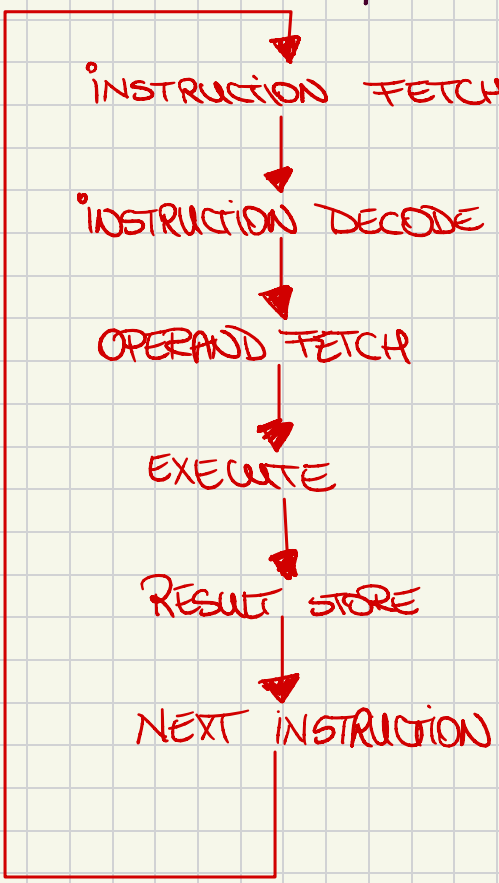
MIPS
Microprocessor with Interlocked Pipeline Stages

ISA - Instruction Set Architecture
(the interface between hardware and software)
→ Components:
- memory organization
- registers
- data types and data structures
- instruction formats
- instruction set
- addressing modes
- flow of control
- input / output
- interrupts



INSTRUCTION FETCH — read instruction from program memory

INSTRUCTION DECODE — determine the instruction format or encoding

OPERAND FETCH — locate and obtain the operands of the instruction and the location of the result

EXECUTE — compute results of the instruction (value, status, address)

RESULT STORE — write/save the result in register or memory

NEXT INSTRUCTION — determine the next instruction to be executed (address)
- normal op.
- jumps, conditions, branches

// for 32 X 32
↳ R0 - R31 general purpose registers
- R0 fixed value of 0
- R31 holds the return add of a procedure call

→ PC - program counter
- the address of the next instruction to be fetched

→ Hi, Lo - store partial result of multiplication and division

Memory Organization

- a large, single-dimension array
- a memory address = an index in the array

// for 32x32

- ↳ word 32 bits (4 bytes)
- ↳ 32 bit addresses ⇒ 2^{32} bytes ⇒ 2^{30} words

  0, 1, 2, ..., $2^{31}-1$

  0, 4, 8, ..., $2^{32}-4$

- endianness – little / big
- data alligment
  - for 16 bit word starts at multiple of 2 bytes addresses

PC → 0040 0000 hex

0000 0000 hex

- 3 instruction formats
  - R-type : used for arithmetical / logical operations
  - I-type : used for arithmetical / logical operations with immediate values, memory data transfers and conditional jumps or branches
  - J-type : used for unconditional jumps

	3	3	3	3	1	3
R type :	opcode	rs	rt	rd	sa	function

	3	3	3	7
I type :	opcode	rs	rt	address / imm

	3	13
J type :	opcode	target address

for R type, opcode is always 0 and the function field identifies the ALU operation for each instruction
- also 3 bits

⇒ 15 instructions : 8 R type
7 I type
1 J type

I chose to implement these 16 instructions :

R type :

