

COMP 30080 Processor Design

3. Single Cycle MIPS Processor

Dr Chris Bleakley



**UCD School of Computer Science
and Informatics.**

**Scoil na Ríomheolaíochta agus an
Faisnéisíochta UCD.**

Introduction

1. Organisation

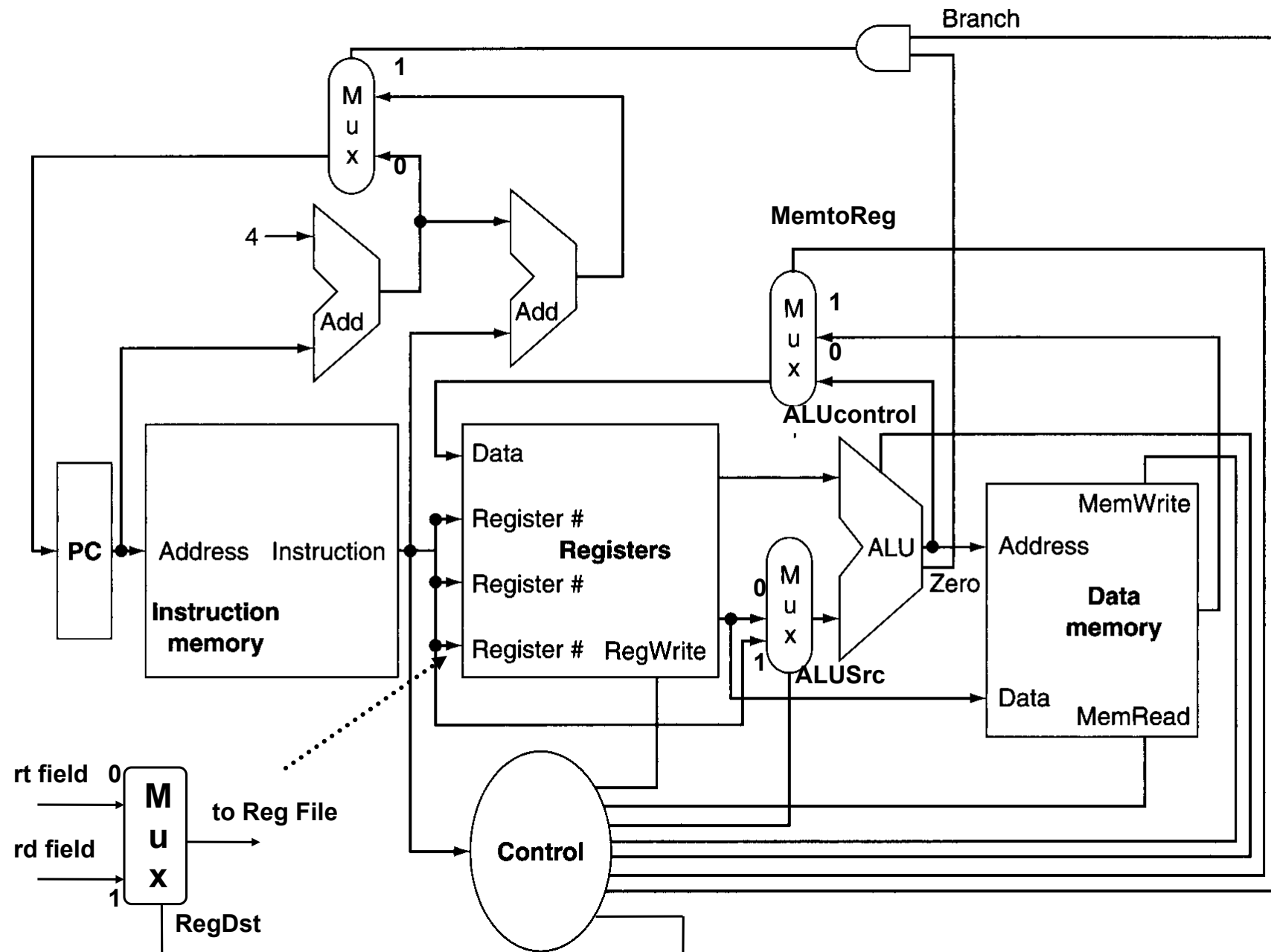
2. Gate-Level Implementation

3. Single cycle MIPS Processor

Organisation

MIPS Processor

- Supports subset of core MIPS instruction set:
 - Memory access - `lw`, `sw`
 - ALU - `add`, `sub`, `and`, `or`, `slt`
 - Flow control - `beq`, `j`
- All instructions take a single clock cycle to perform. Von Neumann cycle:
 - Program Counter (PC) use to address memory and fetch instruction.
 - Control Unit decodes the instruction and sends control signals to functional units.
 - Functional units execute instruction.
- Uses separate instruction & data memories.



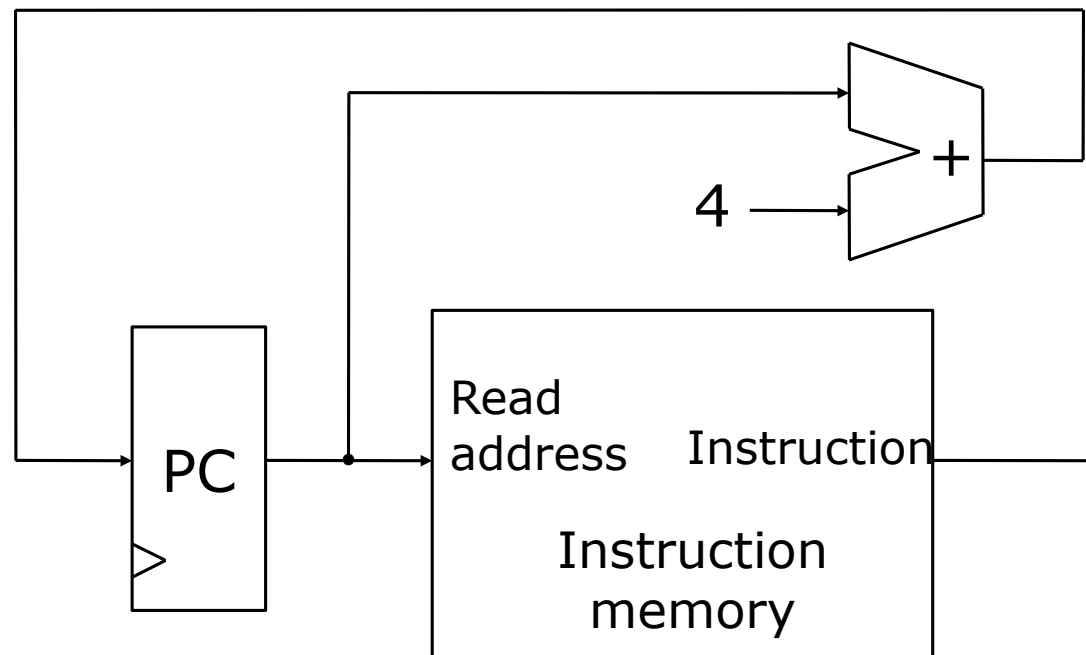
3. Single cycle MIPS Processor

What functional units are needed?

- Program Counter
 - register
 - adder to increment PC and way to select between address of branch target and next sequential address
- Main memory
 - Instruction memory, read only
 - Data memory, read and write
- Registers
 - Register file, at most read two and write one register per instruction
- ALU
 - add, subtract, compare, or, and, etc.
- Control Unit

Program Counter

- Datapath element: "A functional unit used to operate on or hold data within a processor".
- Program Counter (PC): "The register containing the address of the instruction being executed."



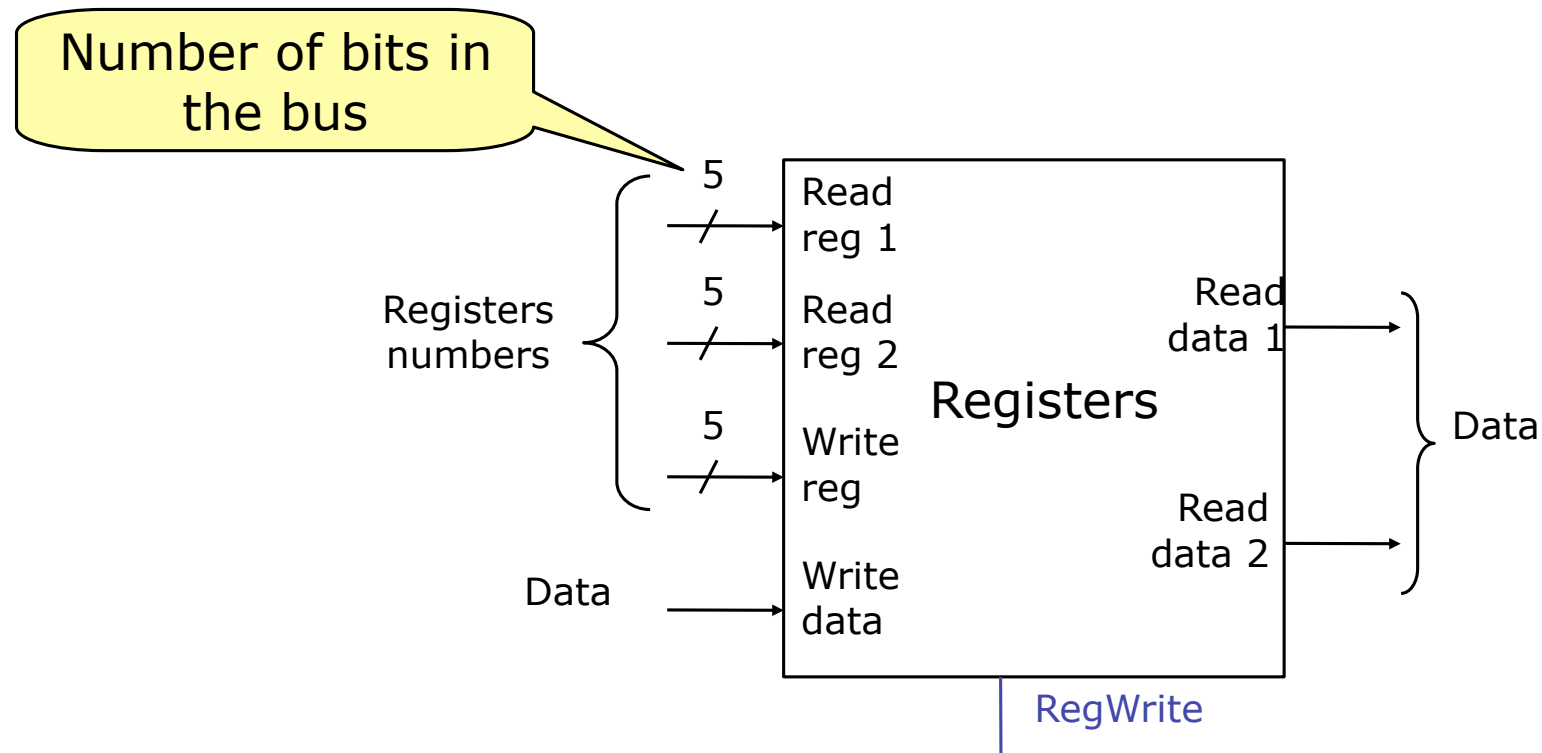
Program Counter

- PC register
 - Function: Holds address of the current instruction. Loads input into register and on to output on the positive edge of the clock.
 - Input: Address of next instruction
 - Output: Address of current instruction
- Instruction memory
 - Function: Stores machine instructions.
 - Input: Instruction address
 - Output: Machine instruction contained stored at that address
- Adder
 - Function: Adds two numbers
 - Input: Two numbers
 - Output: Total of the numbers

Datapath

Register File

- Register file: "A state element that consists of a set of registers that can be read and written by supplying a register number to be accessed."



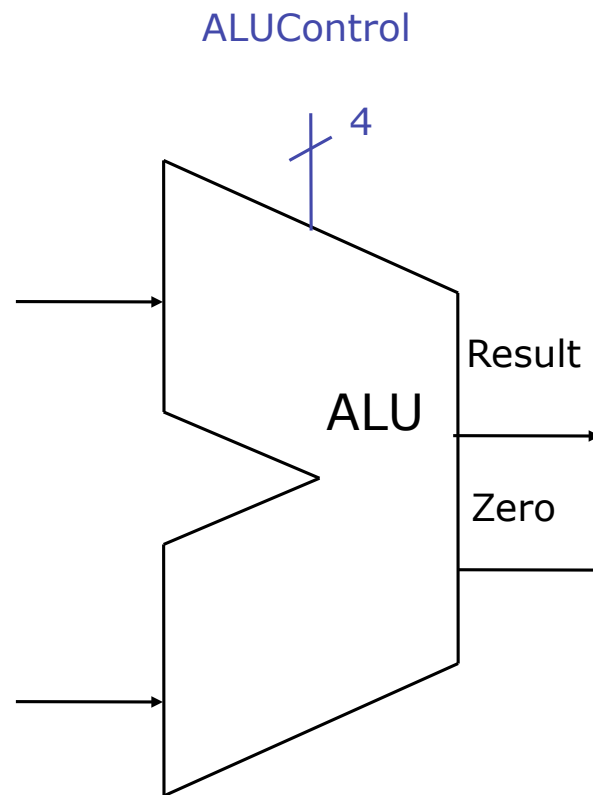
Datapath

Register File

- Function: Stores data in registers (numbered 0-31). Can read two registers and write one register at a time. The written register is updated on the positive edge of the clock.
- Inputs:
 - ReadReg1: number of register to be read.
 - ReadReg2: number of register to be read.
 - WriteReg: number of register be written.
 - Write data: data to be written.
 - RegWrite: control signal. If 1 then do a write. If 0 then do not do a write.
- Outputs:
 - ReadData1: data contained in register num=ReadReg1.
 - ReadData2: data contained in register num=ReadReg2.

Datapath

ALU



Datapath

ALU

- Function: Performs arithmetic or logic operations on data. The operation to be performed is selected using ALUControl.
- Inputs:
 - two data inputs
 - ALUControl: control signal (see next slide).
- Outputs:
 - Result: the result of the operation applied to the data inputs.
 - Zero: Set to 1 if Result is equal to 0. Otherwise set to 0.

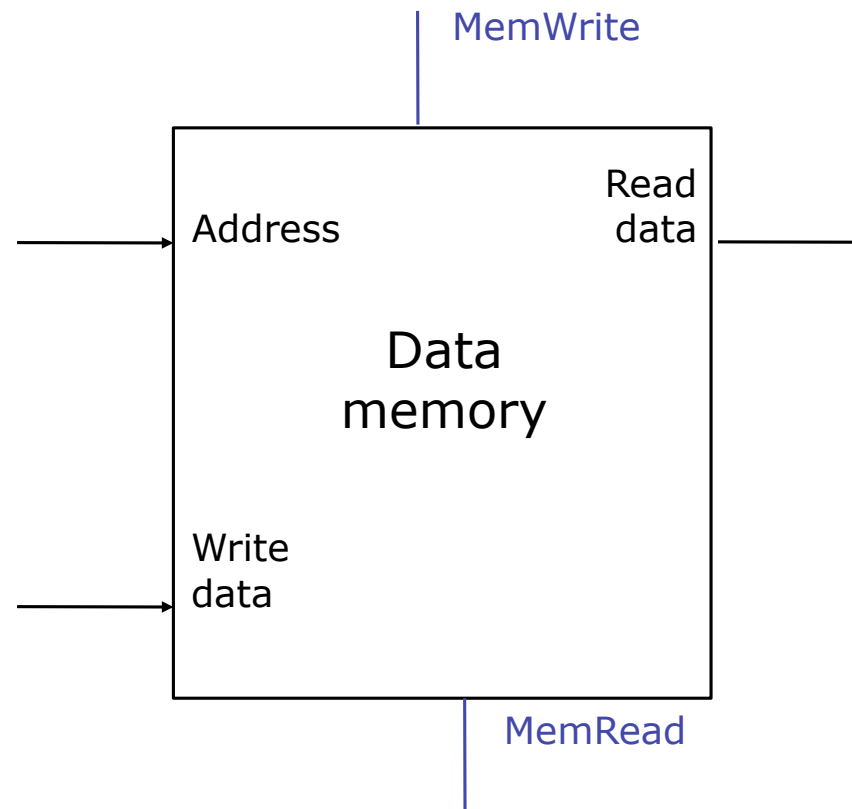
Control

ALU

ALUControl	Function
0000	AND
0001	OR
0010	Add
0110	Subtract
0111	Set on less than
1100	NOR

Datapath

Data memory



Datapath

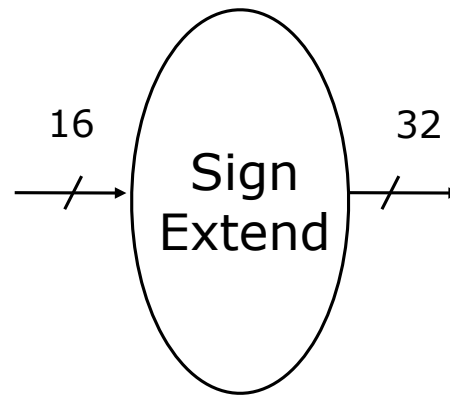
Data memory

- Function: Stores data. Can read or write the contents of a single address every cycle. Read data is available immediately. Writes take place on the positive edge of the clock.
- Inputs:
 - Address: Address of data to be access.
 - WriteData: Data to be written to Address.
 - MemRead: Control signal. If 1 then read data from the memory. Otherwise, do nothing.
 - MemWrite: Control signal. If 1 then write data to the memory. Otherwise, do nothing.
- Output:
 - ReadData: Data contained stored at that address.

