# MIPS Load & Stores

Pick up a handout

## Today's lecture

- **MIPS Load & Stores**
  - Data Memory
  - Load and Store Instructions
  - Encoding
  - How are they implemented?

# We need more space!

- Registers are fast and convenient, but we have only 32 of them, and each one is just 32-bits wide.
  - That’s not enough to hold data structures like large arrays.
  - We also can’t access data elements that are wider than 32 bits.

- We need to add some main memory to the system!
  - RAM is cheaper and denser than registers, so we can add lots of it.
  - But memory can be significantly slower, so registers should be used whenever possible.

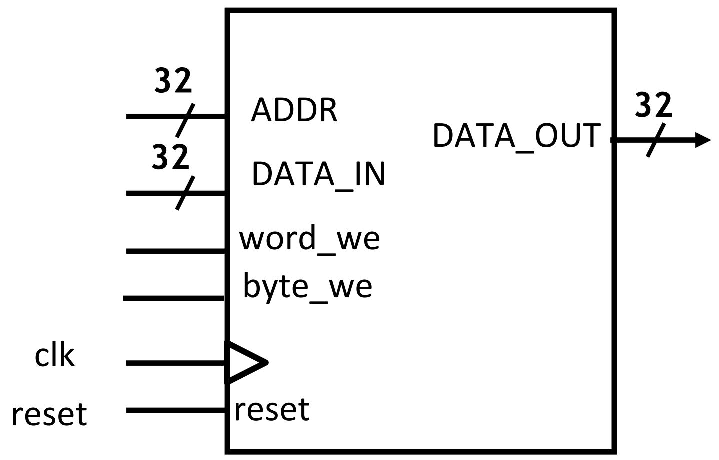
## Harvard Architecture

- It's easier to use a Harvard architecture at first, with programs and data stored in *separate* memories:
  - 1 Instruction memory:
    - Contains instructions to execute
    - It is read-only
  - 2 Data memory
    - Contains the data of the program
    - Can be read/written

## MIPS memory

- MIPS memory is byte-addressable, which means that each memory address references an 8-bit quantity.
- The (original) MIPS architecture can support up to 32 address lines.
    - This results in a $2^{32}$ x 8 RAM, which would be 4 GB of memory.

## Data Memory



| word_we | byte_we | Operation |
|---|---|---|
| 0 | 0 | Read |
| 0 | 1 | Write byte in ADDR |
| 1 | 0 | Write word in ADDR |

- ADDR specifies the memory location to access
- To write to the memory,
  - when word_we=1, the 32 bits in DATA_IN are stored in ADDR
  - when byte_we =1, DATA[0:7] bits are stored in ADDR.
- To read the memory,
  - word_we=0 and byte_we=0. DATA_OUT are the 32 bits stored in ADDR.

# Loading and storing words

- The MIPS instruction set includes load and store instructions for accessing memory.
- MIPS uses indexed addressing.
  - The address operand specifies a signed constant and a register.
  - These values are added to generate the effective address.
- The MIPS "load woard" instruction lw transfers one word of data from the data memory to a register.

```
lw $12, 4($3)
```

$\$12 = Mem[\underbrace{4 + R[\$3]}_{Address}]$

- The "store word" instruction sw transfers one word of data from a register into main memory.

```
sw $12, 4($3)
```