ADD

add $d, $s, $t

000_rs_rt_rd_0_000

$d ← $s + $t

SUB

sub $d, $s, $t

000_rs_rt_rd_0_001

$d ← $s - $t

rs ALWAYS the source register

rt – for R type : source register

– for I type : destination register

rd ALWAYS destination register ONLY for R type

SLL
sll $d, $t, a
000_rs_rt_rd_a_010
$d ← $t << a

SRL
srl $d, $t, a
000_rs_rt_rd_a_011
$d ← $t >> a

AND
and $d, $s, $t
000_rs_rt_rd_0_100
$d ← $s & $t

OR
or $d, $s, $t
000_rs_rt_rd_0_101
$d ← $ | $t

XOR
xor $d, $s, $t
000_rs_rt_rd_0_110
$d ← $s ^ $t

NOR
nor $d, $s, $t
000_rs_rt_rd_0_111
$d ← !($s | $t)

I type

ADDI
addi $t, $s, imm
001_rs_rt_$\underbrace{iiiiiii}_{7}$
$t ← $s + imm

LW
lw $t, offset ($s)
010_rs_rt_ttttttt
$t ← MEM[$s + offset]

ex: lw $t0, 8($t1)
↳ puts in to what is in memory at (t1+8)

SW
sw $t, offset ($s)
011_rs_rt_ttttttt
MEM[$s + offset] ← $t

BEQ
beq $s, $t, offset
100_rs_rt_ttttttt

```
if $s == $t
    advance_programcounter(offset<<2);
else
    advance_programcounter(4)
```

why <<2 ??? because jumps works with words (4 bytes), so the offset (which is a word) turns into bytes if we multiply by 4 => <<2

ANDI

andi $t, $s, imm

101_rs_rt_iiiiiii

$t ← $s & imm

ORI

ori $t, $s, imm

101_rs_rt_iiiiiii

$t ← $s | imm

J-type :

j target

111_ $\underbrace{jjjjjjjjjjjjj}_{13}$

PC ← nPC; $\boxed{nPC = (PC \,\&\, 0xf000) \,|\, (target << 2)}$

why???

target is stored like an offset, so word type => we need to shift it by 2 to get the bytes

PC & 0xf000 → the 4 bits of the superior PC value; do this to make sure you stay in the same memory segment

when " | (target <<2) ", we jump in the same memory segment, but we change the last 12 bits

I chose to do the following algorithm:
→ Fibonacci numbers

```
0   001_000_001_0000000 -> addi $1, $0, 0
1   001_000_010_0000001 -> addi $2, $0, 1
2   001_000_011_0000000 -> addi $3, $0, 0
3   001_000_100_0000001 -> addi $4, $0, 1
4   011_011_001_0000000 -> sw $1, 0($3)
5   011_100_010_0000000 -> sw $2, 0($4)
6   010_011_001_0000000 -> lw $1, 0($3)
7   010_100_010_0000000 -> lw $2, 0($4)
8   000_001_010_101_0_000 -> add $5, $1, $2
9   000_000_010_001_0_000 -> add $1, $0, $2
10  000_000_101_010_0_000 -> add $2, $0, $5
11  111_000_000_0001000 -> j 8
```

store 0 in $1, $3
store 1 in $2, $4

offset instruction

Components of the MIPS:

→ PC program counter
16-bit edge trigger D flip-flop

→ IM instruction memory (ROM)
- one input bus: instruction address
- one output bus: instruction data
- memory word is 16 bit (selected by instruction address)
- NO control signals

→ RF register file
- 2 read addresses and 1 write address
- 8 16-bit registers (rs, rt, rd encoded on 3 bits)
- 2 16-bit data outputs (Read data 1, Read data 2)
- 1 16-bit data input (write data)
- multiple accesses (2 asynchronous reads and 1 synchronous edge triggered write); during the read operation, the register file behaves as a combinational logic block
- 1 control signal (RegWrite); when RegWrite is asserted the value on the Write Data line is written in the register indicated by the write address line

→ DM data memory (RAM)
- 1 16 bit input address bus: Address
- 1 16 bit input data bus: Write Data
- 1 16 bit output data bus: Read Data
- 1 control signal: MemWrite