Datapath

Sign extension

- Sign extend: "To increase the width of a binary number by replicating the most-significant bit".
- Converts 16-bit 2s complement number to 32-bit 2s complement number (i.e. the sign bit is preserved).



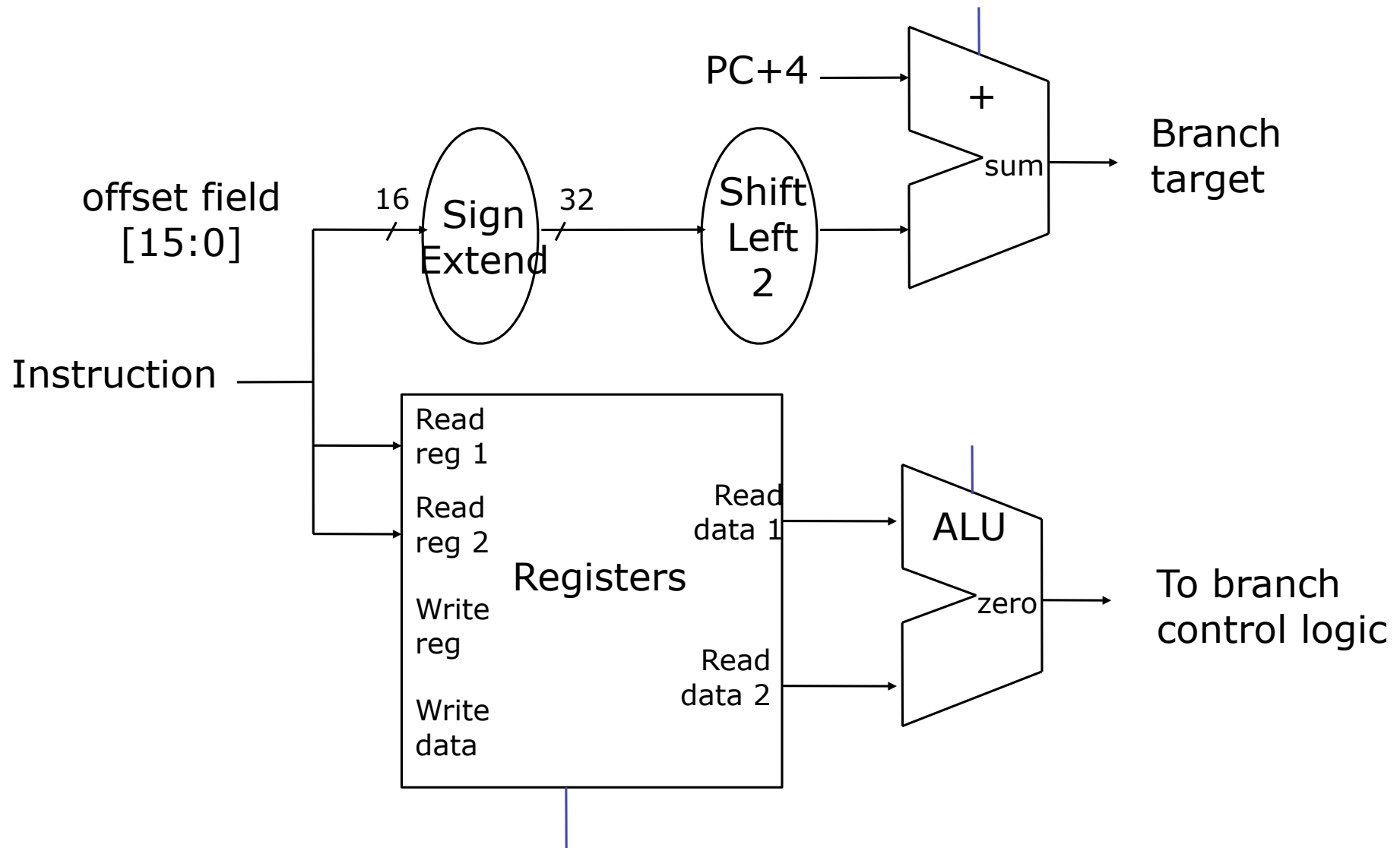
Datapath

Branch

- Branch instructions require selection between two different addresses for the next instruction:
- EITHER
- Condition true. Branch taken. $PC \leq \text{target}$
 - Branch target address: "Address specified in a branch instruction."
- OR
- Condition false. Branch not taken. $PC \leq PC+4$

Datapath

Branch



Control

- We now have all the functional units we needed and know what they do.
- How do we coordinate their activity so that the right instruction is performed?

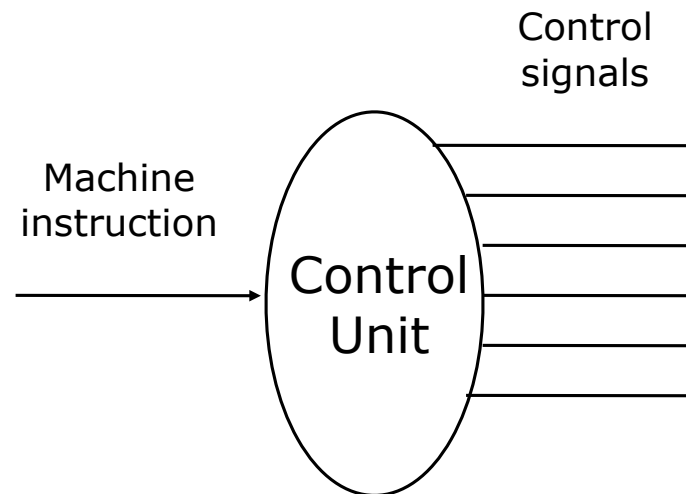
1. Control Unit

2. Multiplexer

Control

Control Unit

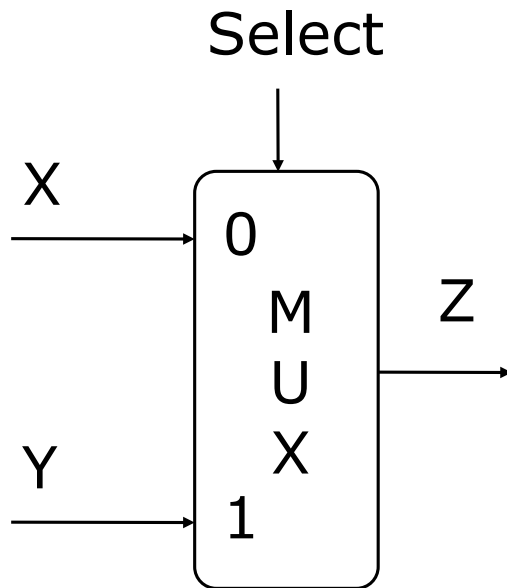
- Function: Converts machine instructions into control signals, e.g. MemRead.
- Input: Machine instruction.
- Output: Control signals.



Control

Multiplexer

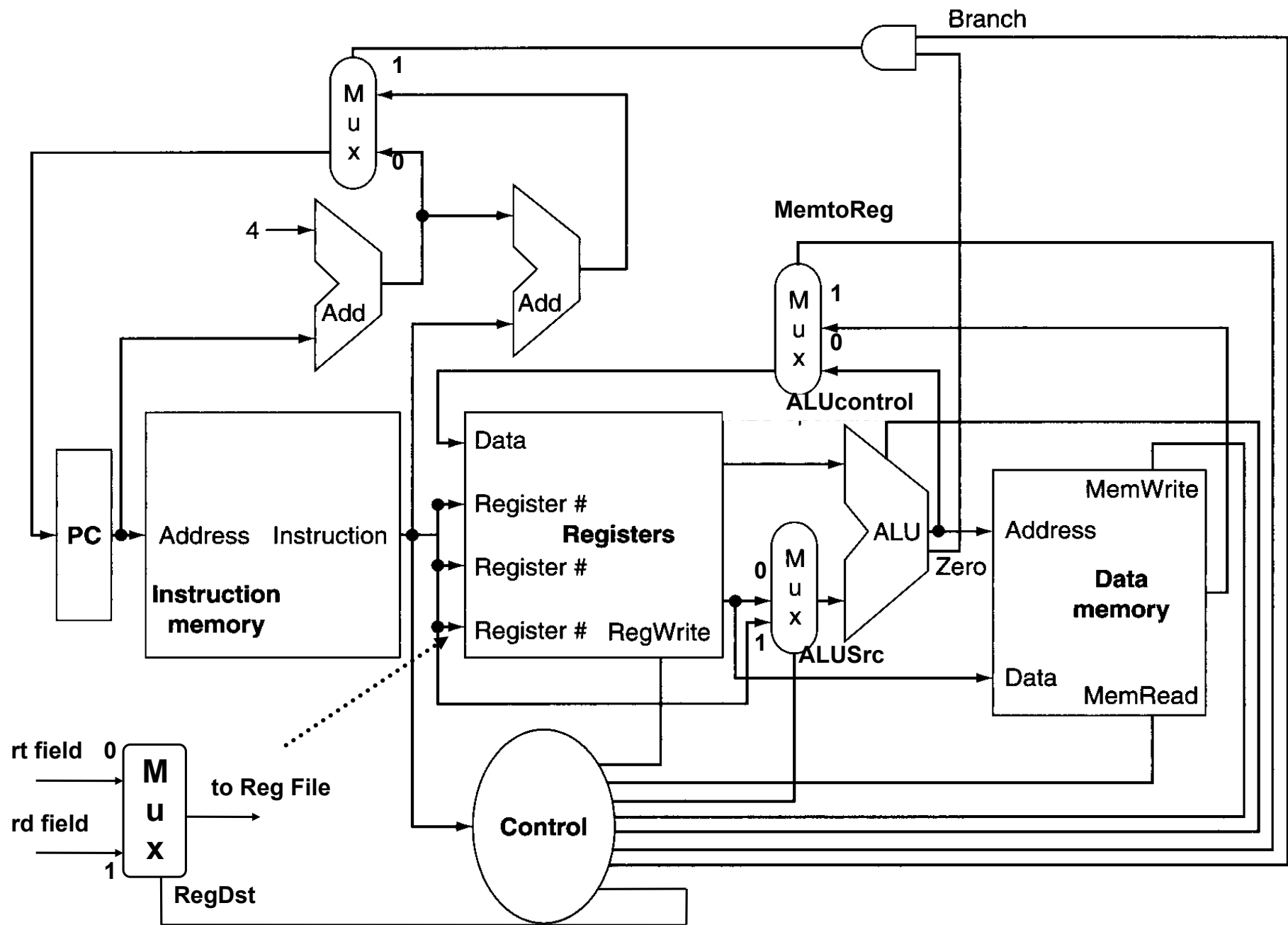
- Used to select between various processing options.



Symbol

Select	Z
0	X
1	Y

Function



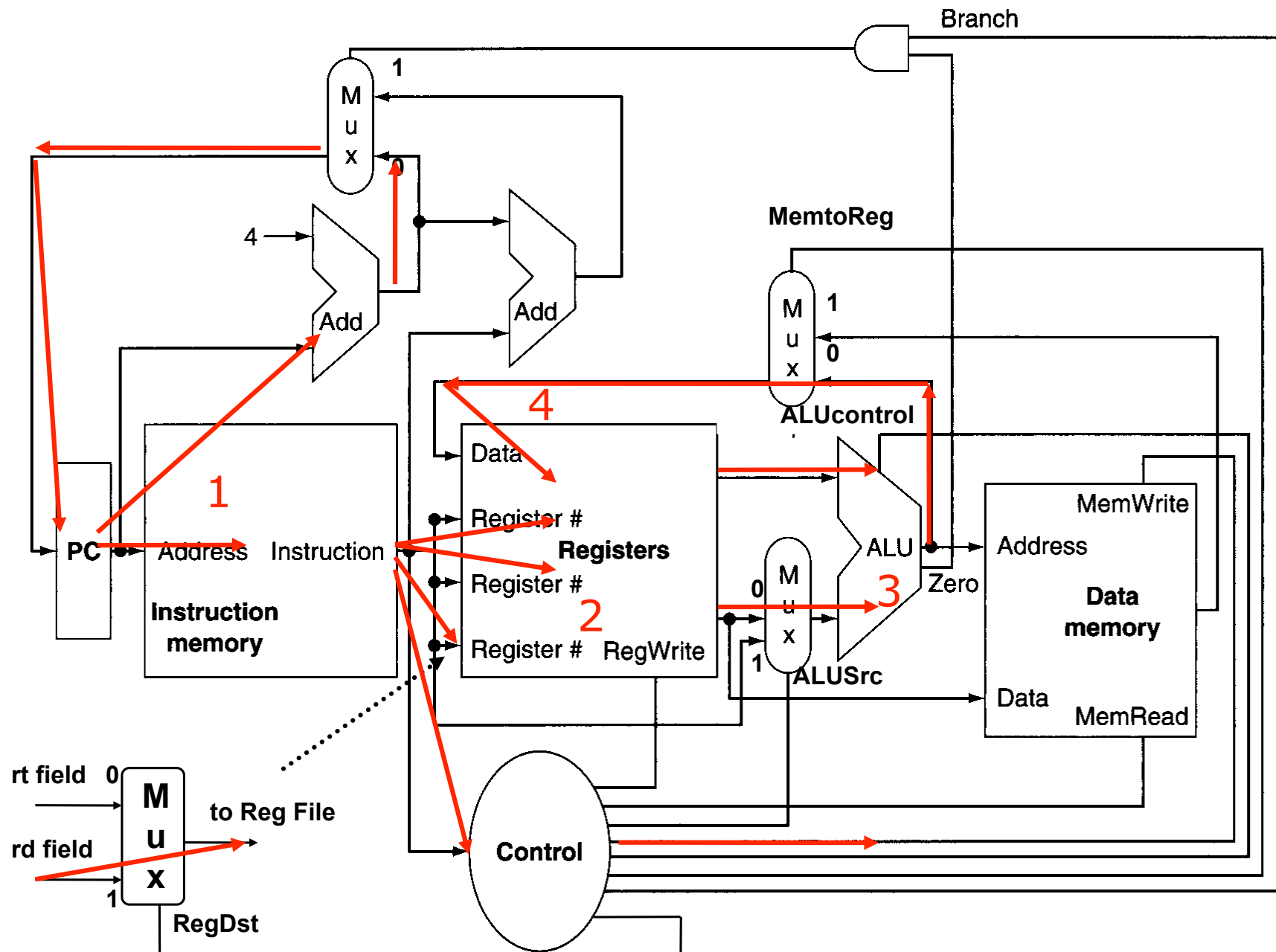
Instruction

`add $t1, $t2, $t3`

op [31:26]	rs [25:21]	rt [20:16]	rd [15:11]	shamt [10:6]	funct [5:0]
000000	10010	10011	10001	00000	100000

Instruction

add \$t1, \$t2, \$t3



Instruction

```
add $t1, $t2, $t3
```

1. Instruction fetched, PC+4 is calculated.
2. \$t2 and \$t3 read from register file. The Control Unit sets up the control signals.
3. ALU operates on data from register file according to control signals (adds).
4. Result from ALU written back to register file.
5. On the positive edge of clock, the \$t1 register and the PC are updated.

Instruction

```
add $t1, $t2, $t3
```

- RegDst = 1
- ALUSrc = 0
- ALUControl = 0010 (add)
- MemtoReg = 0
- RegWrite = 1
- MemRead = 0
- MemWrite = 0
- Branch = 0

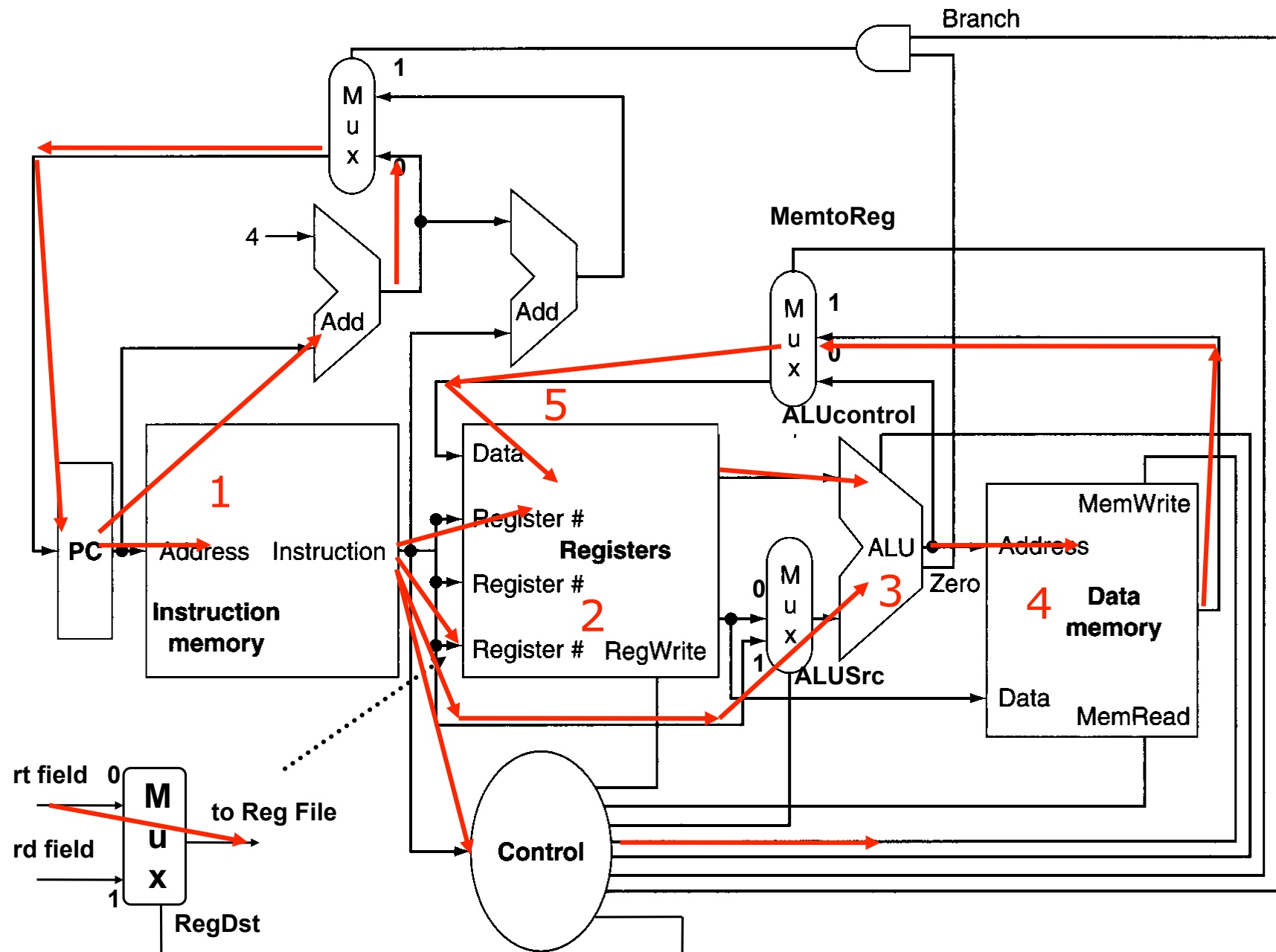
Instruction

```
lw $t1, 32($t2)
```

op [31:26]	rs [25:21]	rt [20:16]	offset [15:0]
100011	10010	10001	0000_0000_0001_0000

Instruction

lw \$t1, 32(\$t2)



Instruction

```
lw $t1, 32($t2)
```

1. Instruction fetched, PC+4 is calculated.
2. \$t2 is read from the register file. The Control Unit sets up the control signals.
3. ALU computes the sum of the register file output and the sign extended offset from the 16 LSBs from the instruction.
4. The output of the ALU is used to address data memory.
5. The data from memory is written back to the register file.
6. On the positive edge of clock, the \$t1 register and the PC are updated.

