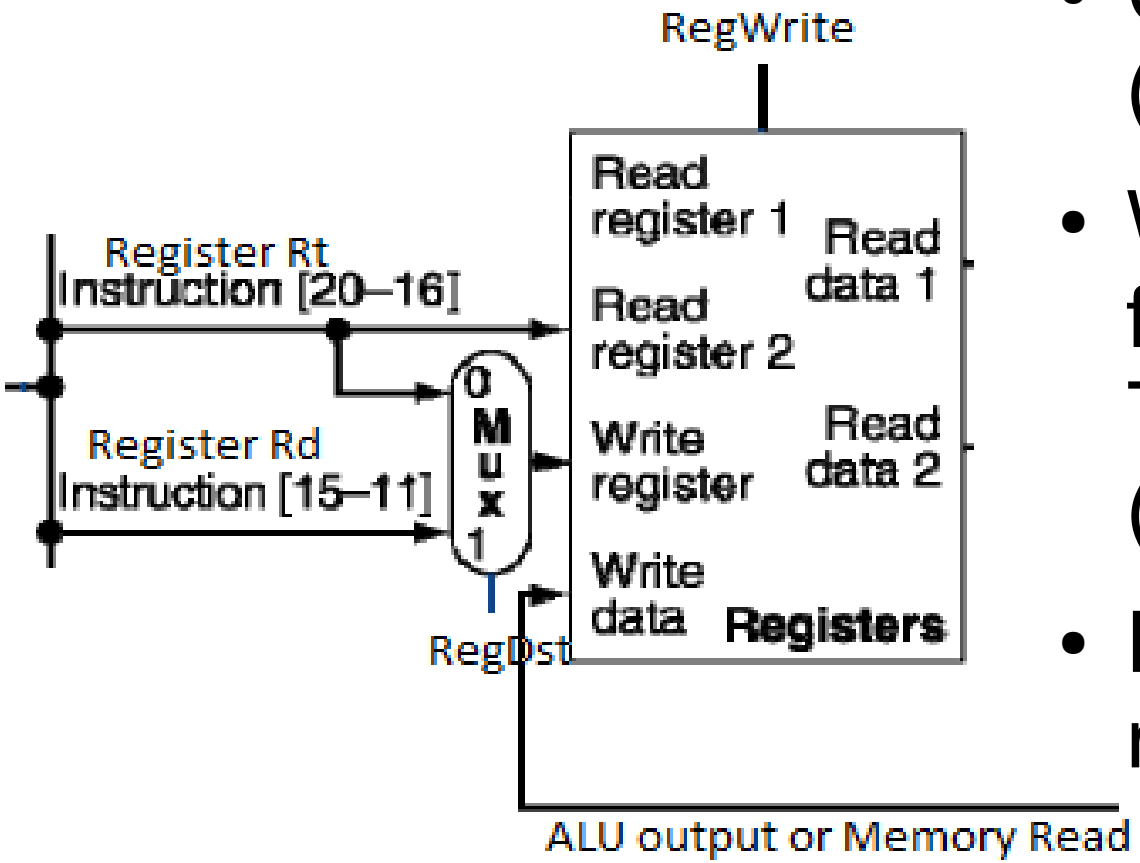
Memory Read/Write



Memory Read/Write

- MemRead controls whether or not to read (which will output from Read Data)
- MemWrite controls whether or not to write to the address specified by “Address”
- Address specified by output of ALU
- Write data specified by Rt
- MemToReg specifies whether to save data stored from Memory or to save output of ALU.

Write back to reg



- Can either write to Rt (load) or Rd (R-type)
- Write data comes from result of ALU (R-Type) or memory (load)
- RegWrite controls if register is written to

Branch Logic

- Consult full datapath.
- PC + 4 is always computed using a specialized ALU.
- The immediate field (bits 15 to 0) is always sign extended and shifted left by 2 (AKA multiplied by four), regardless of instruction type.
- PC + 4 + 4*Imm is always calculated by yet another specialized ALU.
- The Branch signal decides whether to store PC + 4 or PC + 4 + 4*Imm into the PC register.

“Interesting” Facts

- There is no RegRead signal to determine whether or not to read from the register, even though we will assume some instructions (jump) where we don't need to read from the register.
- However, there is a MemRead signal. Why?