→ EU extension unit
- ExtOp = 1 ⇒ sign extension
  = 0 → zero extension

→ ALU arithmetic logic unit
- performs arithmetical/logical operations

! identify all the operations that the ALU needs to perform after completing the definition of the instructions

! identify how many control bits are necessary to encode the ALU operations (ALU ctrl)

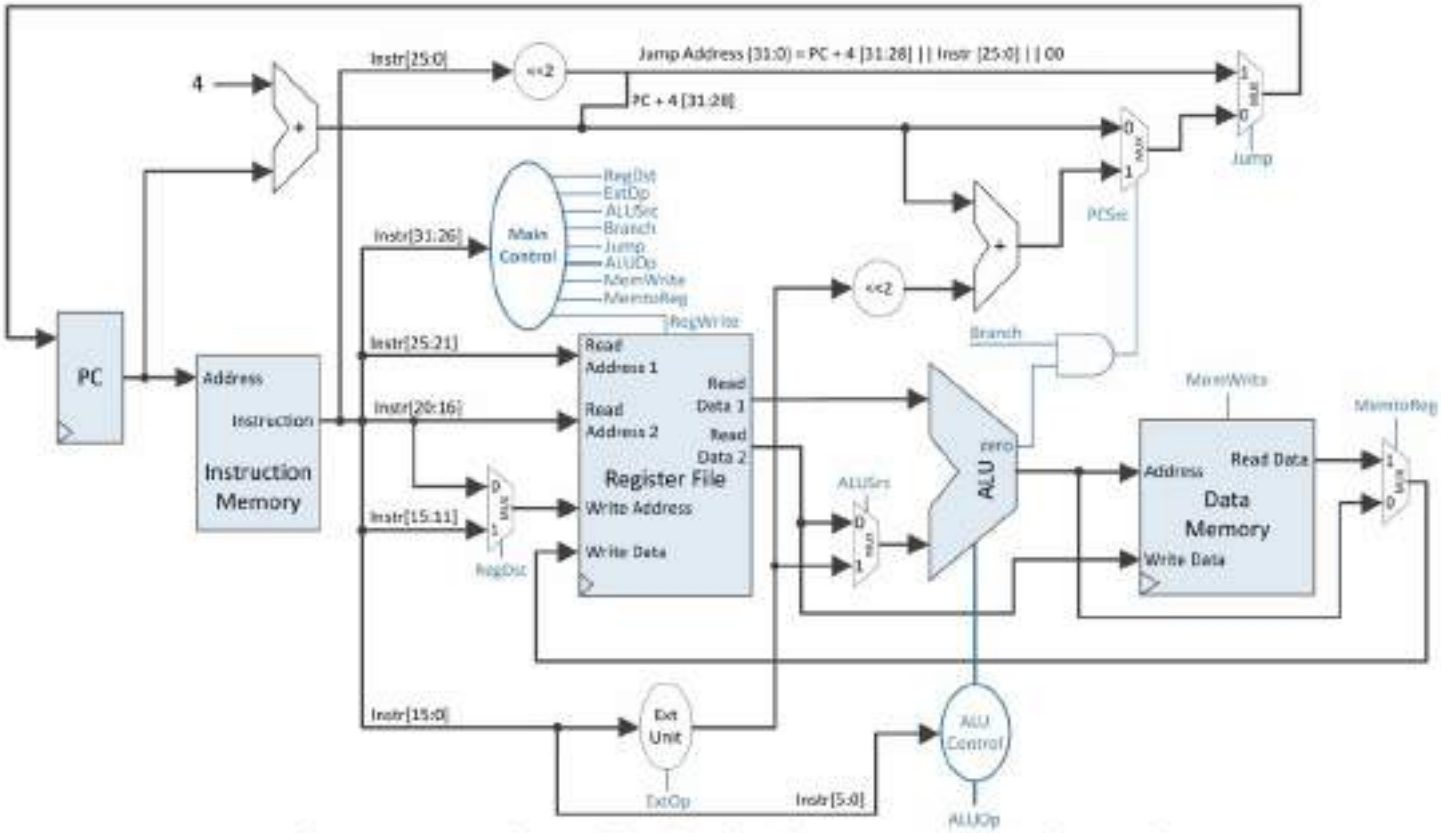


Figure 1: MIPS 32 Single-Cycle Data-Path + Control

→ instruction execution cycle