Instruction

```
lw $t1, offset($t2)
```

- RegDst = 0
- ALUSrc = 1
- ALUControl = 0010 (add)
- MemtoReg = 1
- RegWrite = 1
- MemRead = 1
- MemWrite = 0
- Branch = 0

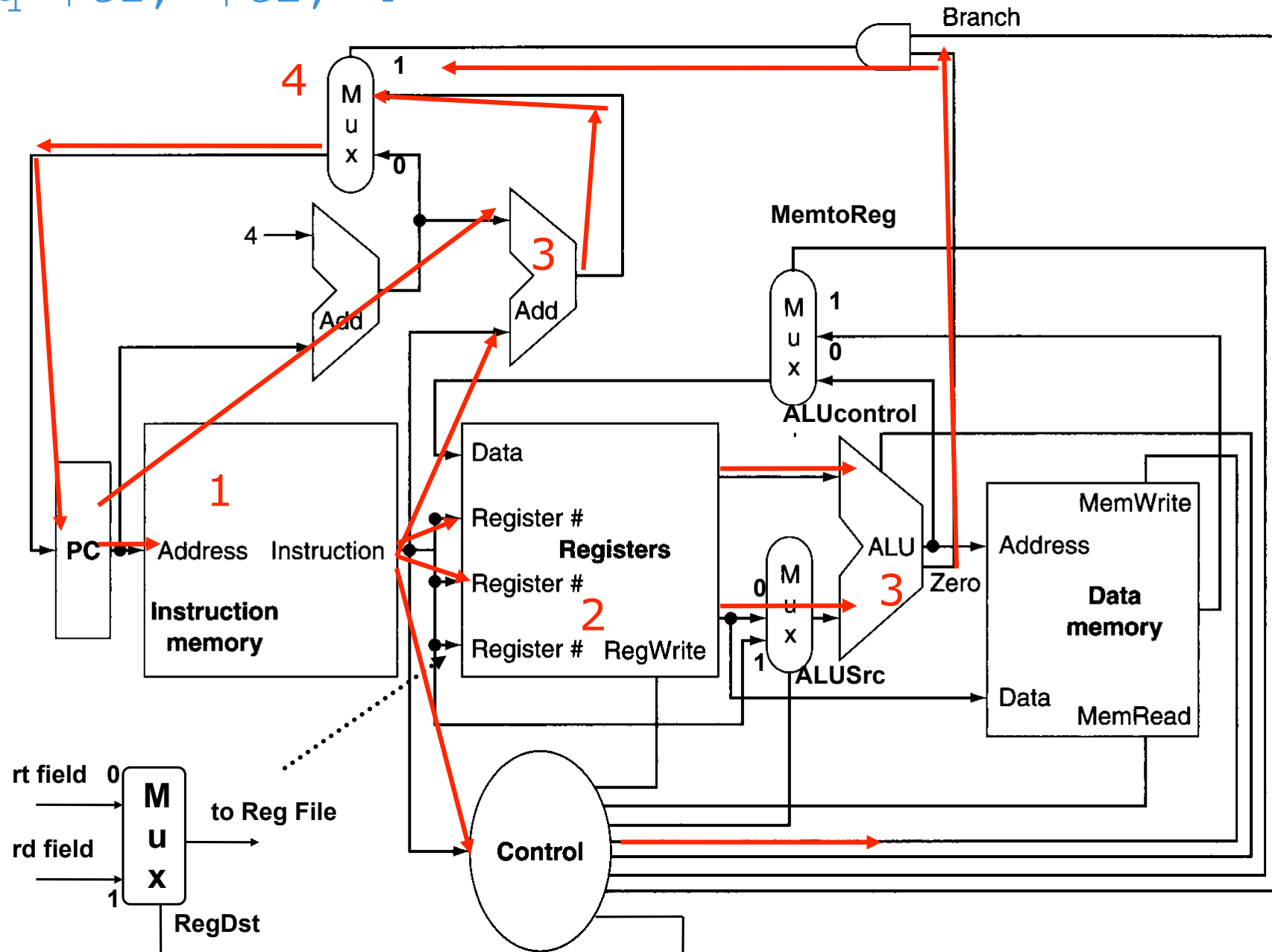
Instruction

beq \$t1, \$t2, 4

op [31:26]	rs [25:21]	rt [20:16]	offset [15:0]
000100	10001	10010	0000_0000_0000_0100

Instruction

beq \$t1, \$t2, 4



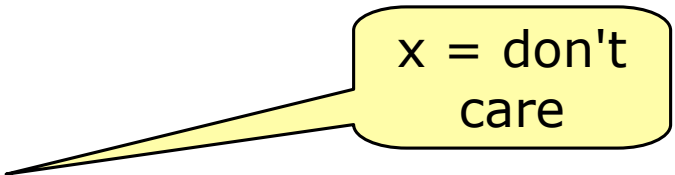
Instruction

```
beq $t1, $t2, 4
```

1. Instruction fetched, PC+4 is calculated.
2. \$t1 and \$t2 are read from the register file. The Control Unit sets up the control signals.
3. ALU subtracts one register file output from the other. PC+4 is added to the left shifted by 2 and sign extended offset from the instruction.
4. The Zero output from the ALU is used to select between the two branch destinations.
5. On the positive edge of clock, the PC is updated.

Instruction

`beq $t1, $t2, 4`



x = don't
care

- RegDst = x
- ALUSrc = 0
- ALUControl = 0110 (subtract)
- MemtoReg = x
- RegWrite = 0
- MemRead = 0
- MemWrite = 0
- Branch = 1

Control

- We now know how data flows and what control signal settings are needed.
- How does the Control Unit convert machine instructions into the appropriate control signal settings?

Control Input

R-format

op	rs	rt	rd	shamt	funct
[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]

op = 00_{hex}

Bit range

I-format

op	rs	rt	constant or address
[31:26]	[25:21]	[20:16]	[15:0]

lw: op = 23_{hex}

sw: op = 28_{hex}

beq: op = 04_{hex}

J-format

op	address
[31:26]	[25:0]

op = 02_{hex}

Control

Direct Connections

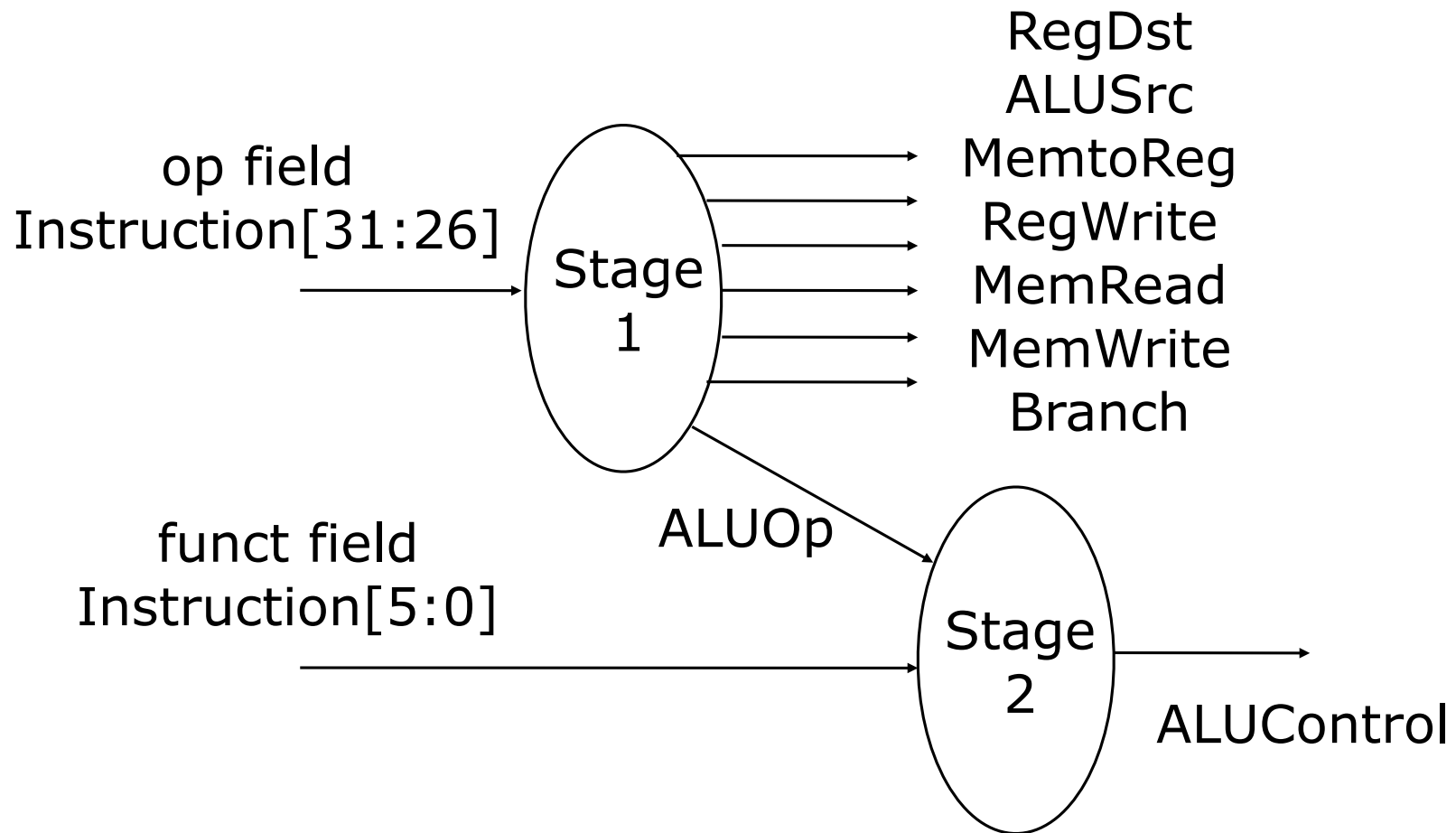
- Register File inputs:
 - ReadReg1 = rs field (Instruction[25:21])
 - ReadReg2 = rt field (Instruction[20:16])
 - WriteReg = instruction rd field (e.g. `add`) (Instruction[15:11]) OR memory data (e.g. `lw`).
- Branch target address connected to address field (Instruction [15:0]).
- Address offset connected to address field (Instruction[15:0]).

Control Output

Control signal name	Effect when deasserted (=0)	Effect when asserted (=1)
RegDst	Write register number comes from rt field (lw, sw instructions)	Write register number comes from rd field (R-format instructions)
RegWrite	None	Writes register file data input to register
ALUSrc	2 nd ALU operand comes from register file	2 nd ALU operand comes from const field
MemRead	None	Reads data from memory to memory output
MemWrite	None	Writes data memory input to memory
MemtoReg	Register file write data input comes from ALU	Register file data inputs comes from data memory
Branch	None	Branch if Zero=1

Control Decode

- Control Unit uses two stage decode.



Control

Decode Stage 1

Control Unit
internal signal

Instr. / Opcode	Reg Dst	ALU Src	Memto Reg	Reg Write	Mem Read	Mem Write	Branch	ALU Op
R-format 00 _{hex}	1	0	0	1	0	0	0	10 funct
lw 23 _{hex}	0	1	1	1	1	0	0	00 add
sw 28 _{hex}	x	1	X	0	0	1	0	00 add
beq 04 _{hex}	x	0	x	0	0	0	1	01 sub

Control

Decode Stage 2

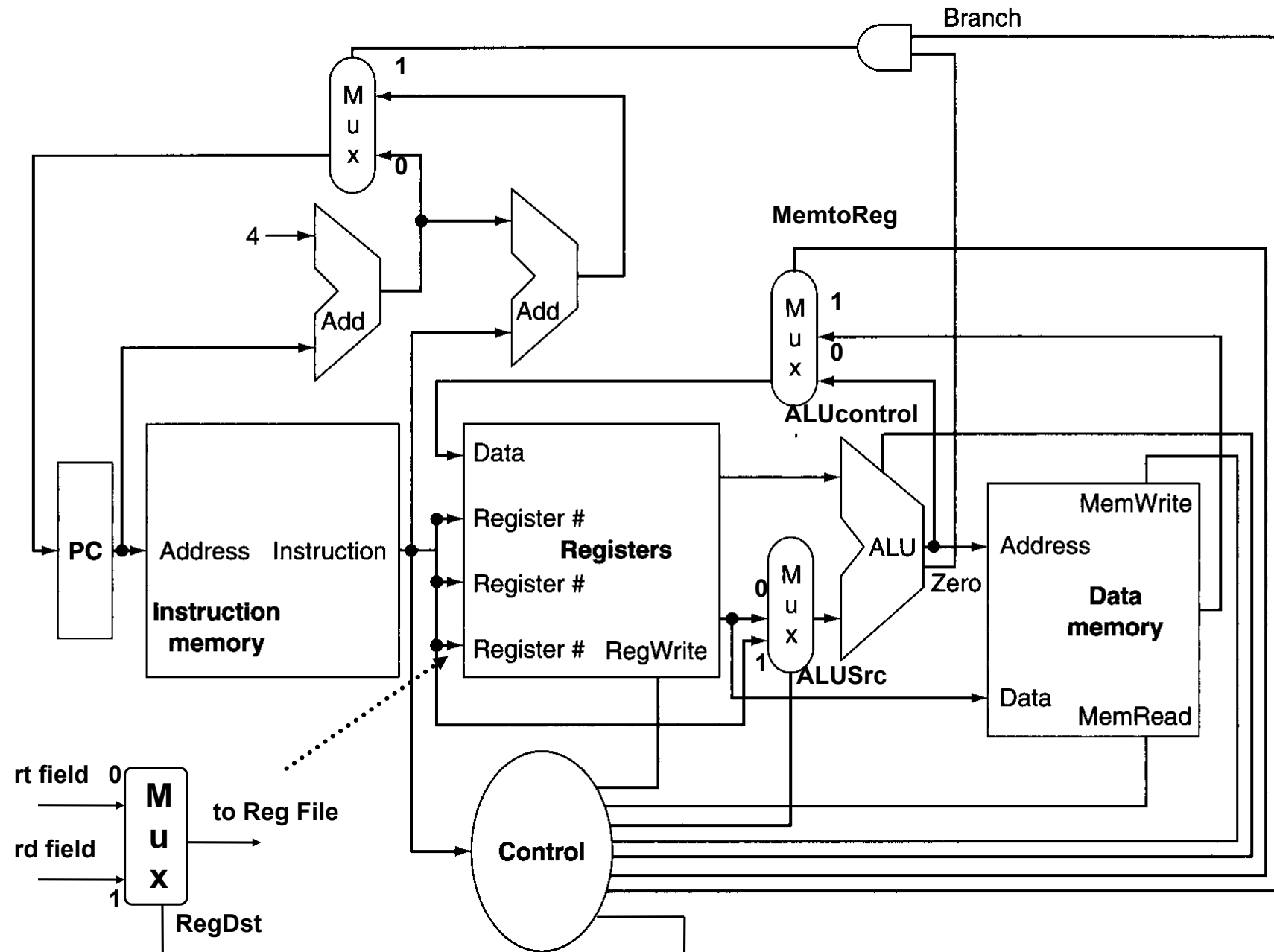
Instruction	ALUop	funct	ALU action	ALU Control
lw	00	xxxxxxx	add	0010
sw	00	xxxxxxx	add	0010
beq	01	xxxxxxx	subtract	0110
add	10	100000	add	0010
sub	10	100010	subtract	0110
and	10	100100	AND	0000
or	10	100101	OR	0001
slt	10	101010	set on less than	0111

Control

Decode

- Tables above are Truth Tables.
- Need combinational logic (gates) to perform decoding.
- Use Karnaugh maps to provide minimum area solution.