$Mem[\underbrace{4 + R[\$3]}_{Address}] = \$12$

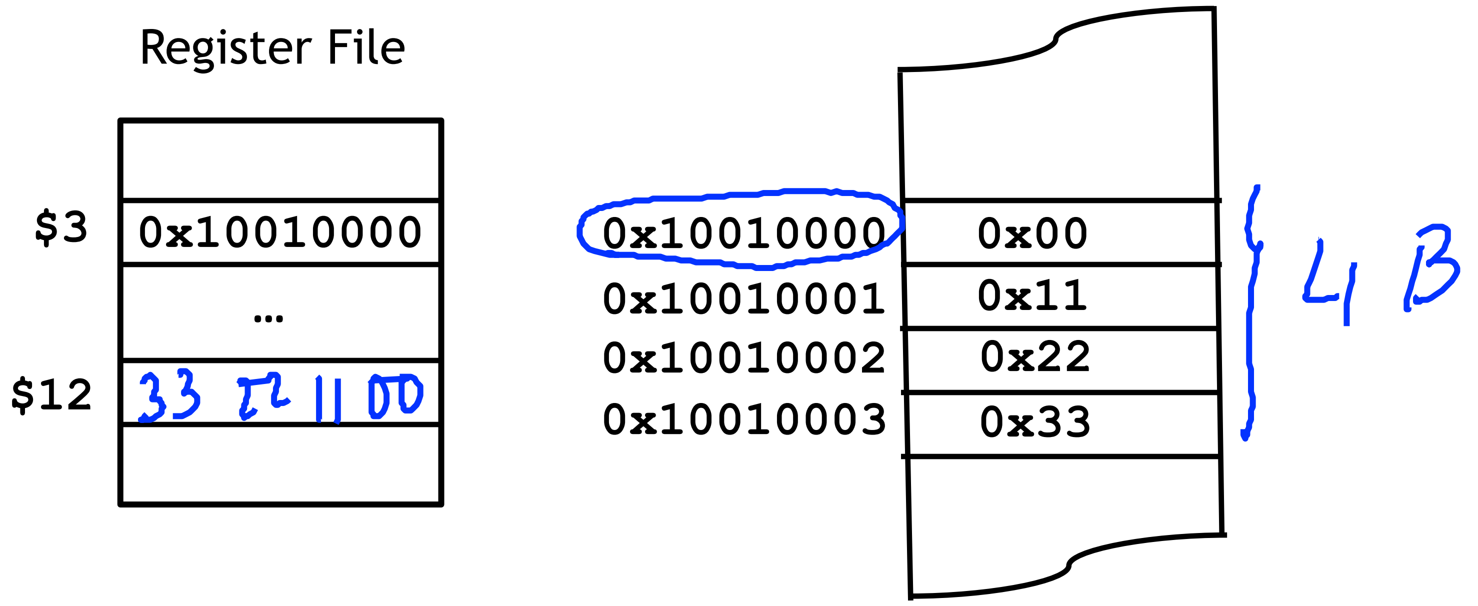
## Example

$12 = Memory[0+[$3]]