“Interesting” Facts

- The register file will consider inputs 00000 to 11111, which are all valid registers.
- The memory module can take 32-bit inputs, which means that arbitrary address can be invalid memory accesses.

“Interesting” Facts

- Recall the memory hierarchy.

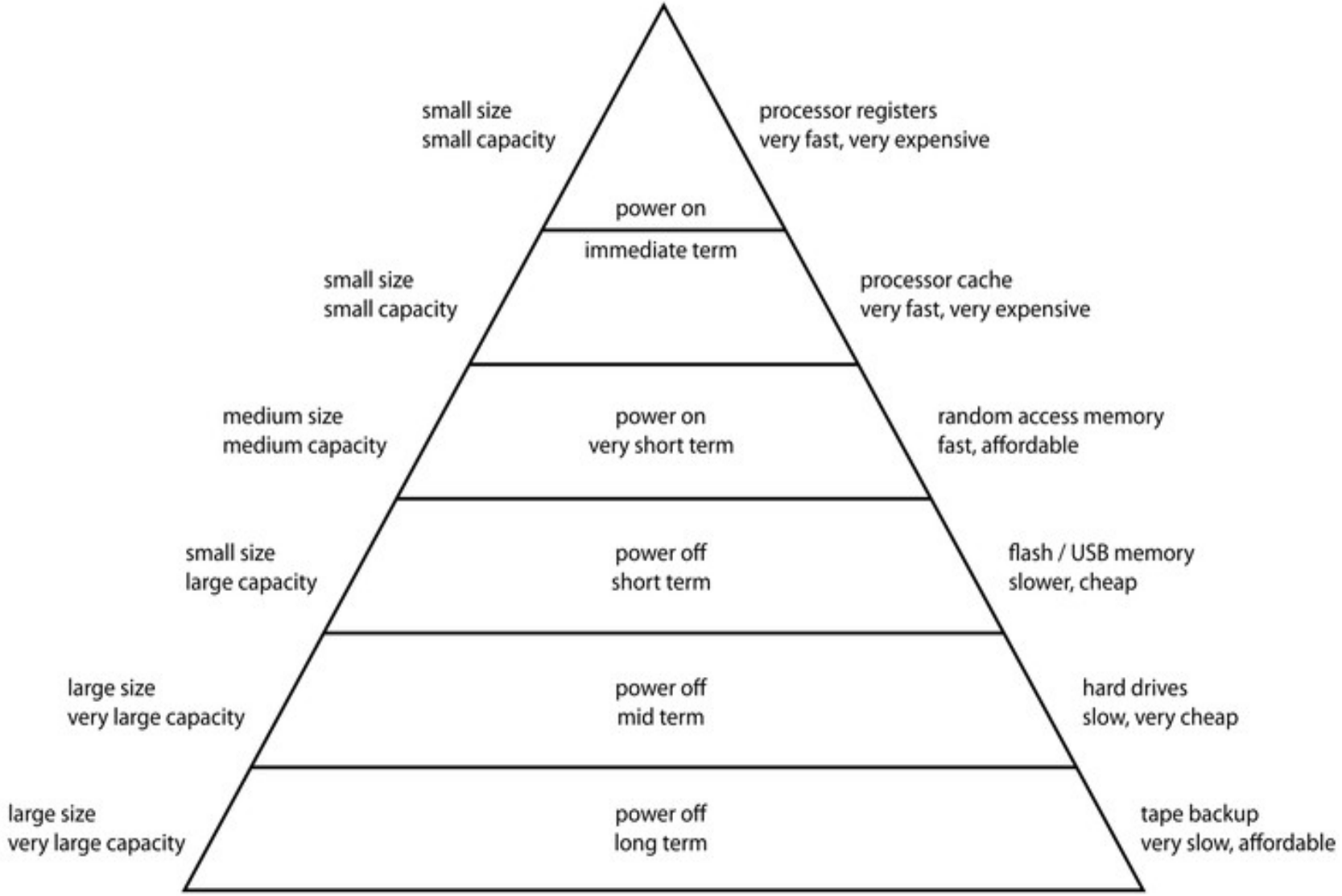


Image courtesy of Dr. Wikipedia

“Interesting” Facts

- Accessing the registers is a trivial task when it comes to latency. Given the bare minimum of components we will use (jump, which still uses an ALU to compute PC+4), always reading the registers is not a problem.
- Accessing memory is comparatively extremely expensive (100-200 times slower!).
- But wait...