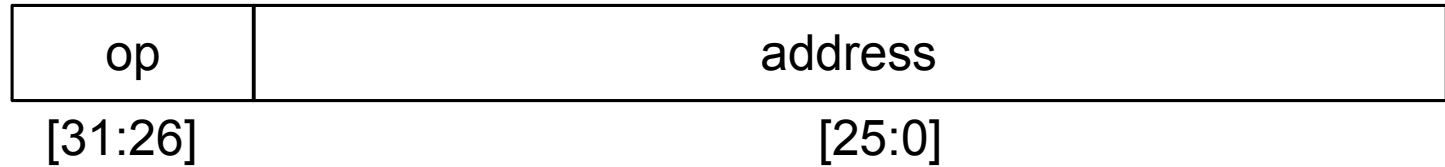
Control



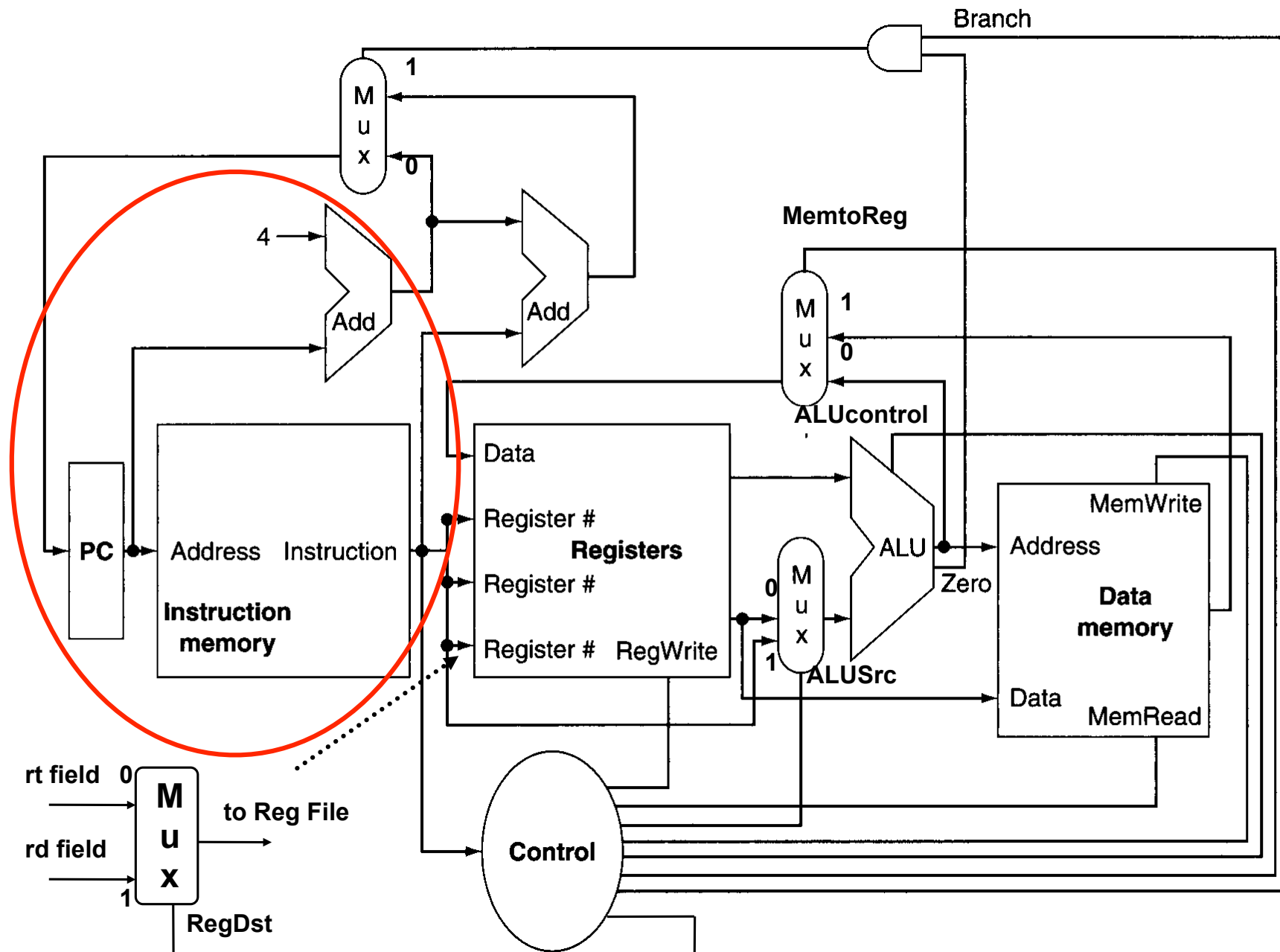
Pop Quiz

- How is the jump instruction added?
 - op = 000010_{two}

J-format



Gate-Level

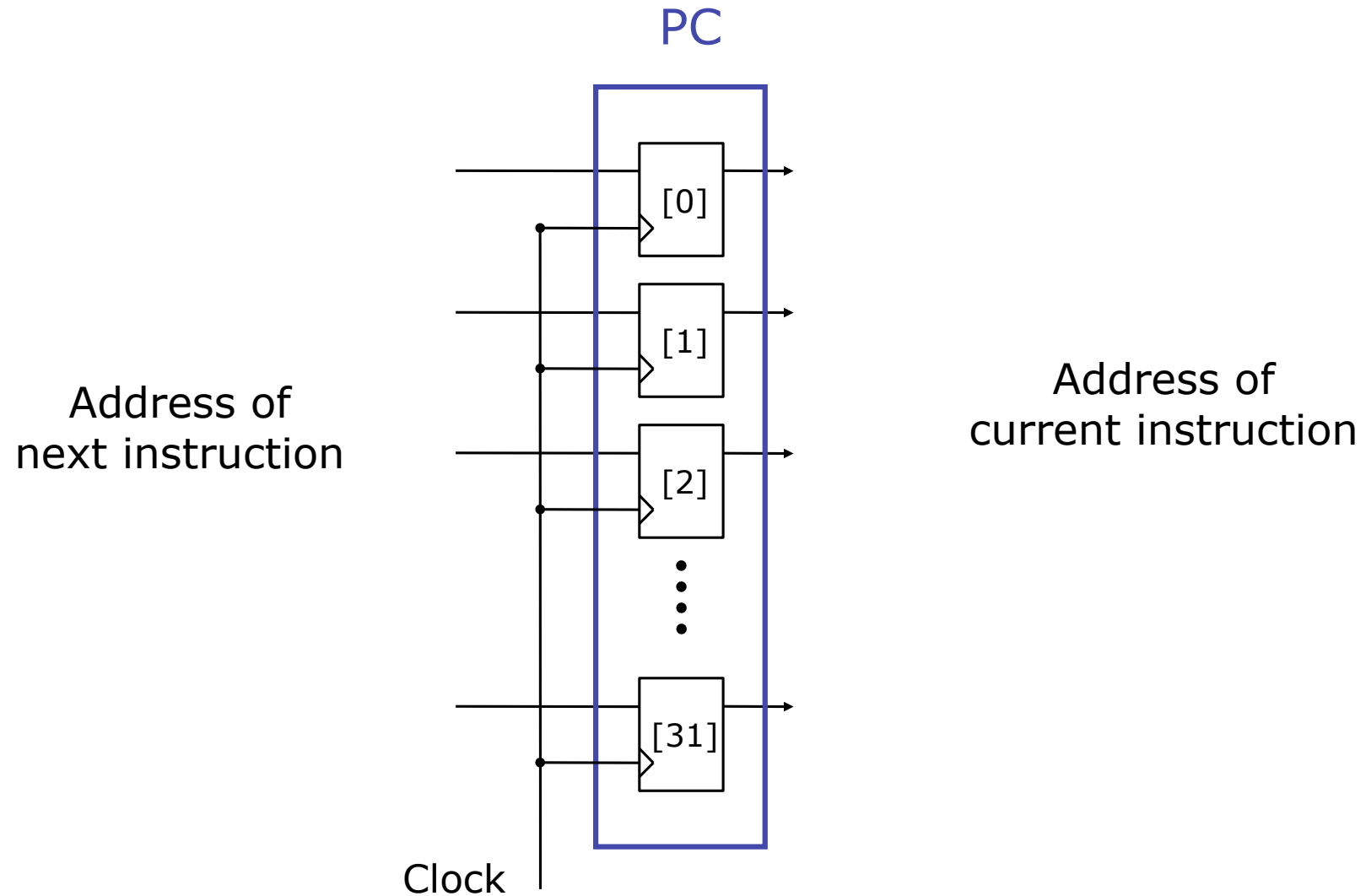


3. Single cycle MIPS Processor

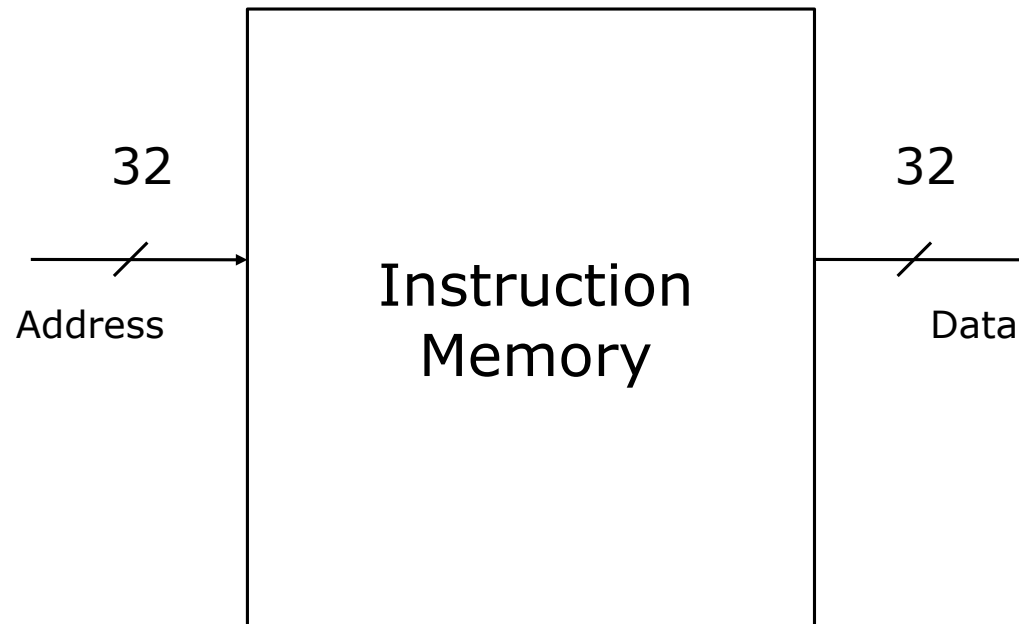
Program Counter

- PC register
 - Function: Holds address of the current instruction. Loads input into register and on to output on the positive edge of the clock.
 - Input: Address of next instruction
 - Output: Address of current instruction
- Instruction memory
 - Function: Stores machine instructions.
 - Input: Instruction address
 - Output: Machine instruction contained stored at that address
- Adder
 - Function: Adds two numbers
 - Input: Two numbers
 - Output: Total of the numbers

Program Counter



Program Counter



Read Only Memory (ROM)
Assume instructions have been pre-loaded

Half Adder

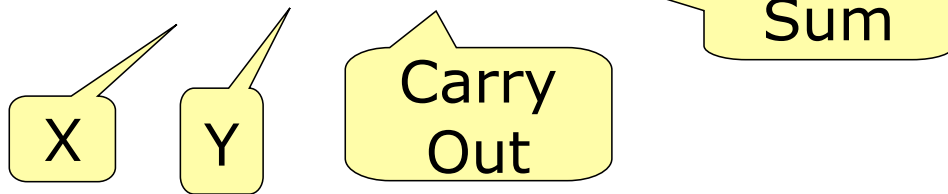
- Consider adding 2 bits:

- $0 + 0 = 00$

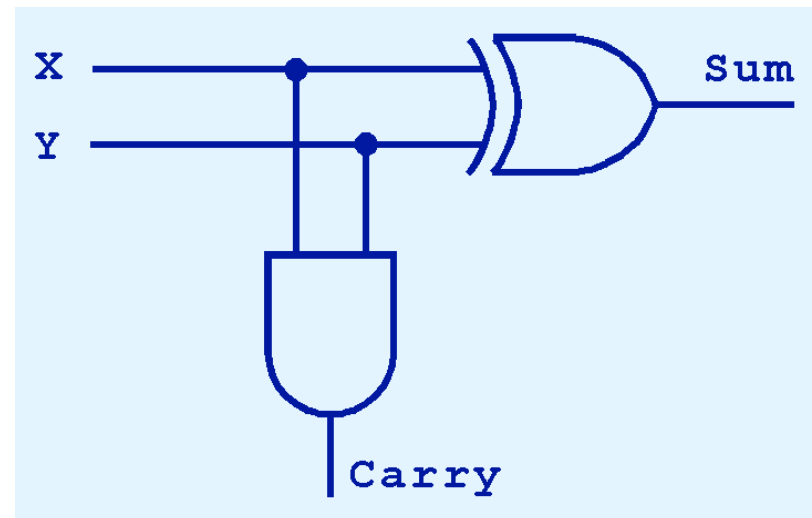
- $0 + 1 = 01$

- $1 + 0 = 01$

- $1 + 1 = 10$



X	Y	CO	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



Full Adder

- Consider add 2 bit PLUS the carry out from the previous column

- $0 + 0 + 0 = 00$

- $0 + 1 + 0 = 01$

- $1 + 0 + 0 = 01$

- $1 + 1 + 0 = 10$

- $0 + 0 + 1 = 01$

- $0 + 1 + 1 = 10$

- $1 + 0 + 1 = 10$

- $1 + 1 + 1 = 11$



Carry In

The diagram illustrates the logic of a full adder by listing all possible combinations of two input bits and a carry-in bit. Each combination is shown as a bullet point with its corresponding sum in binary. Two yellow callout boxes are present: 'Carry In' points to the third bit in each sum (the carry-in), and 'Carry Out' points to the first bit of the sum (the carry-out). For example, in the last row, '1 + 1 + 1 = 11', the 'Carry In' is the third '1' and the 'Carry Out' is the first '1' of the result '11'.

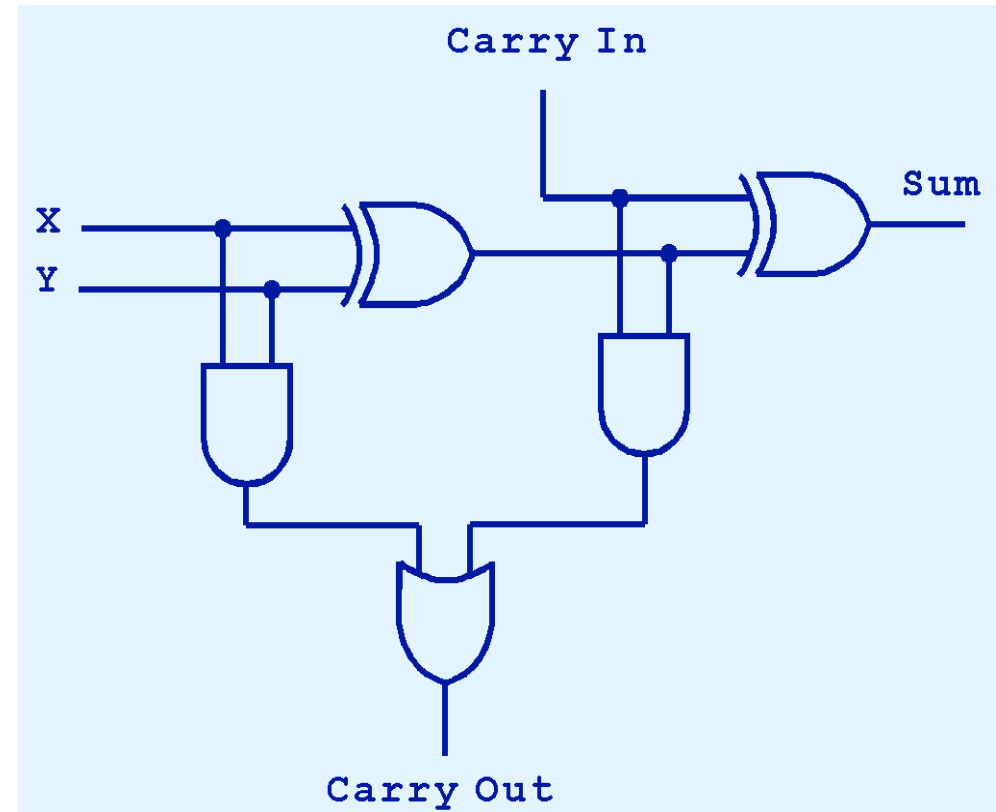
Carry Out

Full Adder

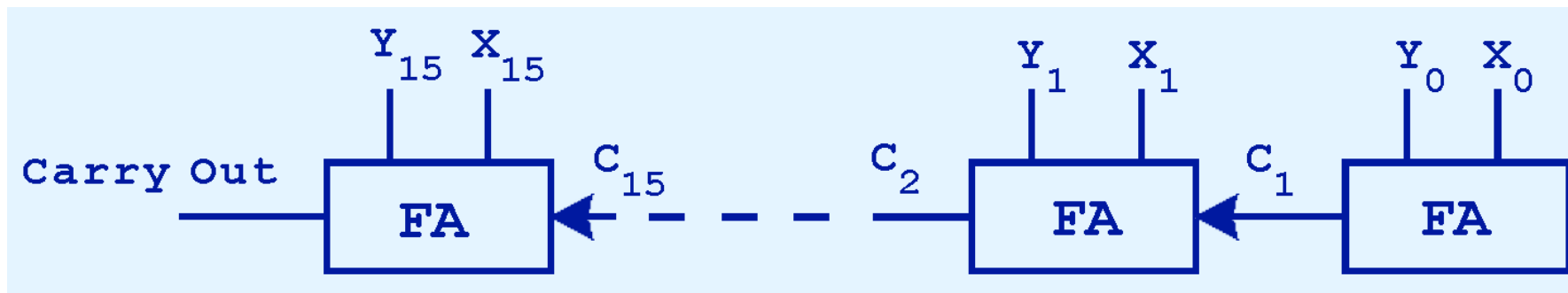
X	Y	CI	CO	S
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