IF, ID, EX, MEM, WB

+ we will need the SSD and the MPG

INSTRUCTION FETCH
- program counter
- instruction memory (ROM)
- adder
- 2 muxs for selecting the next instruction address

↳ inputs:
- clock signal (for PC)
- branch target address
- jump address
- jump control signal
- PCSrc control signal (for branch)

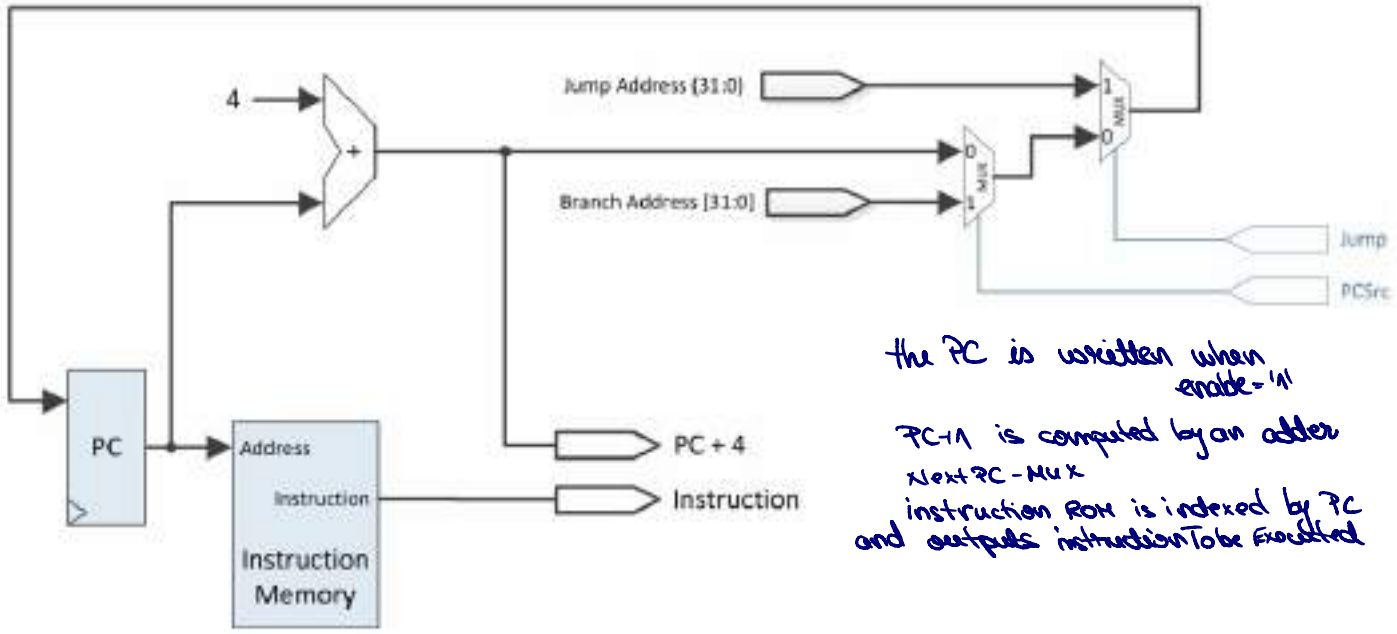
↳ outputs:
- the instruction to be executed by the MIPS
- the next sequential instruct. address (PC+4)