“Interesting” Facts

- If the clock time has to be adjusted to be the max latency through the datapath, does the clock cycle have to account for the huge delay of memory accesses?

“Interesting” Facts

- The two memory modules in the datapath are actually Data and Instruction Caches, which produce acceptable latency.
- The datapath assumes that all memory accesses can be satisfied by the cache.
- If there is a cache miss, the processor must halt either by a “stall” or an exception

“Interesting” Facts

- Most traditional memory systems will force the processor to go through a cache in order to actually access memory.
- Let's revisit the question of why there is a MemRead but no RegRead.
- Because the “data memory” and “instruction memory” are actually caches, if the processor made arbitrary (but valid) memory accesses, these arbitrary memory blocks would be pulled into caches.
- This would ruin the locality provided by the caches. Hence, MemRead.
- But more on caches later...

Ex: Extending Datapath for jri

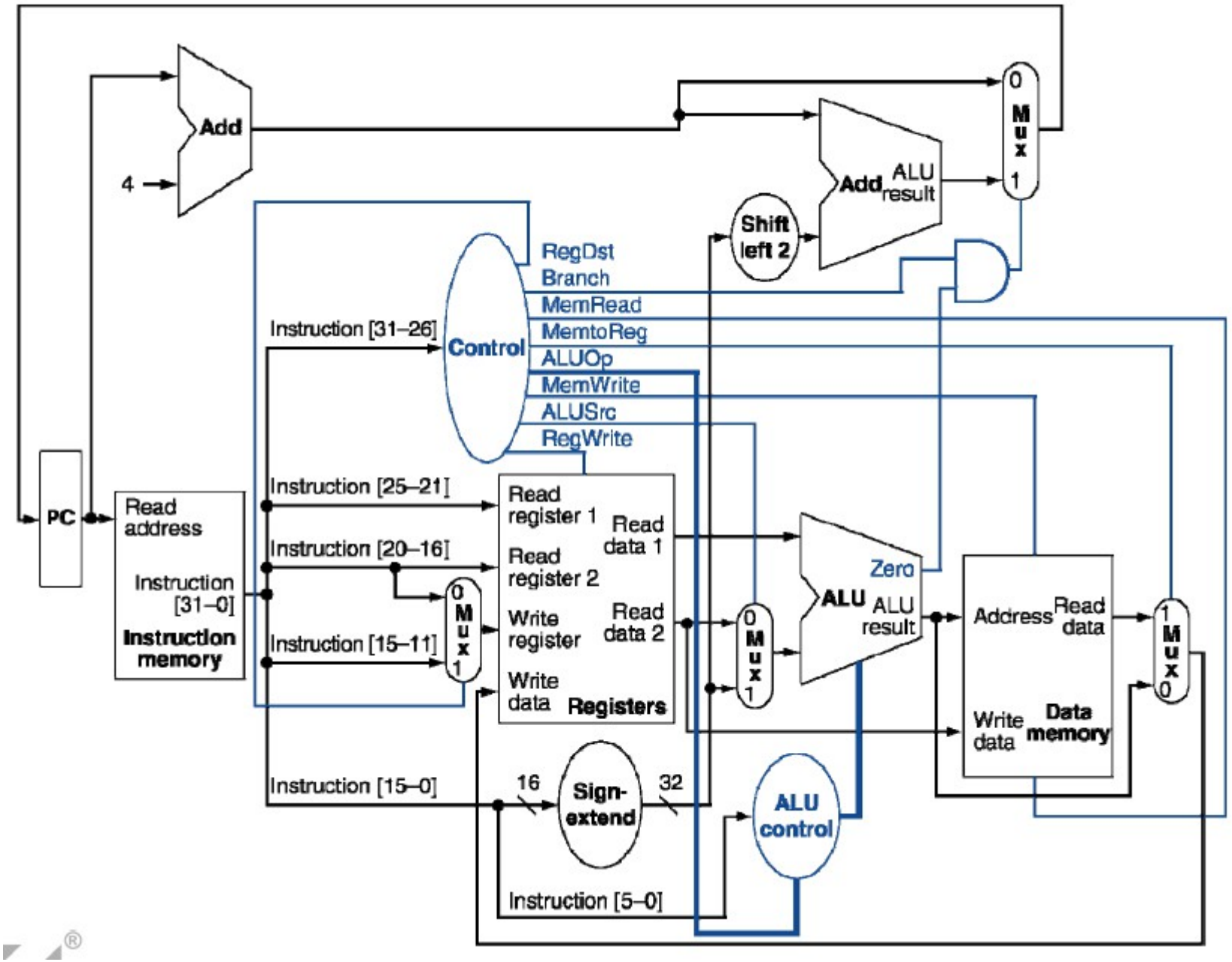
- jri (jump register + immediate)
- PC = R[rs] + Sign Extend(I)
- jri $1 IMM : PC = R[$1] + Sign Extend(IMM)
- What type of instruction should this be?