Full Adder

X	Y	CI	X XOR Y = T	S	A AND Y	CI AND T	CO
0	0	0	0	0	0	0	0
0	1	0	1	1	0	0	0
1	0	0	1	1	0	0	0
1	1	0	0	0	1	0	1
0	0	1	0	1	0	0	0
0	1	1	1	0	0	1	1
1	0	1	1	0	0	1	1
1	1	1	0	1	1	0	1

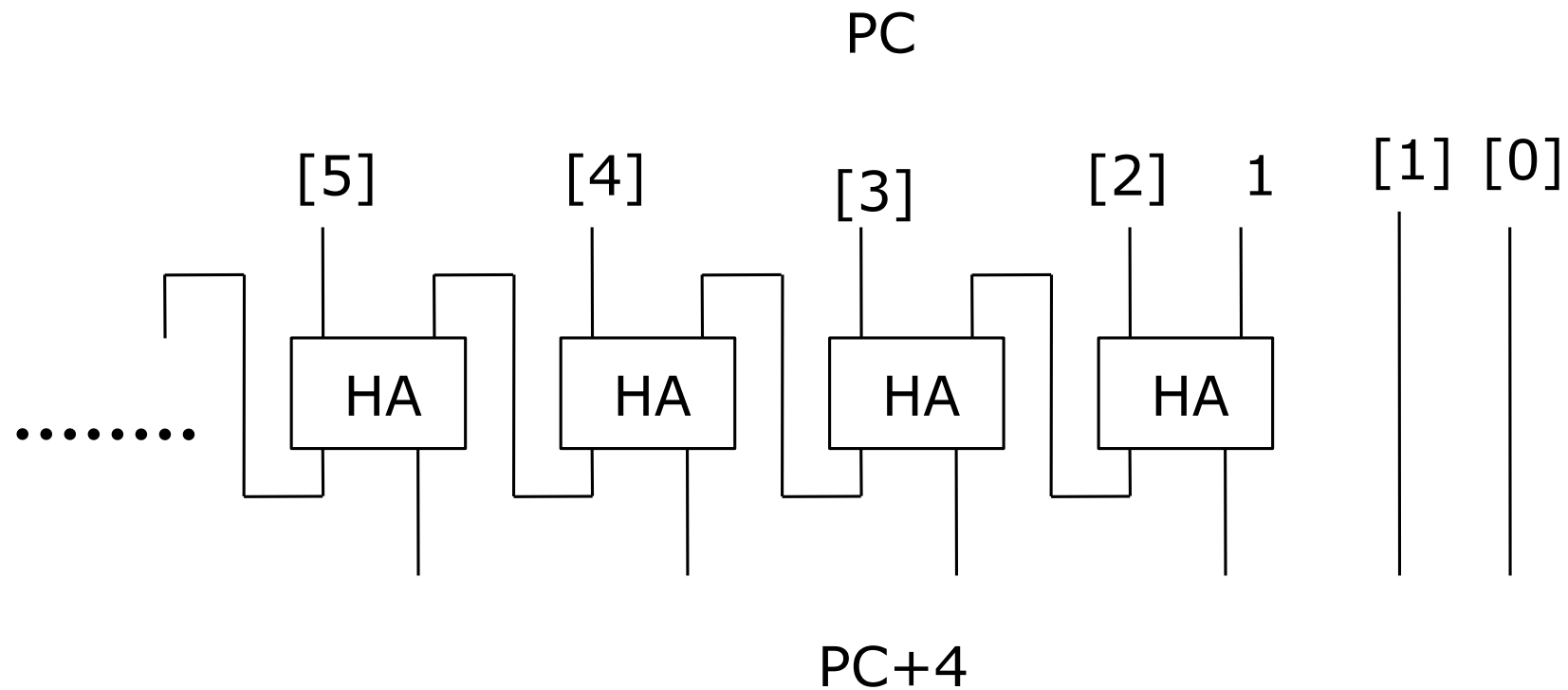


Adder

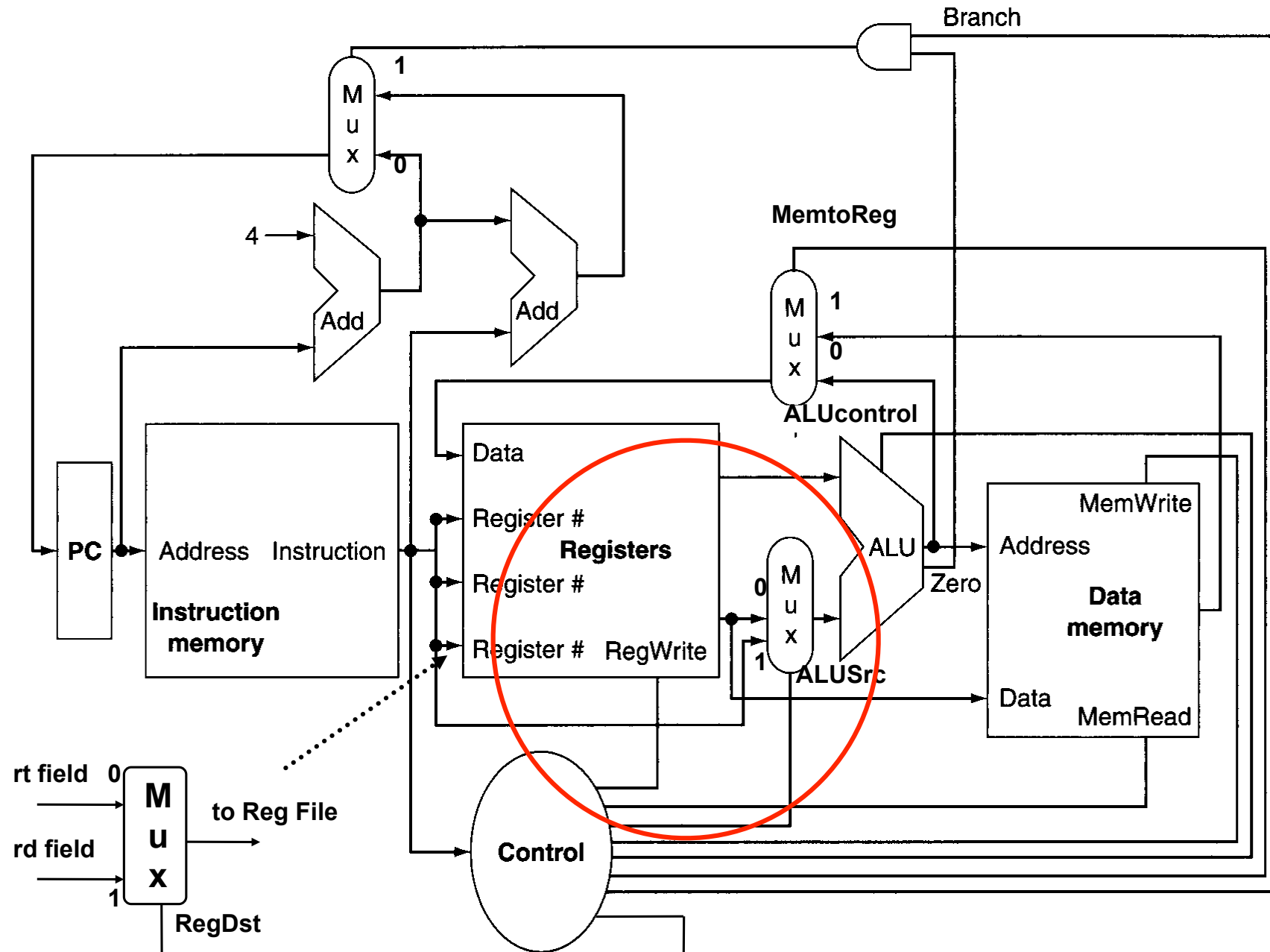


In this case, X input is wired to the output of the PC register and Y is wired to binary 4 = 000100.

Adder

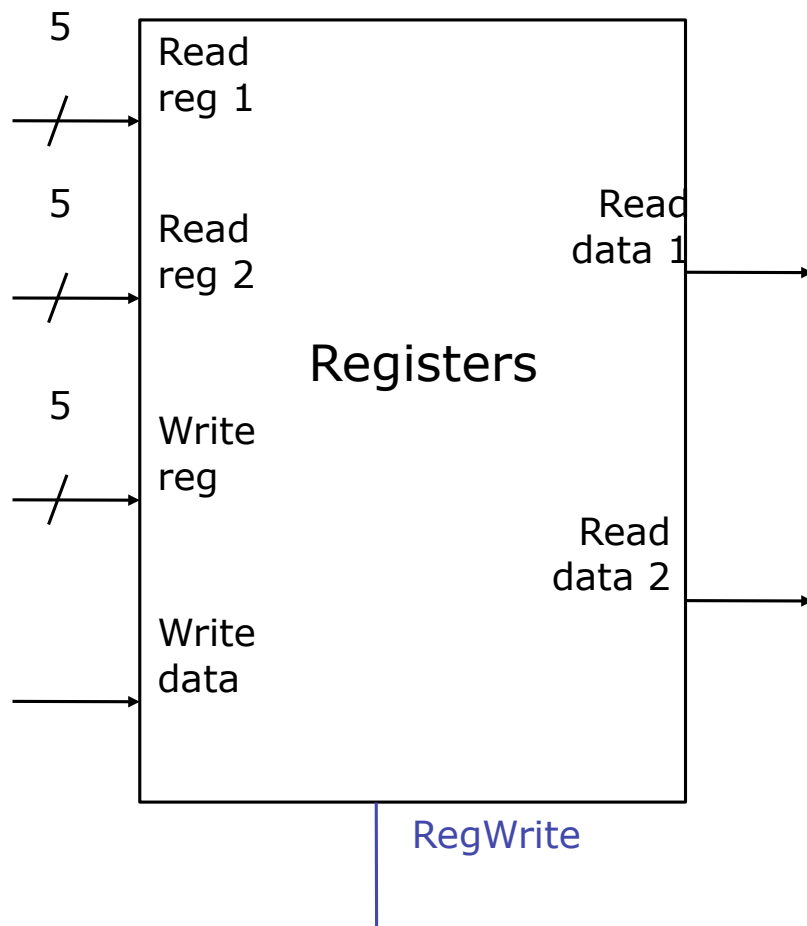


Because 4 is a fixed constant, the adder can be simplified.



3. Single cycle MIPS Processor

Register File



Function:

- Stores register values.
- Can read 2 registers at a time.
- Can write 1 register at a time.

Inputs:

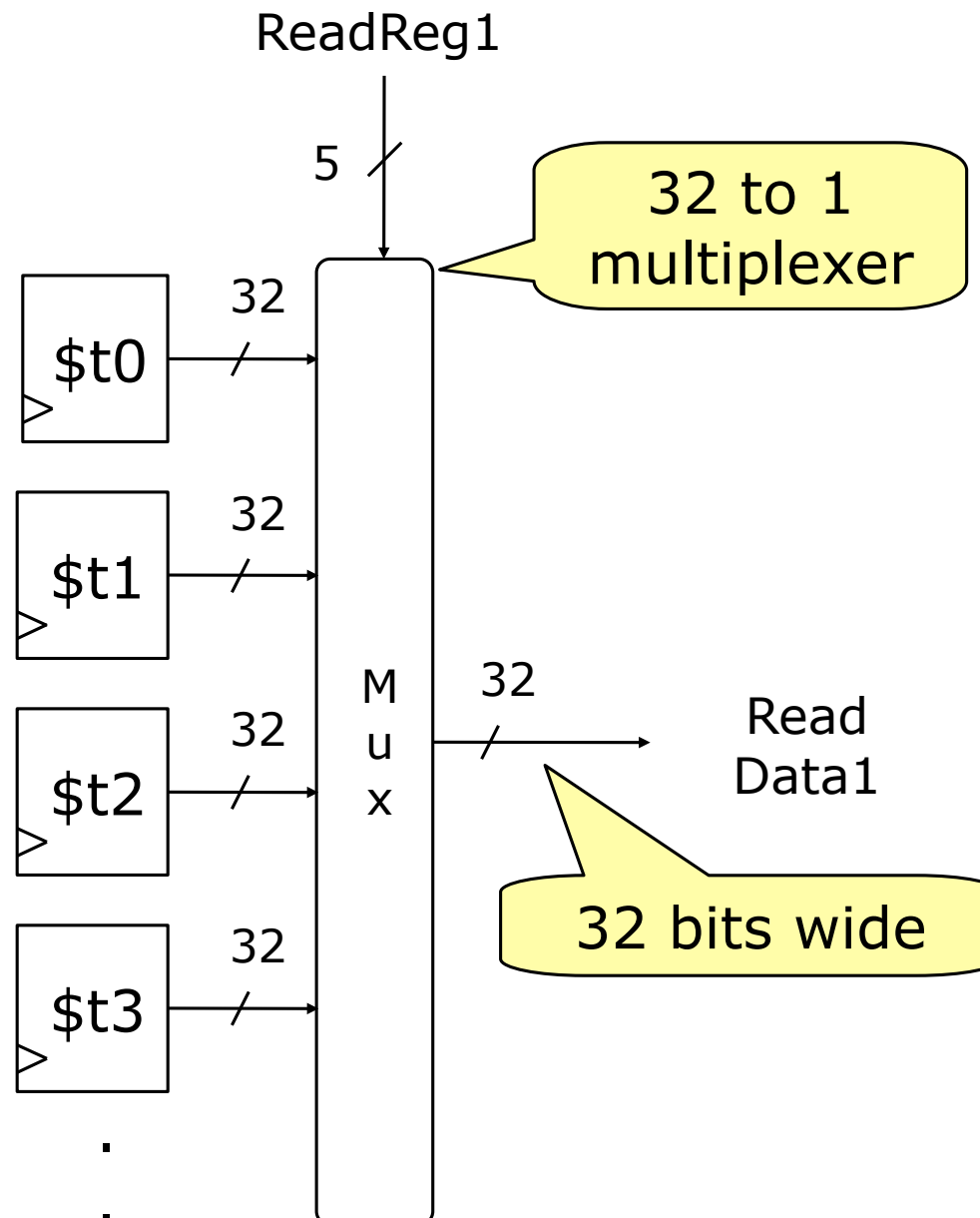
- Number of registers to read
- Number of register to write
- Write control signal (1=write, 0=no write)

Outputs:

- * Values in read registers.