↳ Jump
- L = 1 ⇒ PC ← jump address
- L = 0 ⇒ PCSrc

+4 if 32x32

- L = 0 ⇒ PC ← PC+1
- L = 1 ⇒ PC ← branch address

! the new PC value will be written in the PC register only when a button from the board is pressed (an enable signal from the MPG given as input to the IF Unit) + use another MPG signal to reset the PC register



the PC is written when enable = '1'

PC+1 is computed by an adder

NextPC - MUX

instruction ROM is indexed by PC and outputs instructionToBe Executed

Figure 5: Instruction Fetch Data-Path for MIPS 32

INSTRUCTION DECODE UNIT -> read registers, extract fields, extend imm

- register file
- multiplexer
- sign/zero extender

↳ inputs

- clock (for the Register File Writes)
- the 16 bit instruction
- the 16 bit Write Data for the Register File
- control signals:
  - RegWrite (write enable for the Reg file)
  - RegDst (selects the write address for the Reg file)
  - ExtOp (selects between Sign and Zero extension of the immediate field)

↳ outputs

- register from rs address : 16 bit Read Data 1
- register from rt address : 16 bit Read Data 2
- 16 bit extended immediate
- 3 bit function field
- 1 bit shift amount for R type shift instructions

↳ RegDst

- L = 1 => the write address for the reg file is the rd field of the instruction
- L = 0 => the write address for the reg file is the rt field of the instruction

↳ RegWrite = 1 => write the value provided by the write data signal into the write address register in reg file

↳ ExtOp
L = 0 => perform zero extension of the 16 bit imm
L = 1 => performs sign extension of the 16bit imm