Following two examples courtesy of David Felber

Ex: Extending Datapath for jri

- What type of instruction should this be?
  - We need a register and an immediate. I-type fits the bill.
  - We'd have a unused register if we implement it as an I-type. We could possibly optimize by defining a new instruction type that has one register and one immediate. Tradeoffs?
  - Let's not do that now.
- What connections must be made?

jri


Add
4
0
M
u
x
1
Add
ALU result
Shift left 2
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
16
Sign-extend
32
ALU control
Zero
ALU
ALU result
Address
Read data
Write data
Data memory
1
M
u
x
0

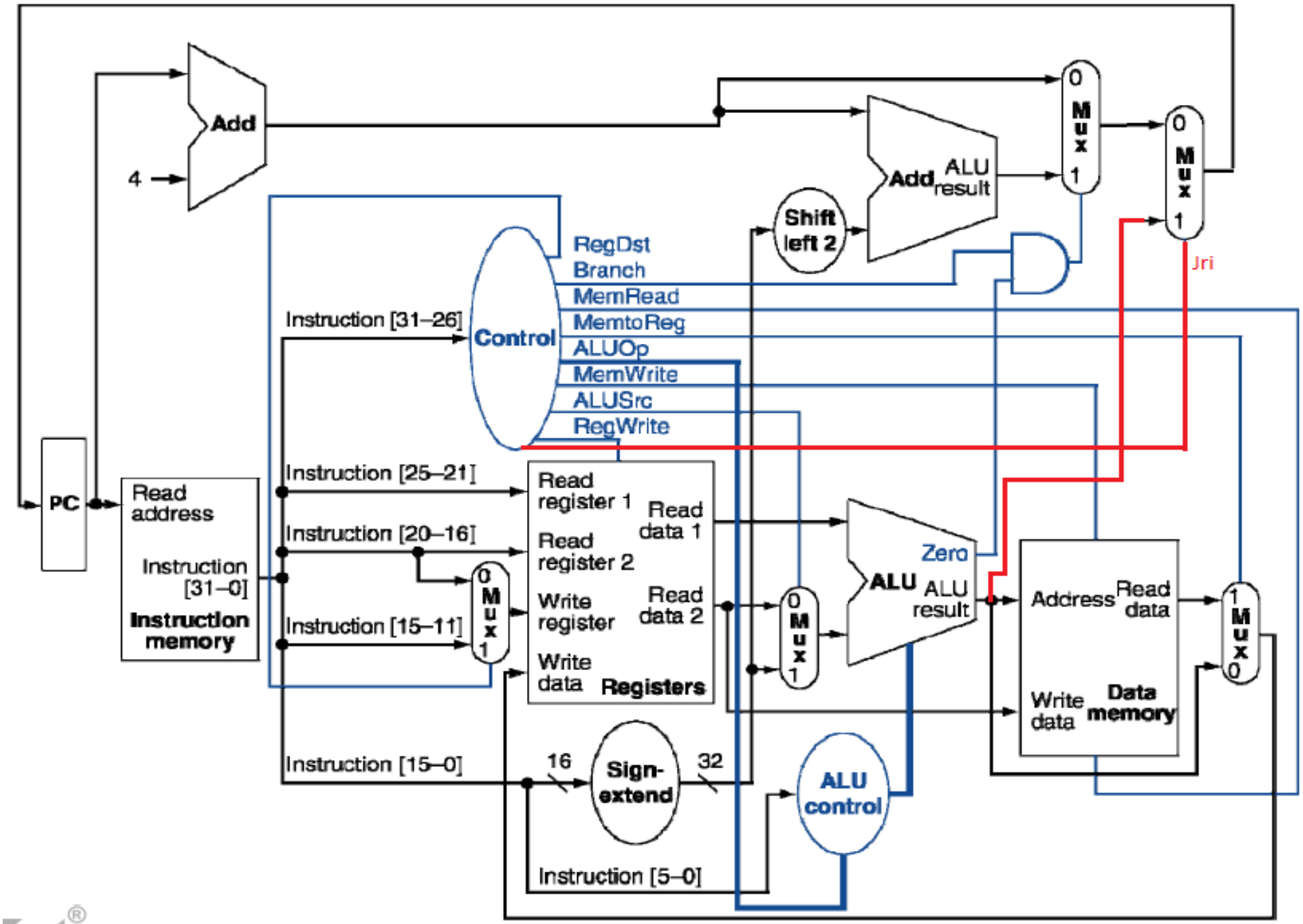
jri

- Additional connections:
  - From output of ALU to PC
  - Already hooked up in such a way as to do addition of register value and a sign extended immediate.
- How do we choose the output of the ALU as the input to the PC?

jri

- How do we choose the output of the ALU as the input to the PC?
  - Need a MUX somewhere before the PC. Presumably also a new control signal to indicate jri.
  - Does it matter if it comes before or after the existing one?

jri


Add
4
Add
ALU result
Shift left 2
0 M u x 1
0 M u x 1
Jri
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
16
Sign-extend
32
ALU control
Zero
ALU
ALU result
Address
Read data
Write data
Data memory
0 M u x 1
1 M u x 0

jri

- What are the control signals?

jri

- What are the control signals?
  - RegDst = don't care, not writing to registers
  - Branch = don't care, choosing from ALU output.
  - MemRead = 0, not reading from mem
  - MemToReg = don't care, not writing to registers
  - MemWrite = 0, not writing to mem
  - ALUSrc = 1, add immediate
  - RegWrite = 0, not writing to regsiters
  - Jri = 1, because of course
  - ALUOp = 00, add regardless of funct