Address = 0+ 0x10010000

```
lw $12, 0($3)
```

word = 32 bits = 4 B



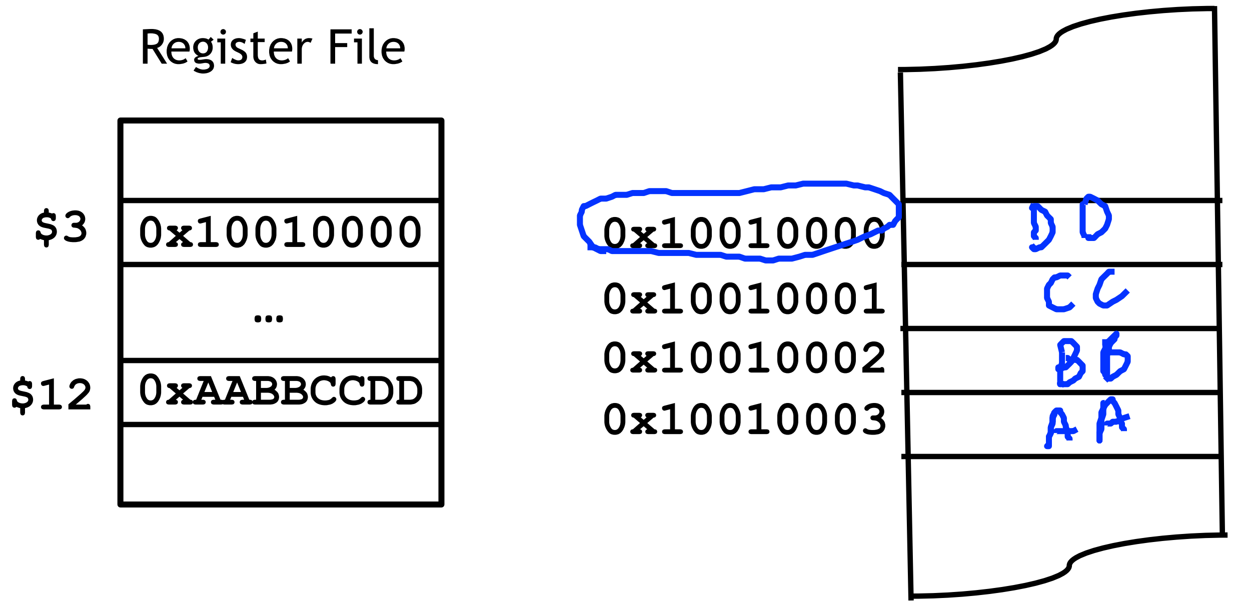
## Example

Memory[0+r[$3]] = $12

Address = 0 + 0x10010000

```
sw $12, 0($3)
```

word = 32 bits = 4B



# Loading and storing bytes

- The MIPS "load byte unsigned" instruction lbu transfers one byte of data from the data memory to a register.

```
lbu $12, 2($3)
```

$12 = Memory [2 + R[$3]]

- The "store byte" instruction sb transfers one byte of data from a register into main memory.

```
sb $12, 2($3)
```