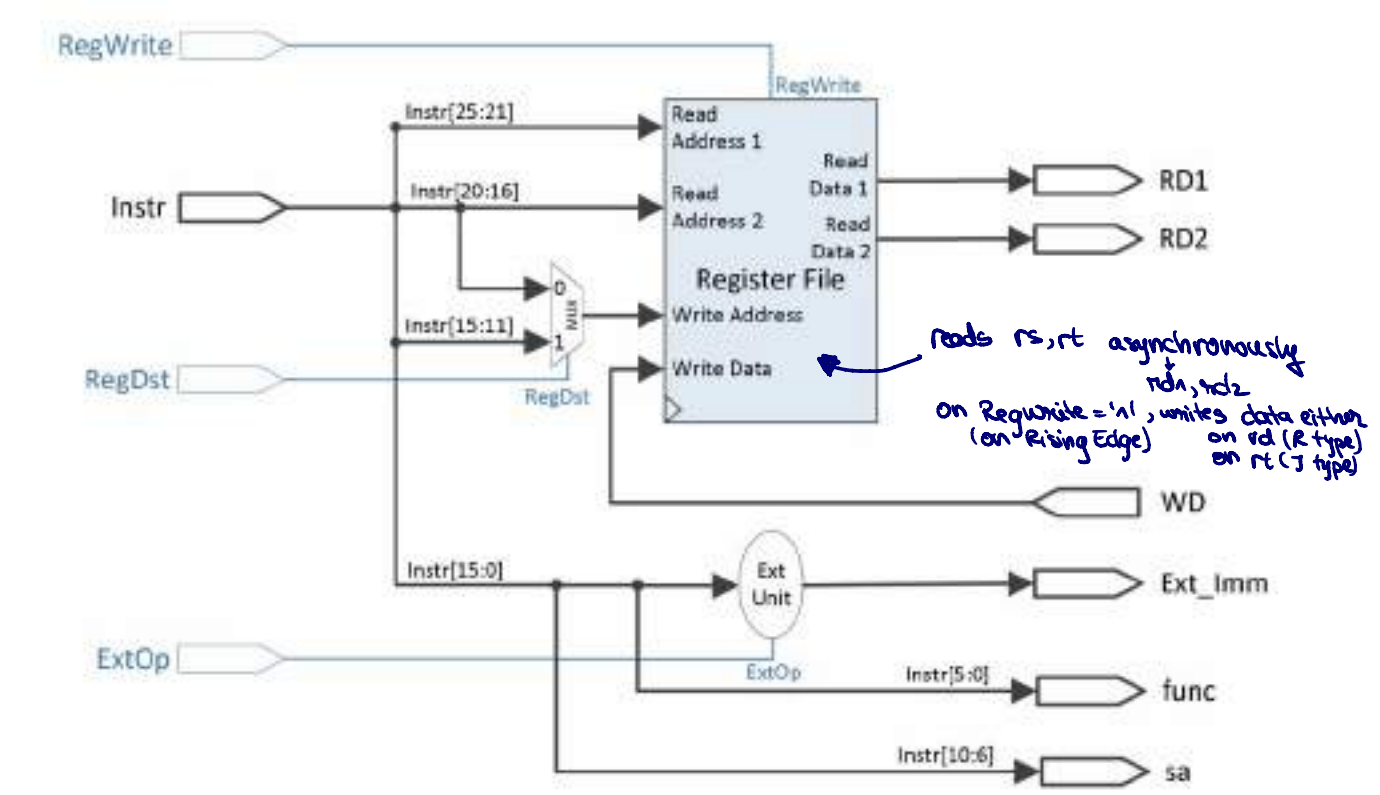


Figure 5: Instruction Decode Data-Path for MIPS 32

MAIN CONTROL UNIT

↳ inputs
L the 3-bit opcode field of the instruction
↳ outputs
L the main data-path control signals (except for ALUCtr)

connect the control signals generated by the Main Control Unit to the IF and ID

! for writing in the register file it is necessary to control the writing mechanism from one of the buttons available on the board ; MUST BE VALIDATED (with an AND gate) WITH ONE OF THE MPG OUTPUTS (use the same enable signal as for write validation in the PC register)

! FOR CONNECTING THE SSD:
use a MUX

sw(7:5) = 000 => display the result on the SSD
sw(7:5) = 001 => display the next PC (PC+1) on the SSD
sw(7:5) = 010 => display RD1 on the SSD
sw(7:5) = 011 => display RD2 on the SSD

sw(7:5)=100 => display the WD signal on the SSD
sw(7:5)=101 => display the ALURes signal on the SSD
sw(7:5)=110 => display the MemData signal on the SSD
sw(7:5)=111 => display the WD signal on the SSD

! on the LEDs, we display the control signals from the Main Control Unit

we need 1 switch to control the display on the LEDs:
- sw(0)=0 => display the 1-bit control signals on the LED
- sw(0)=1 => display the m-bit ALUOp signal on the LED (the other LEDs have'd)