Register File

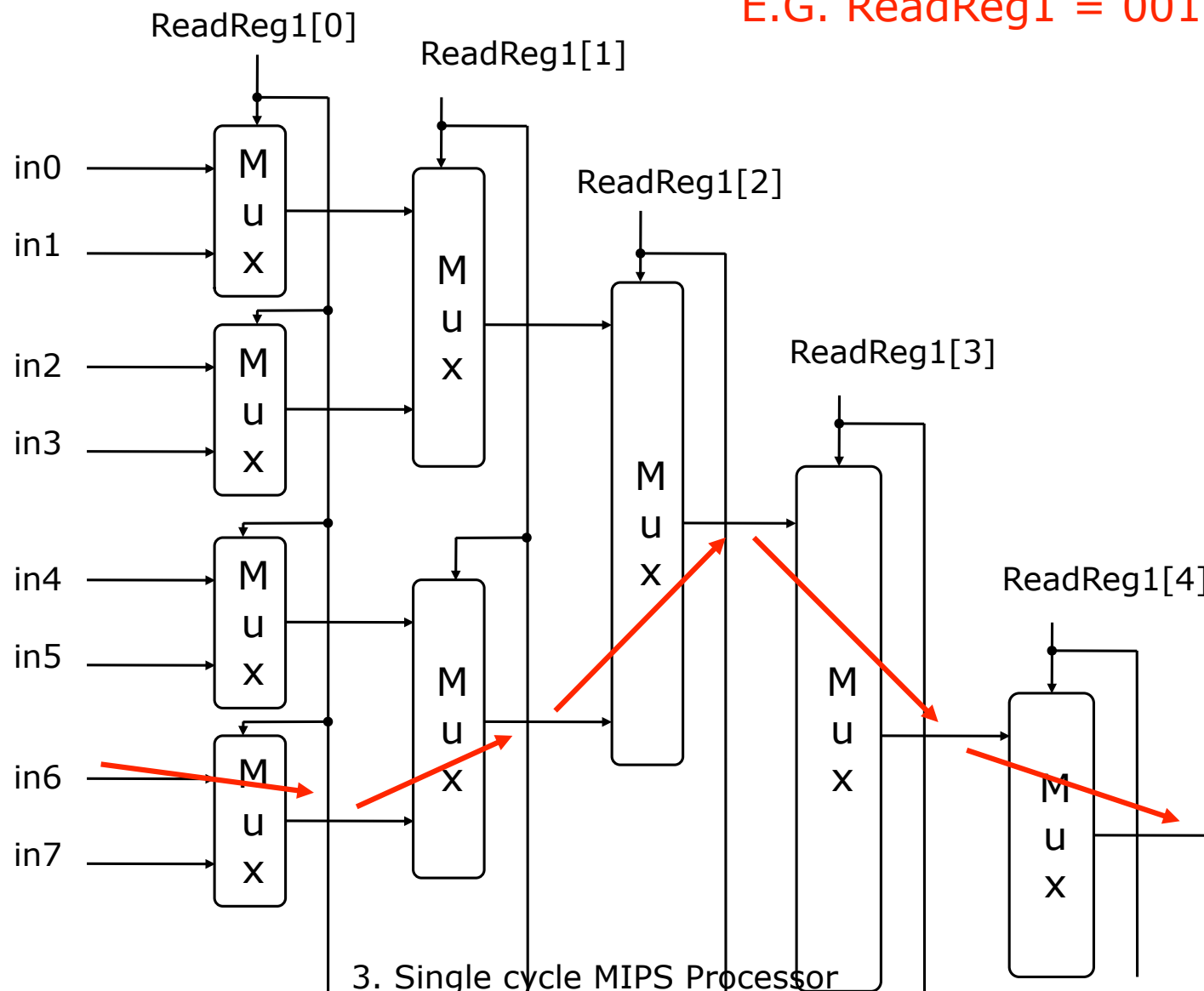
Read Logic



Register File

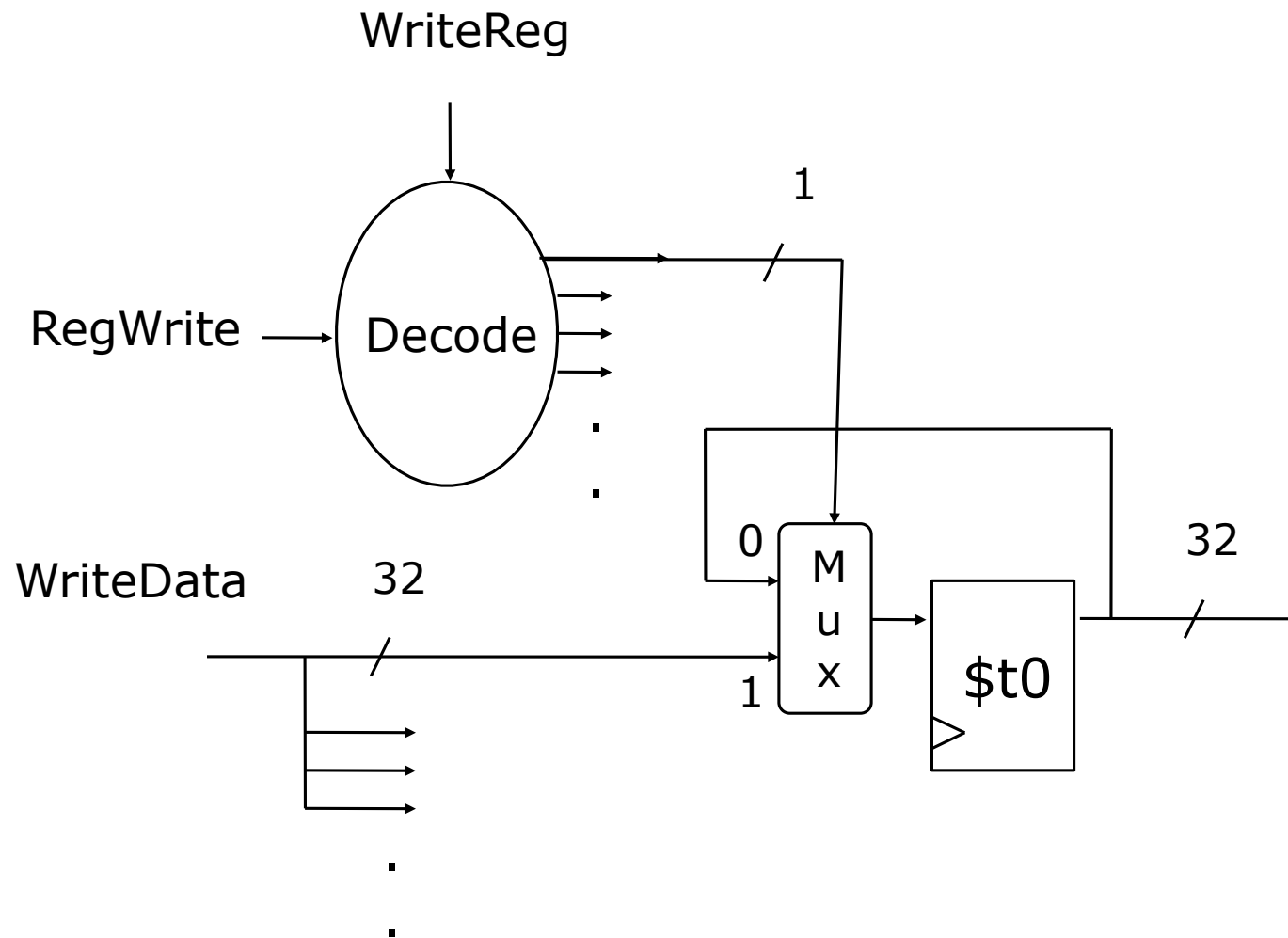
32 to 1 Multiplexer

E.G. $\text{ReadReg1} = 00110_2 = 6_{10}$



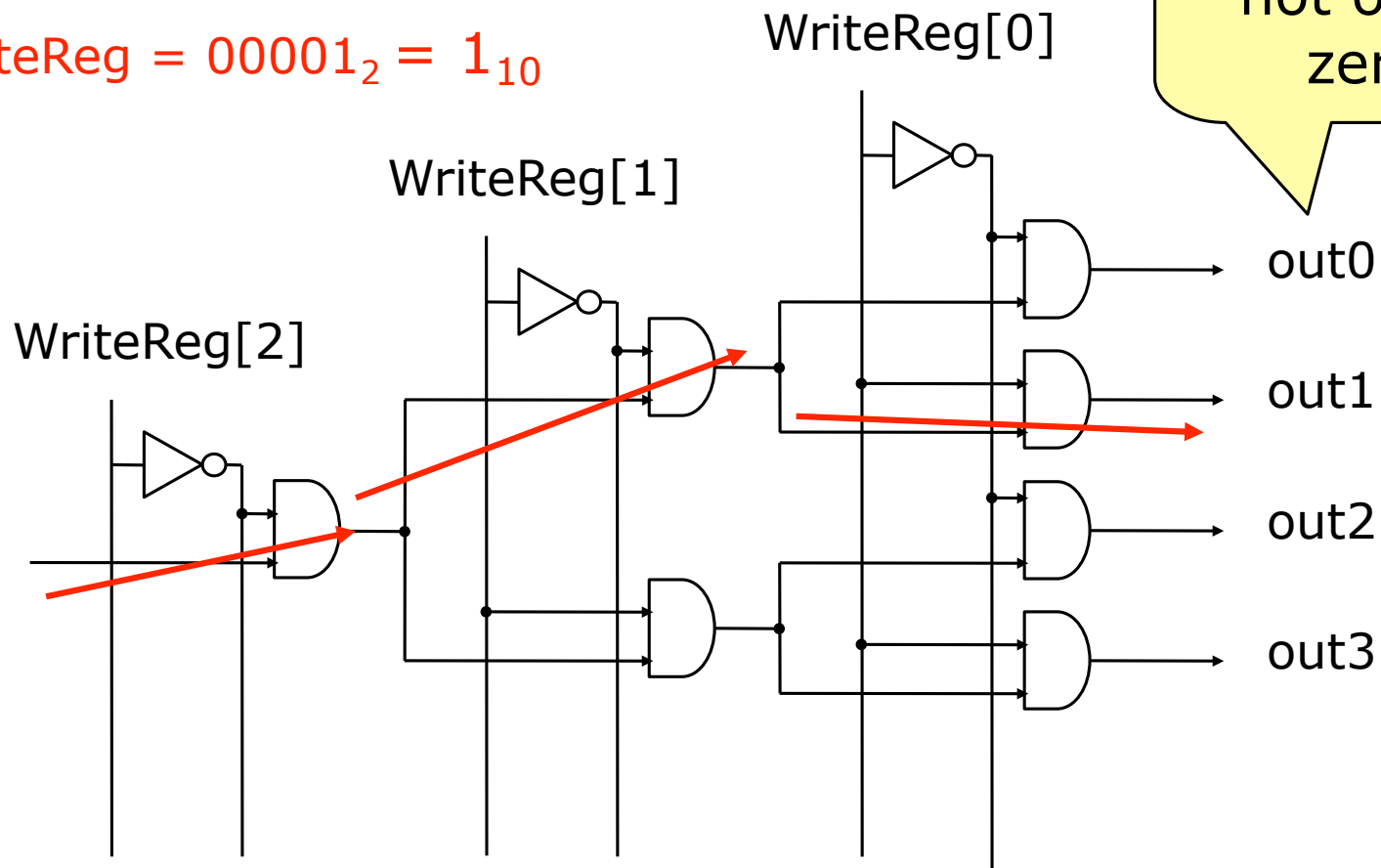
Register File

Write Logic



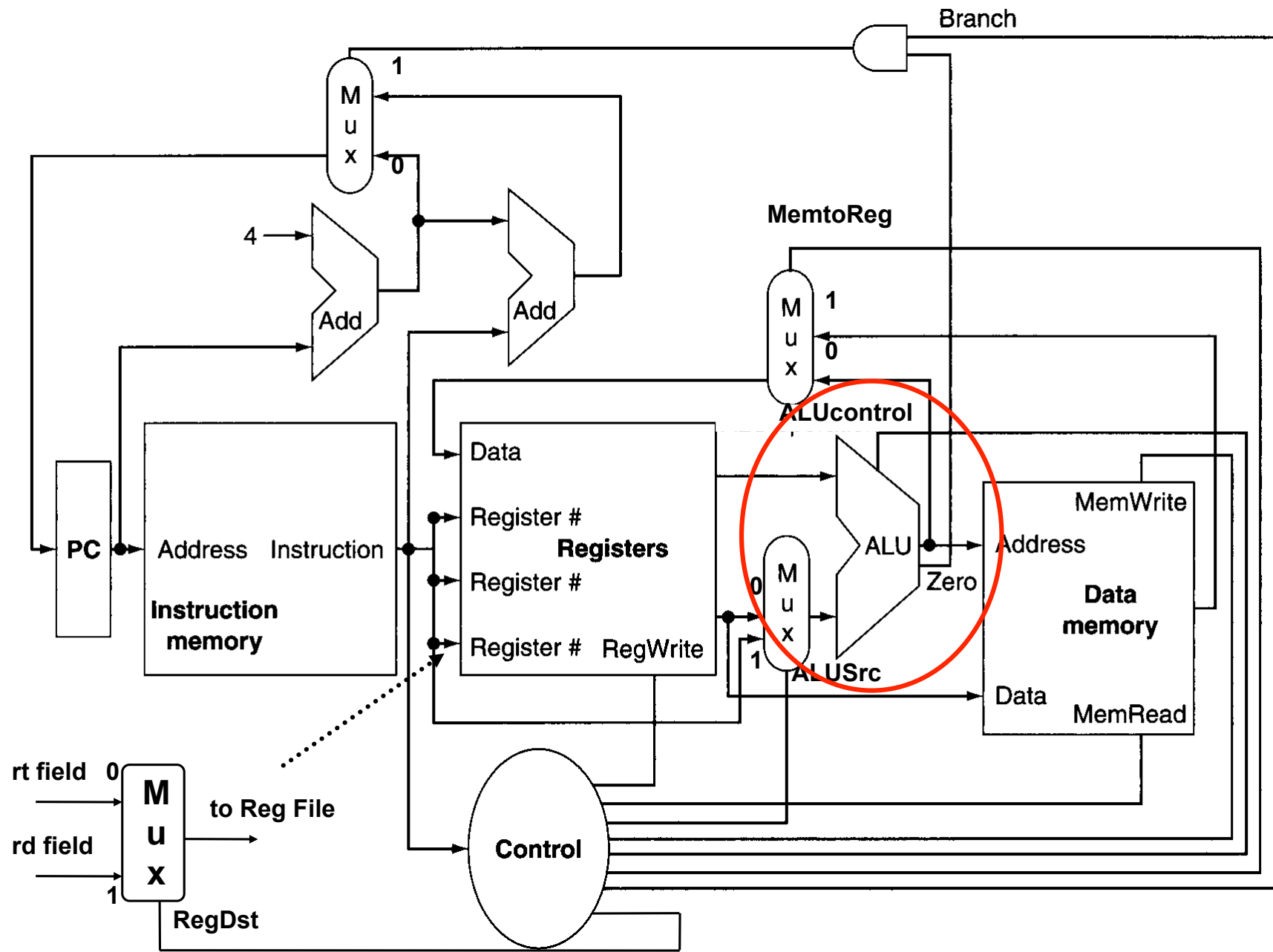
Register File Decoder

E.G. WriteReg = $00001_2 = 1_{10}$

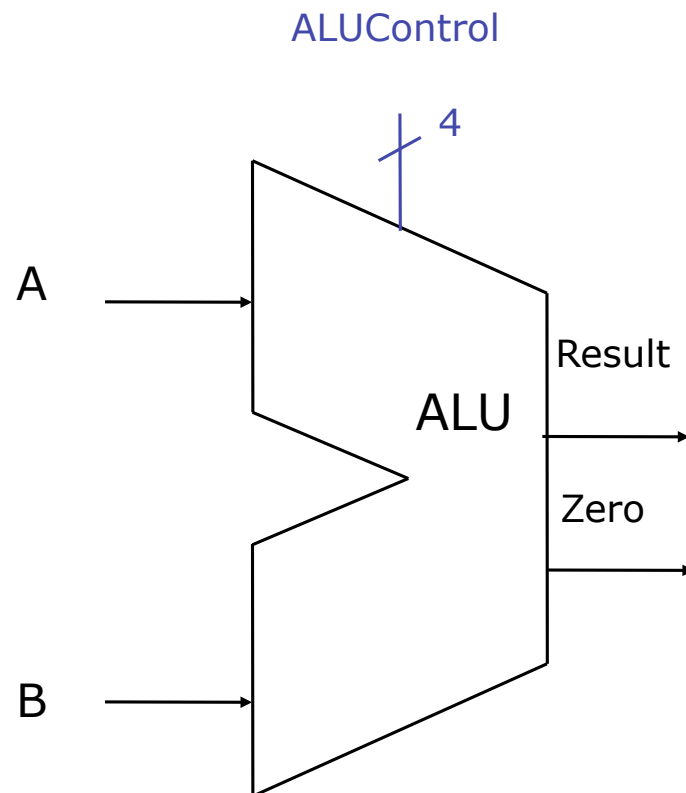


Either one-hot or all zero

...