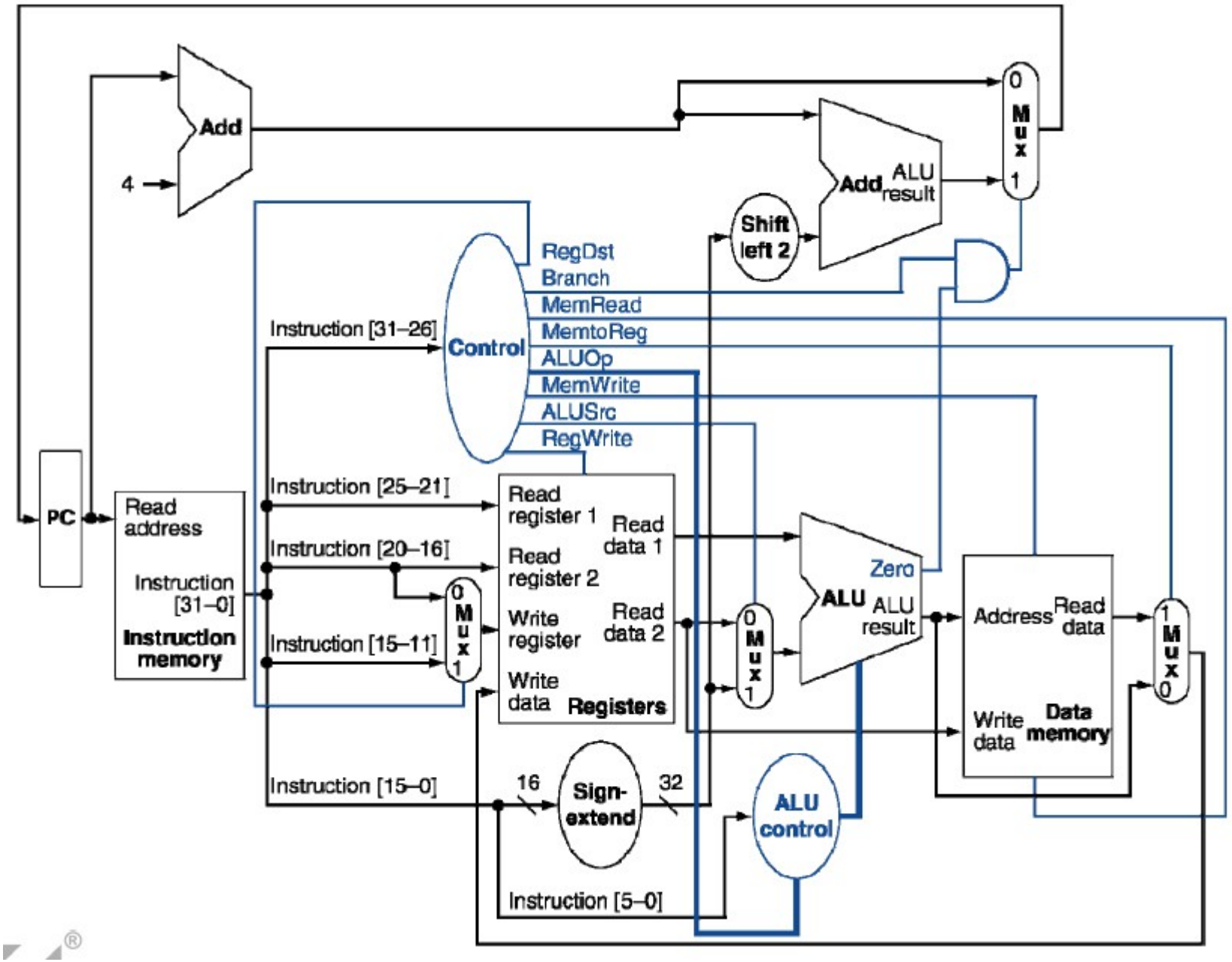
Ex: Extending Datapath for lsr

- lsr:
- R[rd] = R[R[rs][4:0]], AKA storing the value in the register that is specified by the least 5 significant bits of register rs.
- What types of instruction?

Ex: Extending Datapath for lsr

- What types of instruction?
  - We need two registers. This leaves us with R-Type and I-Type. Which is better?
  - I-Type allows us to deal with an immediate.
  - R-Type allows us to specify a specialized ALU operation.
  - lsr needs neither. Let's just use R-Type, specifically rs and rd while rt is unused.
- What connections?

Isr


Add
4
0
M
u
x
1
Add
ALU result
Shift left 2
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
16
Sign-extend
32
ALU control
Zero
ALU
ALU result
Address
Read data
Write data
Data memory

Ex: Extending Datapath for lsr

- What connections?
  - This is trickier because we need to get a value from a register and use that as an address into another register.
  - This means we have two register accesses, but we can't do them simultaneously.
  - Read Register 1 can be reserved for the access to R[rs] while Read Register 2 will be used for the second register access.
  - This will require a connection from Read Data 1 back to Read Register 2.