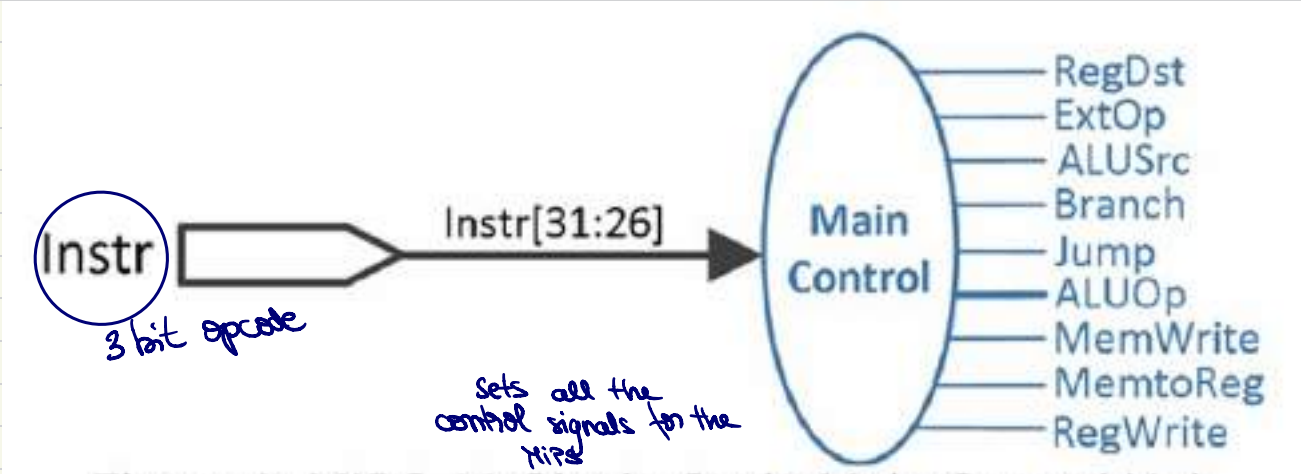


Figure 6: MIPS 32 Single Cycle Main Control Unit

EXECUTE UNIT
- ALU
- ALU Control
- Multiplexer
- Shift Left 2 & Adder for branch target address computation

↳ inputs
- the next sequential IA (PC+1)
- RD1
- RD2
- 16 bit extended immediate
- 3 bit funct field
- 1 bit shift amount
- control signals
  - ALUSrc → selects between RD2 and Ext_imm as input to the second port of the ALU
  - ALUOp → provided by the main control

↳ outputs
- 16 bit branch target address
- 16 bit ALU result
- 1 bit zero-signal
  - this indicates whether the result of the ALU is equal to 0 or not

↳ ALUSrc
- L = 0 ⇒ RD2 is the second input for the ALU
- L = 1 ⇒ Ext_imm is the second input for the ALU