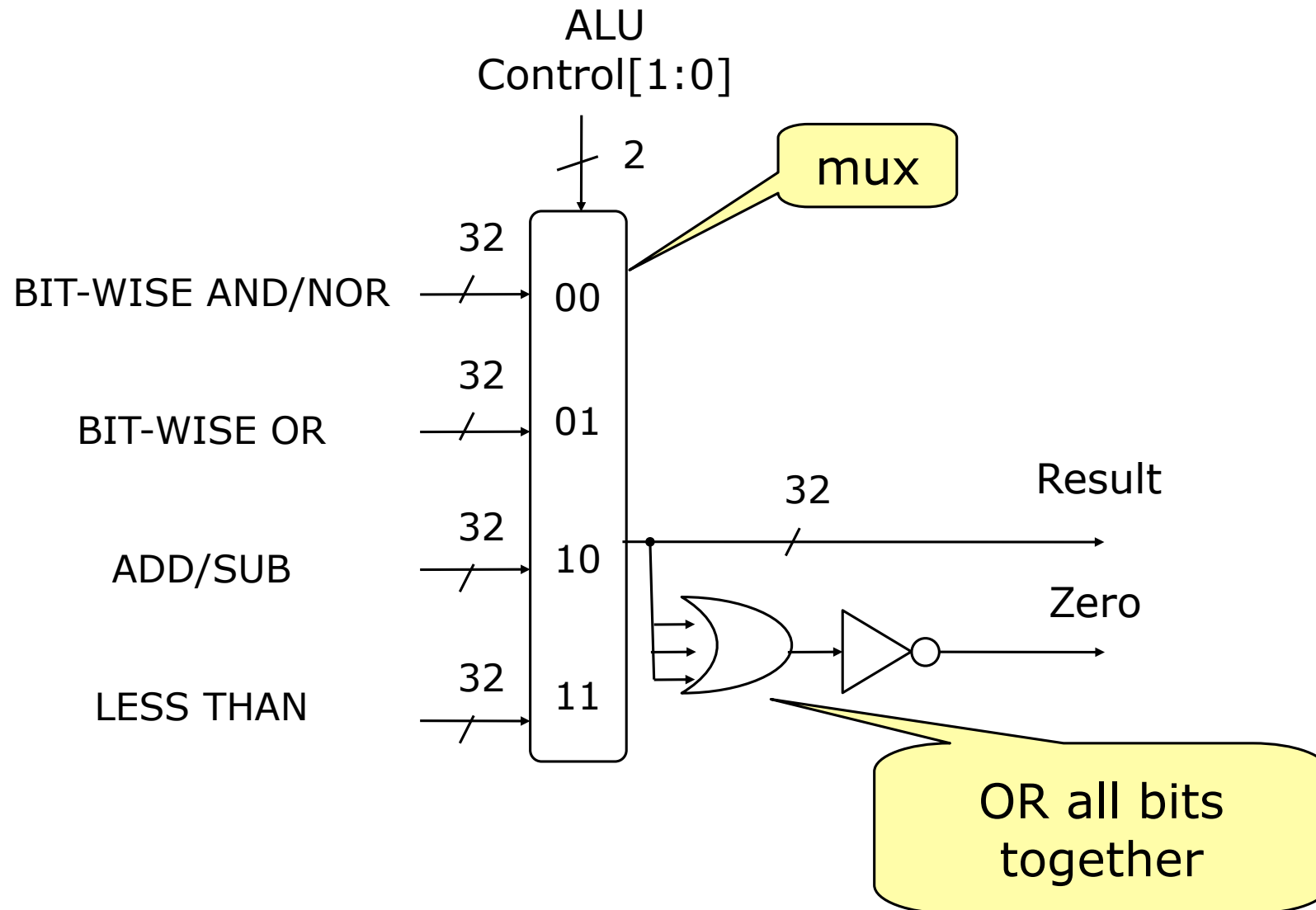
ALU



ALU Control	Function
0000	AND
0001	OR
0010	Add
0110	Subtract
0111	Set on less than
1100	NOR

ALU

Output



ALU

ADD/SUB

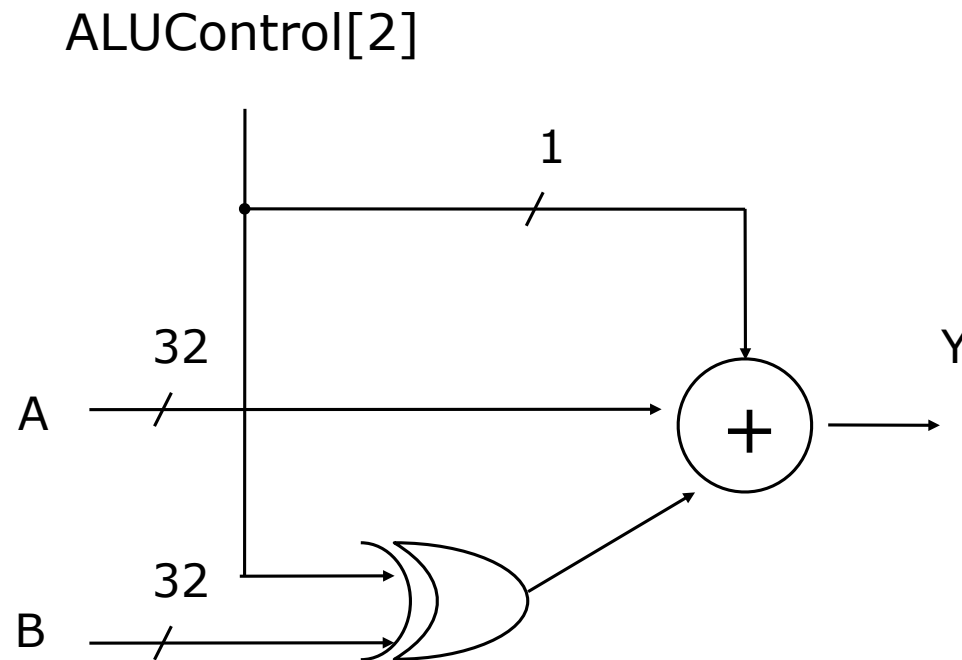
- ADD: $Y = A + B$
- SUB: $Y = A - B = A + (\text{NOT } B) + 1$
- $B = B \text{ XOR } 0$
- $\text{NOT } B = B \text{ XOR } 1$
- ADD: $Y = A + (B \text{ XOR } 0) + 0$
- SUB: $Y = A + (B \text{ XOR } 1) + 1$

ALU

ADD/SUB

ADD: ALUControl[2] = 0

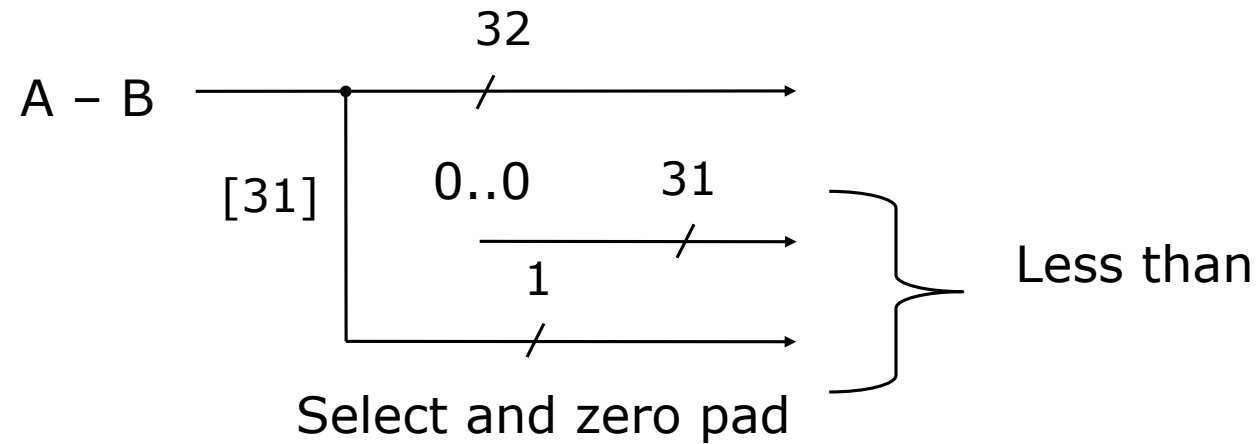
SUB: ALUControl[2] = 1



ALU

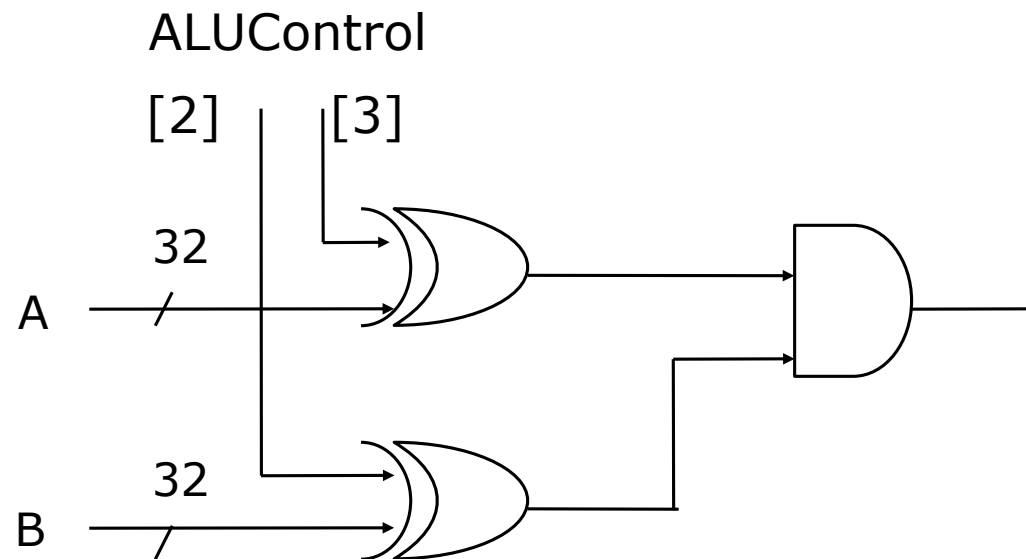
LESS THAN

- $A < B$
- $A - B < 0$
- $\text{SIGN} (A - B)$

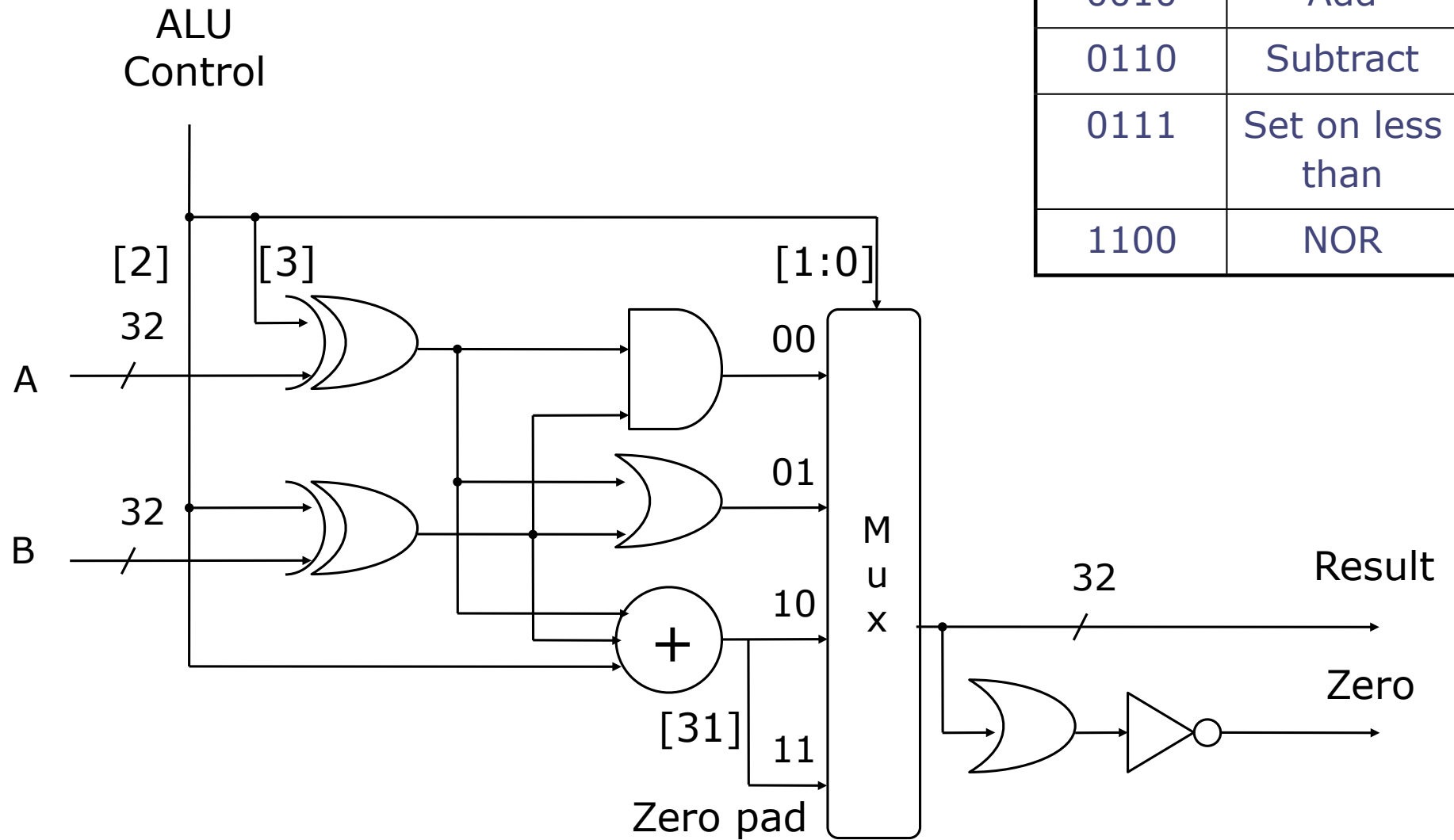


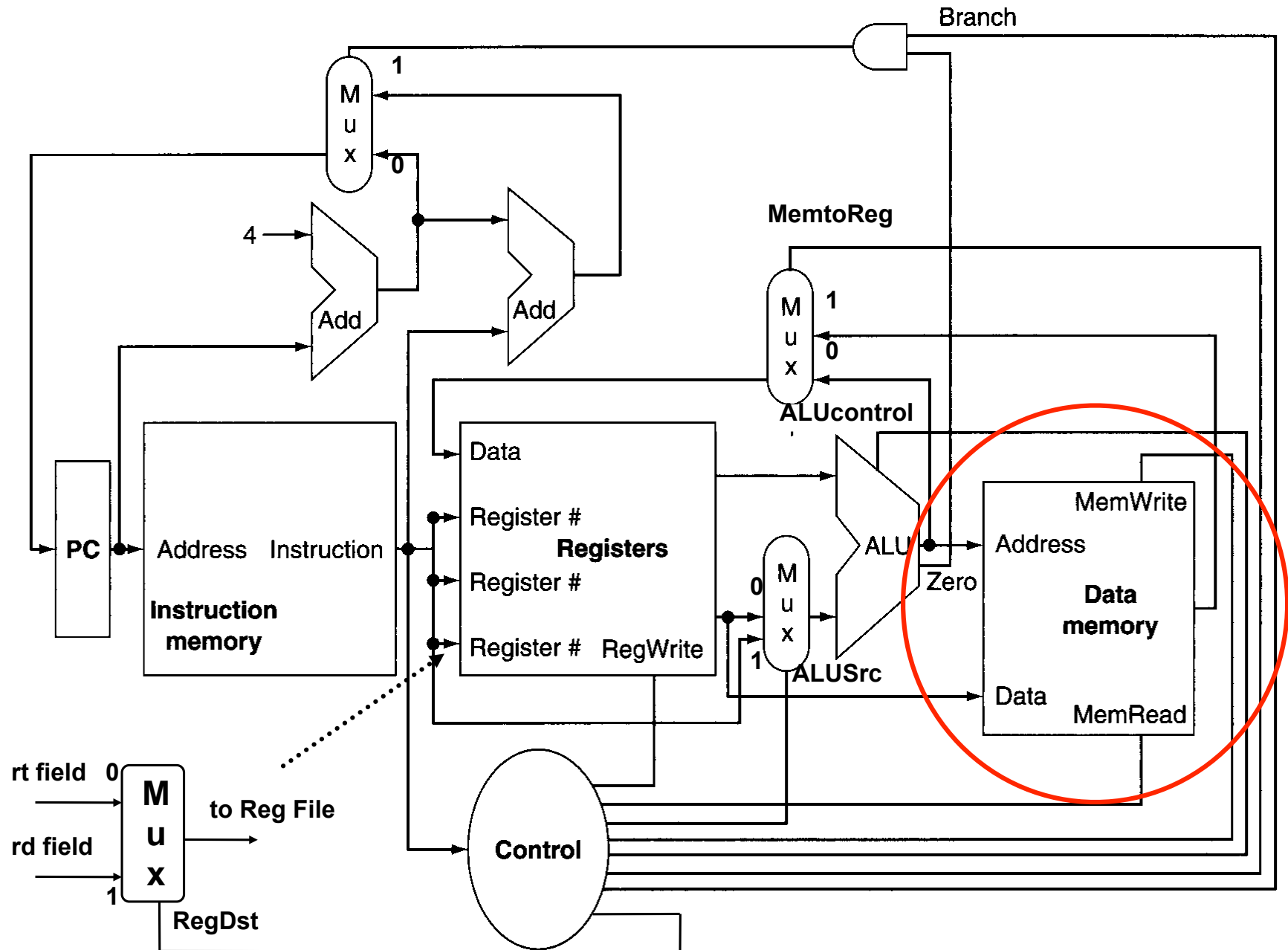
ALU NOR

- $Y = A \text{ NOR } B$
- $Y = \text{NOT } (A \text{ OR } B)$
- $Y = (\text{NOT } A) \text{ AND } (\text{NOT } B)$ de Morgan
- $Y = (A \text{ XOR } 1) \text{ AND } (B \text{ XOR } 1)$