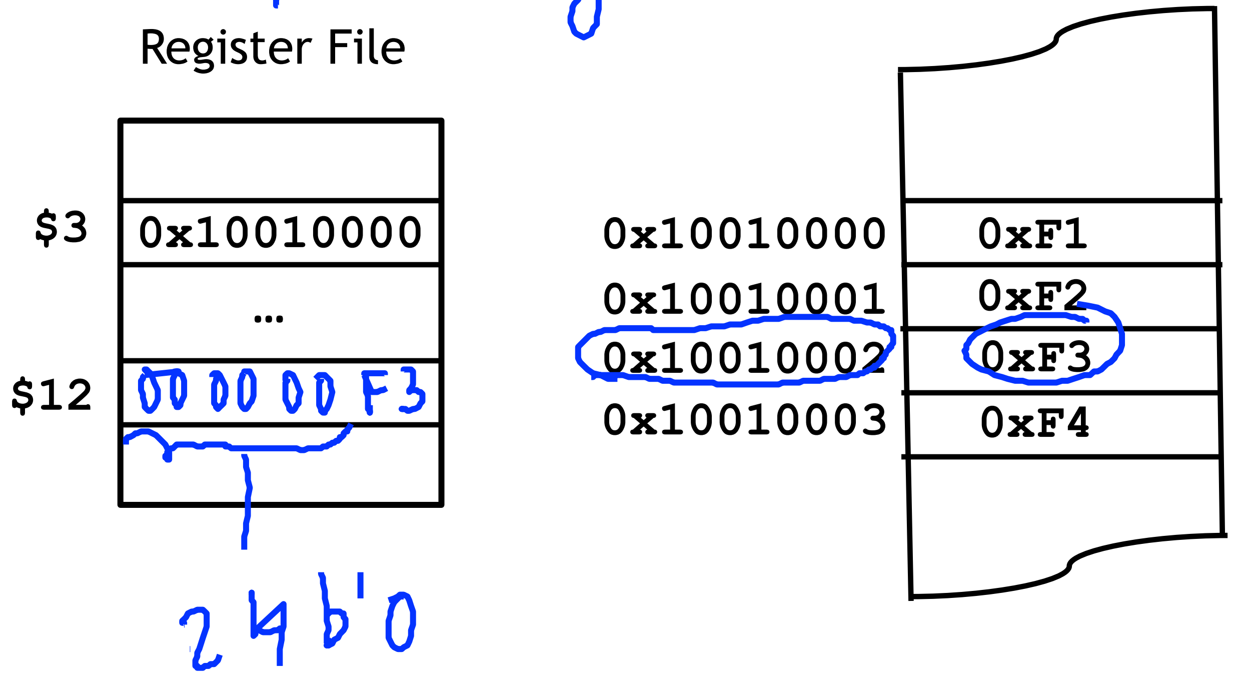
Memory [2 + R [$3]] = $12

## Example

$\$12 = \text{Memory}[2 + R[\$3]]$

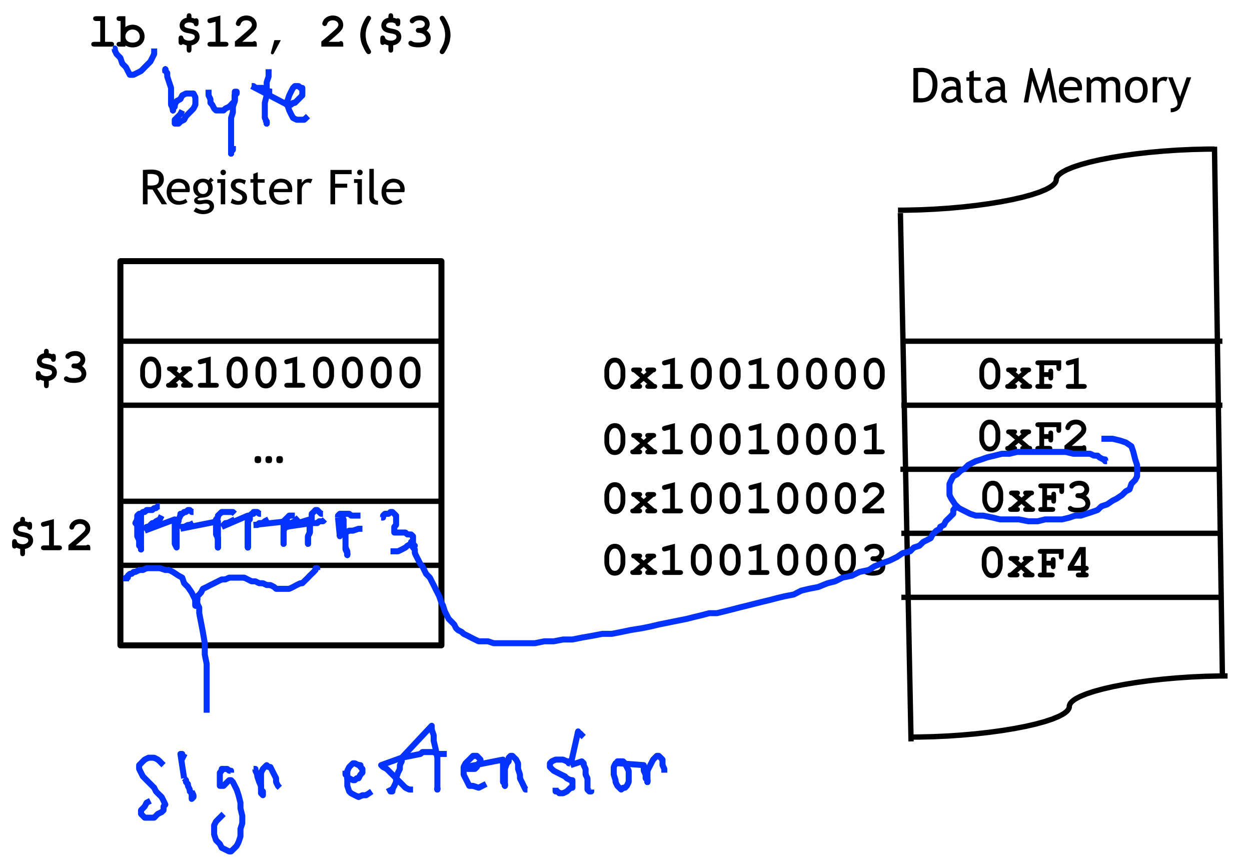
`lbu $12, 2($3)`

byte unsigned



# Example

`lb $12, 2($3)`

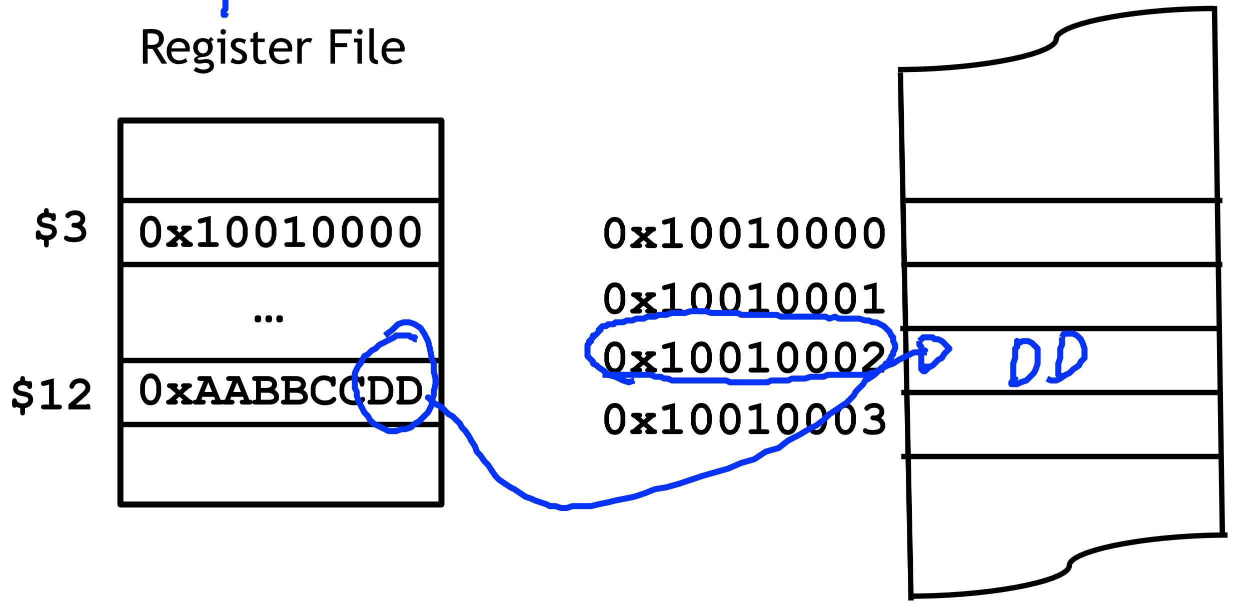


## Example

Memory[2 + r[$3]] = $12

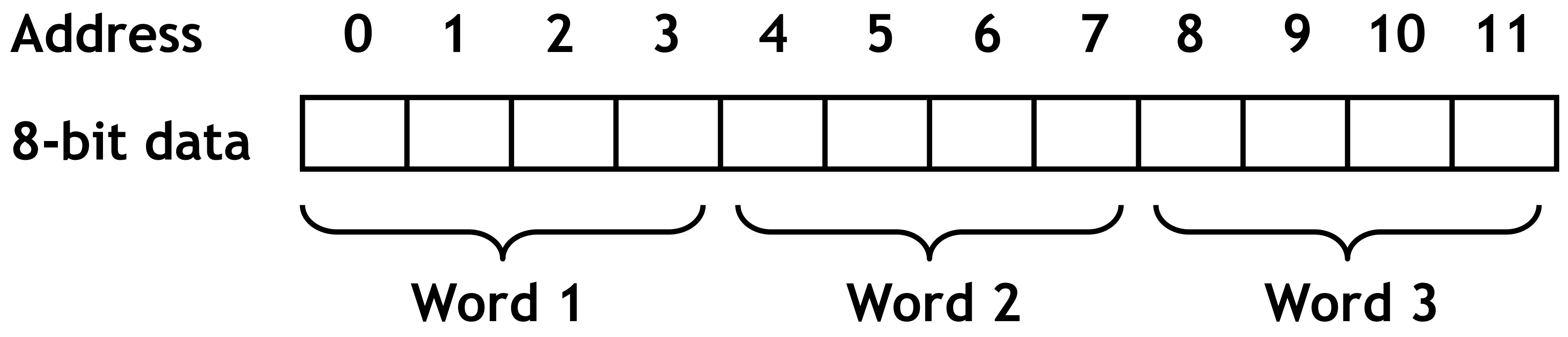
```
sb $12, 2($3)
```

byte



## Memory alignment

- Keep in mind that memory is byte-addressable, so a 32-bit word actually occupies four contiguous locations (bytes) of main memory.



- The MIPS architecture requires words to be aligned in memory; 32-bit words must start at an address that is divisible by 4.
  - 0, 4, 8 and 12 are valid word addresses.
  - 1, 2, 3, 5, 6, 7, 9, 10 and 11 are *not* valid word addresses.
  - Unaligned memory accesses result in a bus error, which you may have unfortunately seen before.
- This restriction has relatively little effect on high-level languages and compilers, but it makes things easier and faster for the processor.