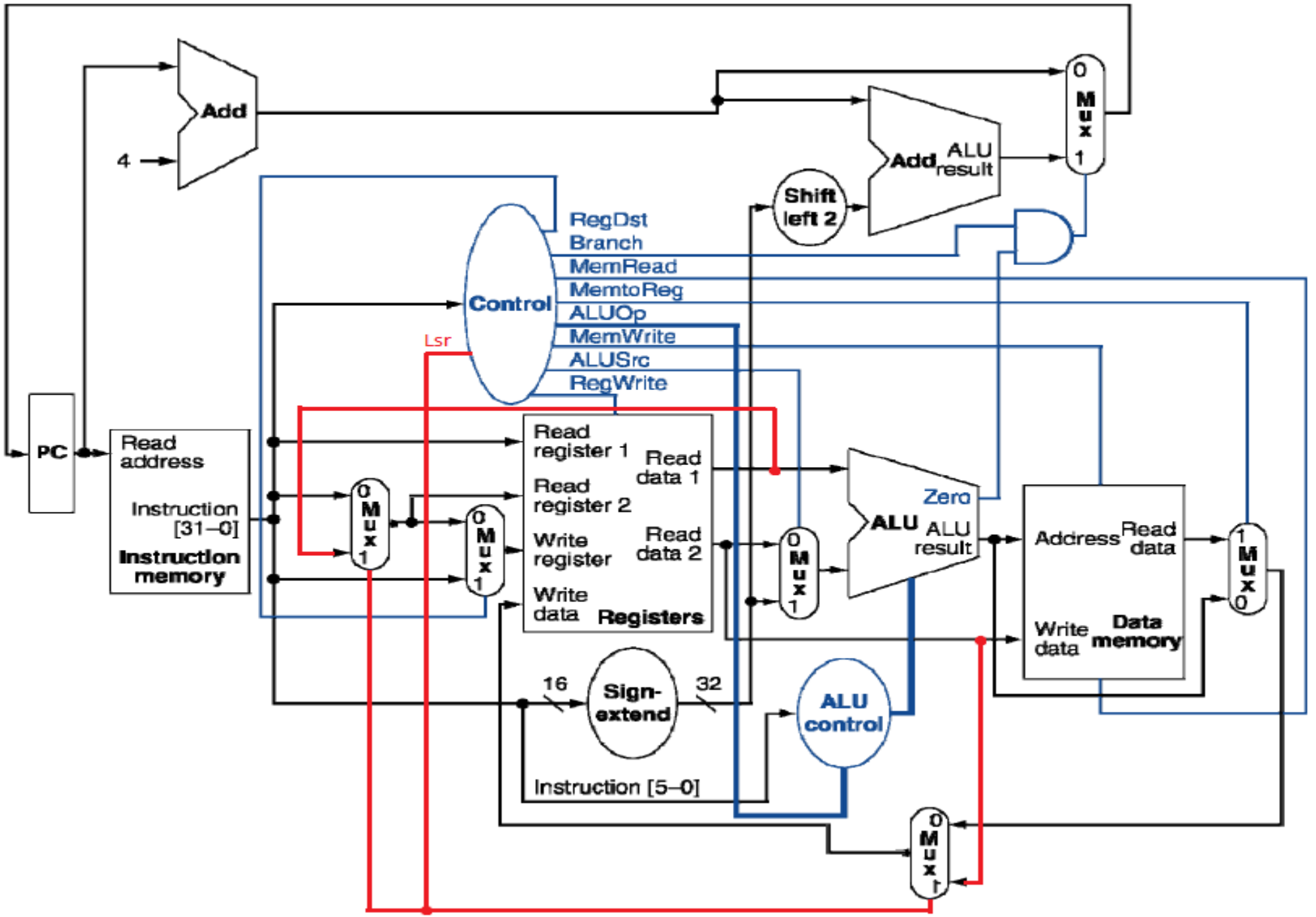
Ex: Extending Datapath for lsr

- What connections?
  - After reading the second register, must be able to connect it back to the Write Data port of register file.
- Any additional hardware?

Ex: Extending Datapath for lsr

- Additional hardware?
  - MUX to choose the input to Read Register 2 as either the output of Read Data 1 or rt.
  - MUX to choose the input to Write Data as either the output of Read Data 2 or the normal output from ALU/Data Memory
  - Two signals to control the MUXs?

lsr


Add
4
M
u
x
0
1
Add
ALU result
Shift left 2
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Lsr
PC
Read address
Instruction [31–0]
Instruction memory
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Zero
ALU
ALU result
Address
Read data
Write data
Data memory
16
Sign-extend
32
ALU control
Instruction [5–0]

Ex: Extending Datapath for lsr

- In this diagram, the control signals are determined in the control unit based on the opcode.
- However, all R-Type opcodes are 000000! This operation demands different control signals from all other R-Types. This won't work!

Ex: Extending Datapath for lsr

- We can resolve this in several ways:
  - ALU Control, which examines the funct field, is allowed to modify each control signal. This would require multiplexers and addition logic for every signal that we might have to override. What a mess.
  - Feed the funct field into the Control module so that it also considers the funct field, when necessary. Reasonable.
  - If this comes up in homework or a midterm, the professor has allowed you simply to define a new unique R-Type opcode that isn't 000000. That takes care of everything.

lsr

- What are the control signals?
  - RegDst = 1, destination is register rd
  - Branch = 0, not branching
  - MemRead = 0, not reading from mem
  - MemToReg = don't care, not choosing that signal
  - ALUOp = don't care, not using ALU
  - MemWrite = 0, not writing to mem
  - ALUSrc = don't care, not using ALU
  - RegWrite = 1, writing to register
  - Lsr = 1,

End of

The Fourth Week

-Six Weeks Remain-