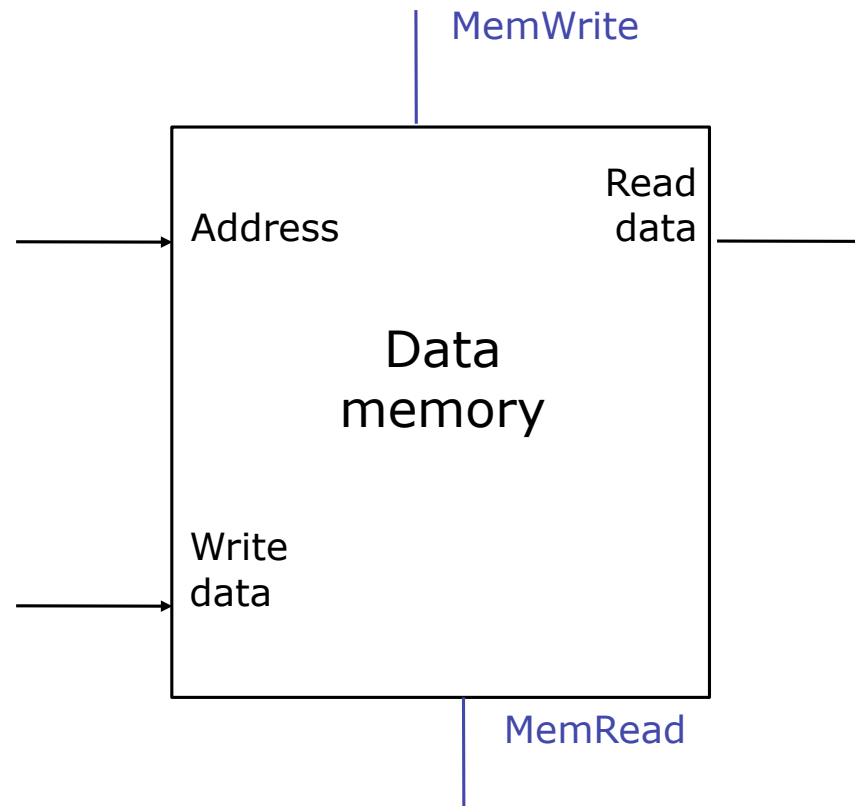
ALU

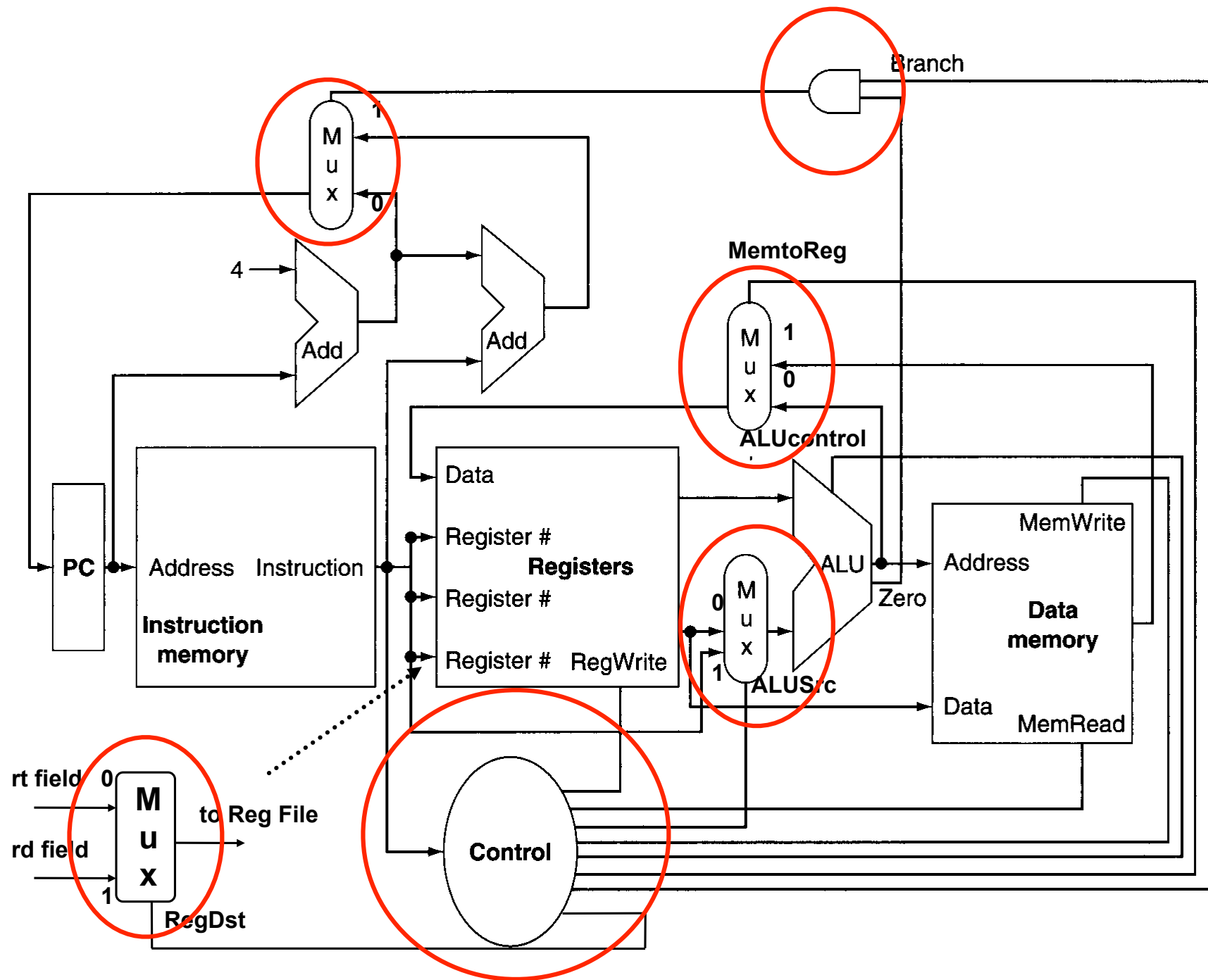




Memory

- Array of memory cells with select lines
- Slower and smaller than register file
- Instruction memory similar but pre-programmed in this case

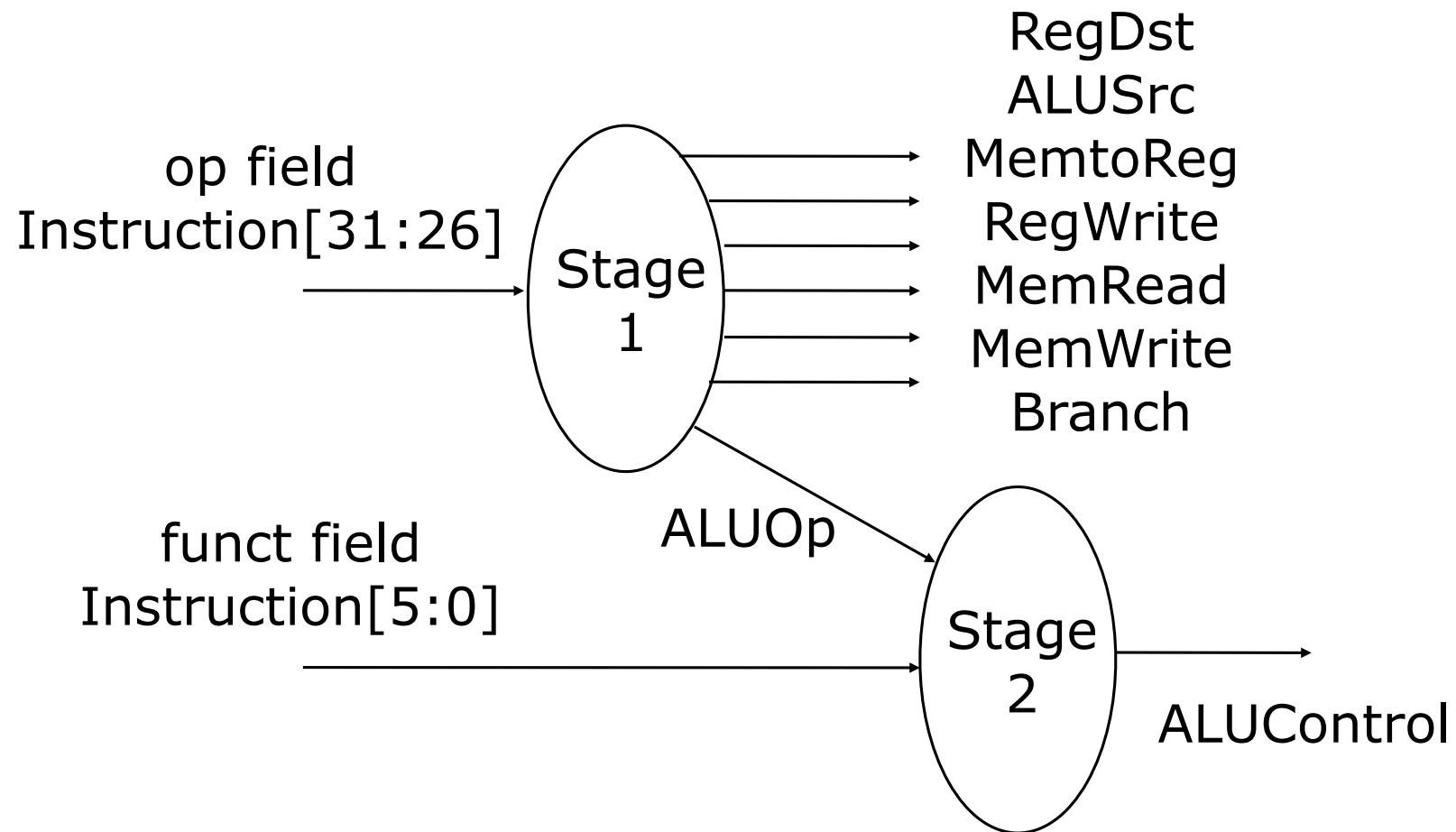




3. Single cycle MIPS Processor

Control Decode

- Control Unit uses two stage decode.



Control

Decode Stage 1

R-Format
lw
sw
beq

Instr. / Opcode	Reg Dst	ALU Src	Memto Reg	Reg Write	MemRead	MemWrite	Branch	ALU Op
00_0000 ₂	1	0	0	1	0	0	0	10
10_0011 ₂	0	1	1	1	1	0	0	00
10_1000 ₂	x	1	X	0	0	1	0	00
00_0100 ₂	x	0	x	0	0	0	1	01

Rformat = $\sim\text{opcode}[2] \ \& \ \sim\text{opcode}[5]$

RegDst = Rformat

ALUSrc = opcode[5]

MemtoReg = opcode[5]

RegWrite = opcode[0] + Rformat

MemRead = opcode[0]

MemWrite = opcode[3]

Branch = opcode[2]

ALUOp[1] = Rformat

ALUOp[0] = opcode[2]

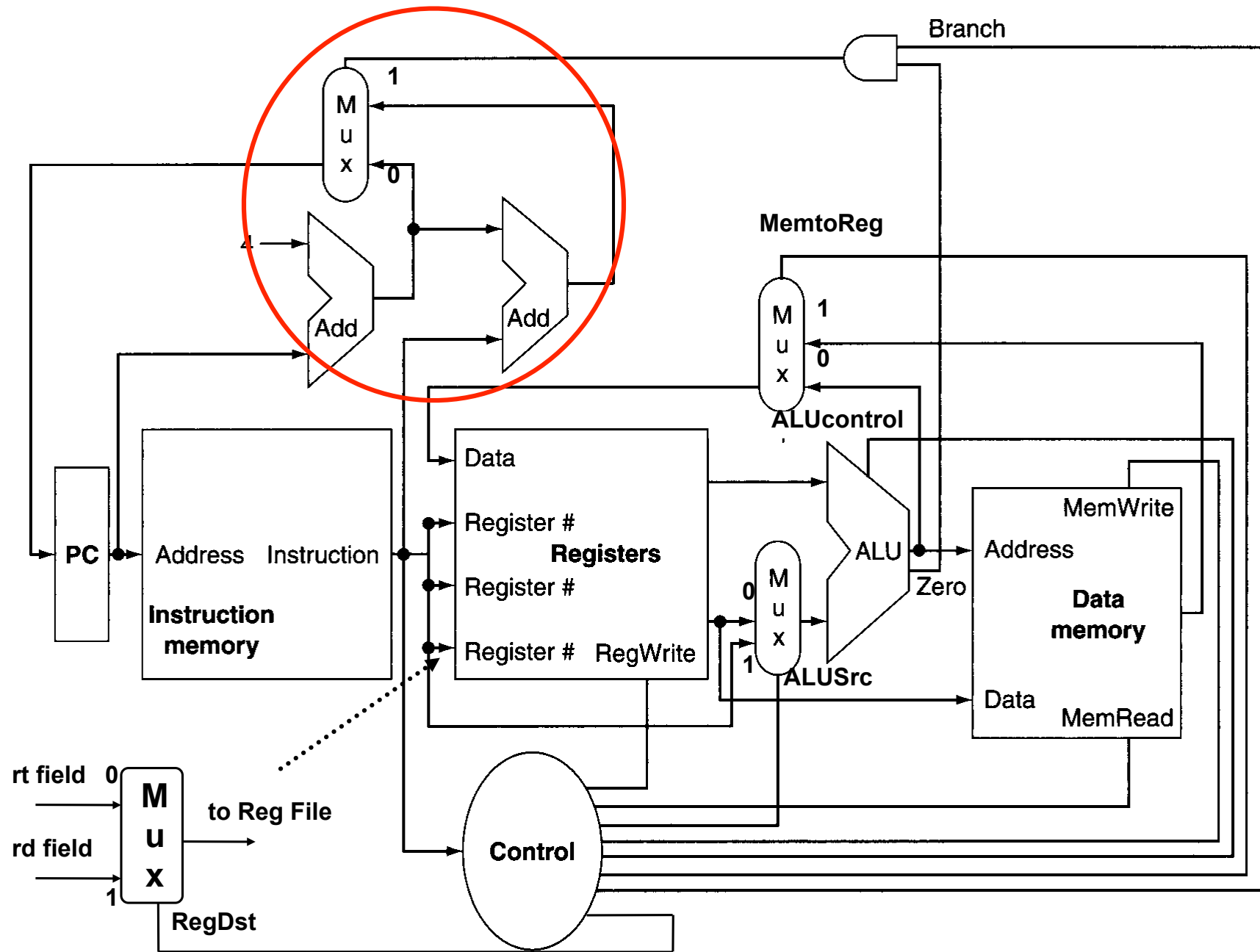
3. Single cycle MIPS Processor

Control

Decode Stage 2

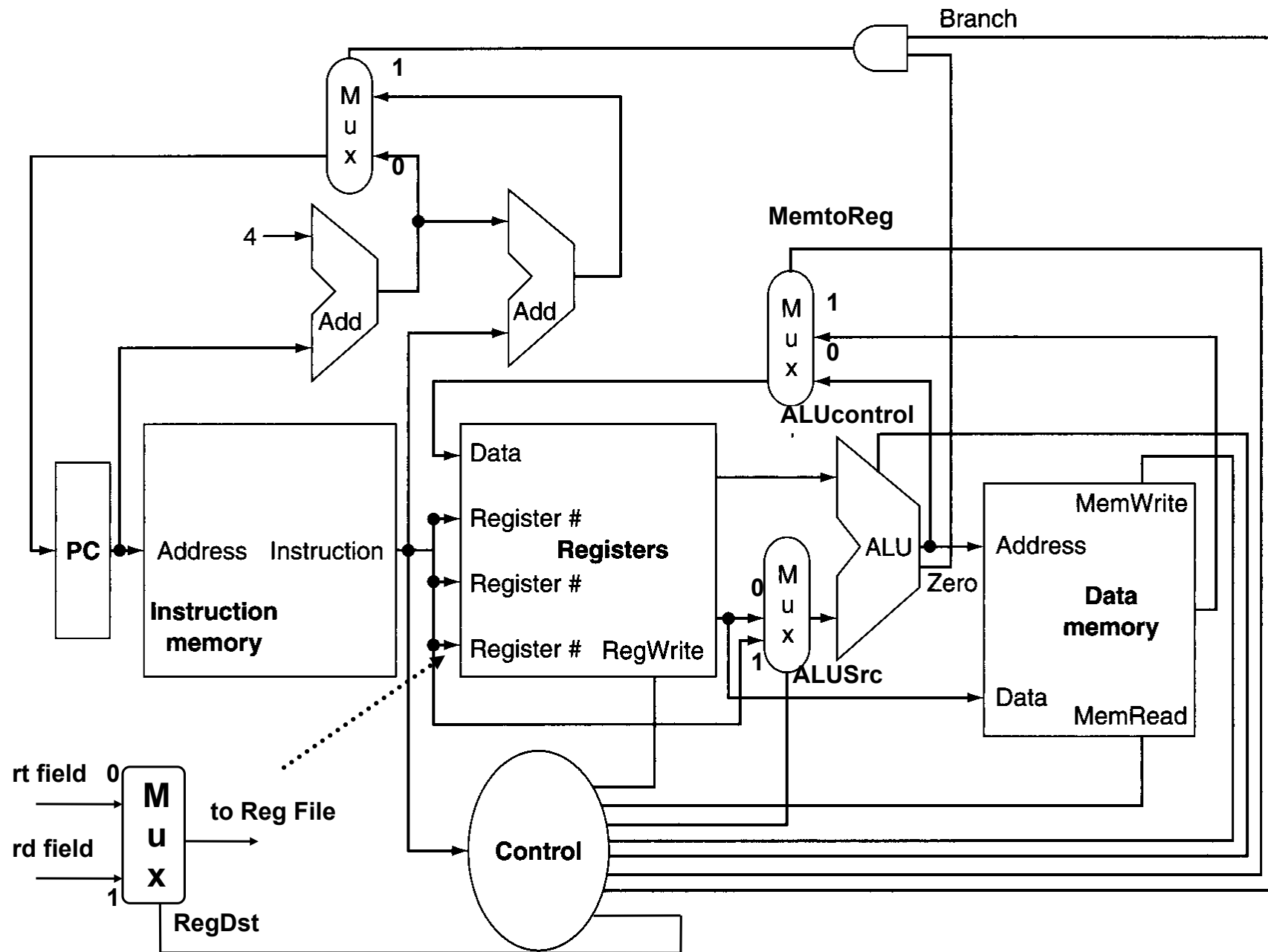
add
add
sub
add
sub
AND
OR
slt

Instruction	ALUop	funct	ALU Control
lw	00	xxxxxx	0010
sw	00	xxxxxx	0010
beq	01	xxxxxx	0110
add	10	100000	0010
sub	10	100010	0110
and	10	100100	0000
or	10	100101	0001
slt	10	101010	0111



Branch Logic

- Extra adder to calculate branch destination by adding relative address (from instruction) to address of next consecutive instruction.
- Multiplier to switch between next consecutive address and branch destination.



Resources

References

- Required reading:
 - Patterson & Hennessy
 - Sections 5.1 – 5.4
 - Harris & Harris
 - Sections 7.1 – 7.4
- Revision:
 - Patterson & Hennessy
 - Appendix B (on the CD) Logic Design
 - Harris & Harris
 - Chapters 2 & 3