Branch Adress ← PC + 1 + S_Ext(Imm) << 2

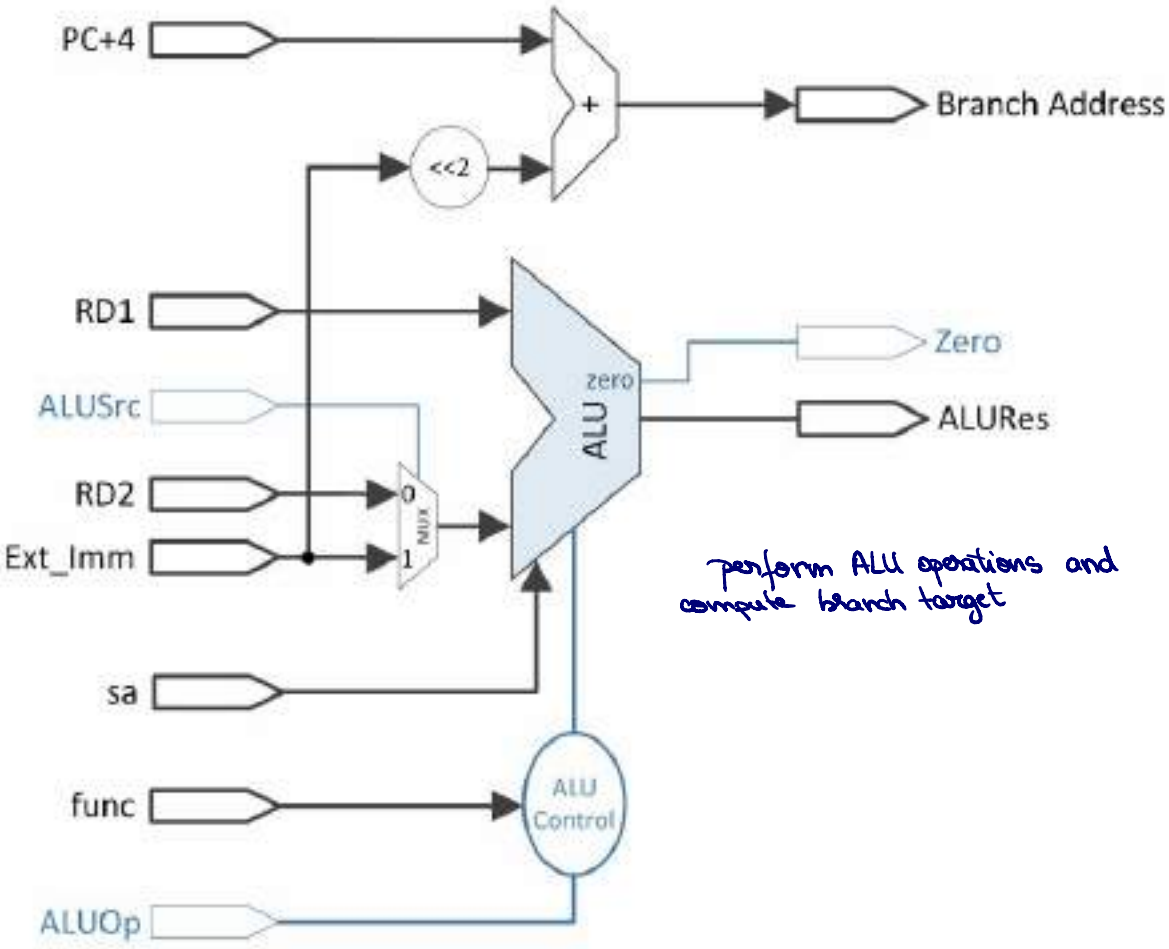


Figure 5: Instruction Execute Data-Path for MIPS 32

MEMORY UNIT
- L data memory

RAM asynchronous read and synchronous write

↳ inputs
- L clock signal (for Data Memory Writes)
- L 16 bit ALURes signal
- L 16 bit RD2 signal
- L MemWrite control signal

↳ outputs
- L 16 bit MemData (used only for load instructions)
- L 16 bit ALURes

the write back unit is just the MUX from fig 1 and the

Other components are: the AND gate for generating the PCSrc, the jump address computation

↳ MemtoReg

L 0=0 => ALURes signal is the input to the Write Data port of the Register file

L=1 => the MemData is the input to the Write Data port of the Register file

PCSrc ← Branch and Zero;

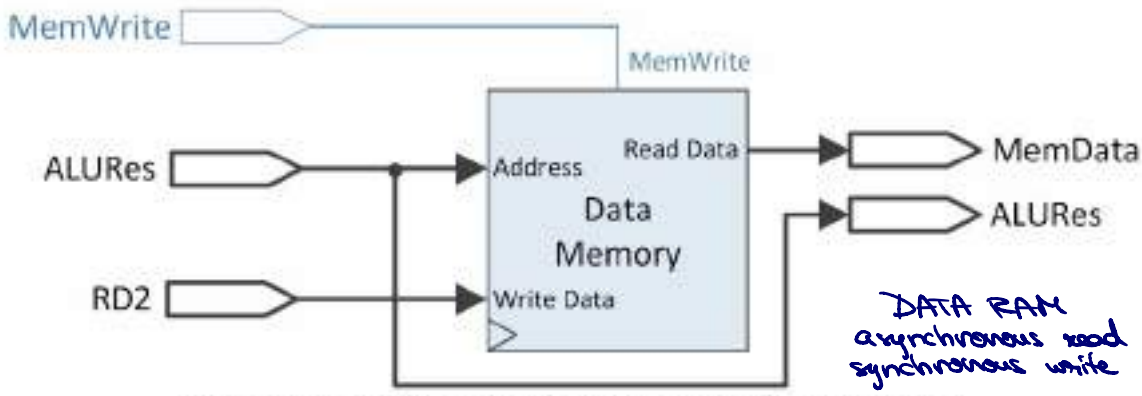


Figure 6: Memory Unit Data-Path for MIPS 32

WriteBack component selects ALU result or memory data for writing registers