## Example Program that Uses Memory

```
int a = 10;
int b = 0;
void main() {
   b = a+7;
}
```

## Example Program that Uses Memory

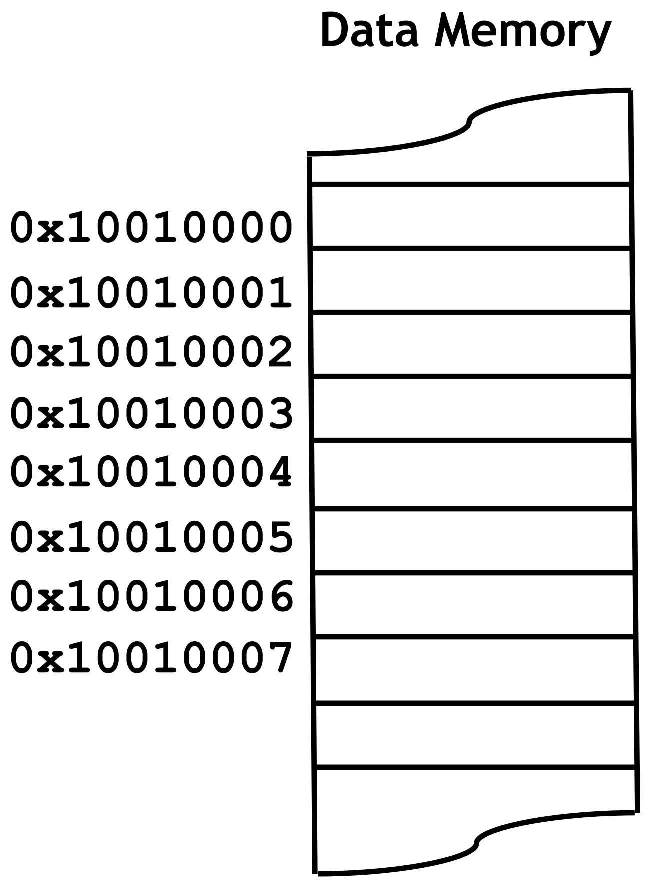
```
int a = 10;
int b = 0;
void main() {
   b = a+7;
}
```

```
.data
a: .word 10
b: .word 0
```

## Example Program that Uses Memory

```
int a = 10;
int b = 0;
void main() {
   b = a+7;
}
```

```
.data
a: .word 10
b: .word 0
.text
main:
   la  $4, a
```



# Example Program that Uses Memory

```
int a = 10;
int b = 0;
void main() {
    b = a+7;
}
```

```
.data
a: .word 10
b: .word 0
.text
main:
    la   $4, a
    ....
```

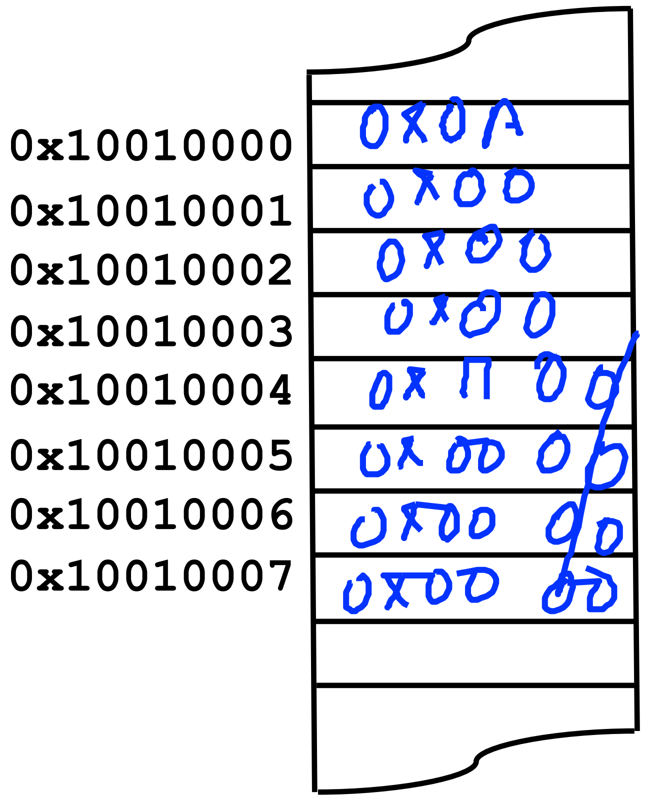
$4 = 0x10010000

A
```
lw   $5, 0($4)
addi $5, $5, 7
sw   $5, 0($4)
```

[✓ B]
```
lw   $5, 0($4)
addi $5, $5, 7
sw   $5, 4($4)
```

C
```
lw   $5, 4($4)
addi $5, $5, 7
sw   $5, 4($4)
```

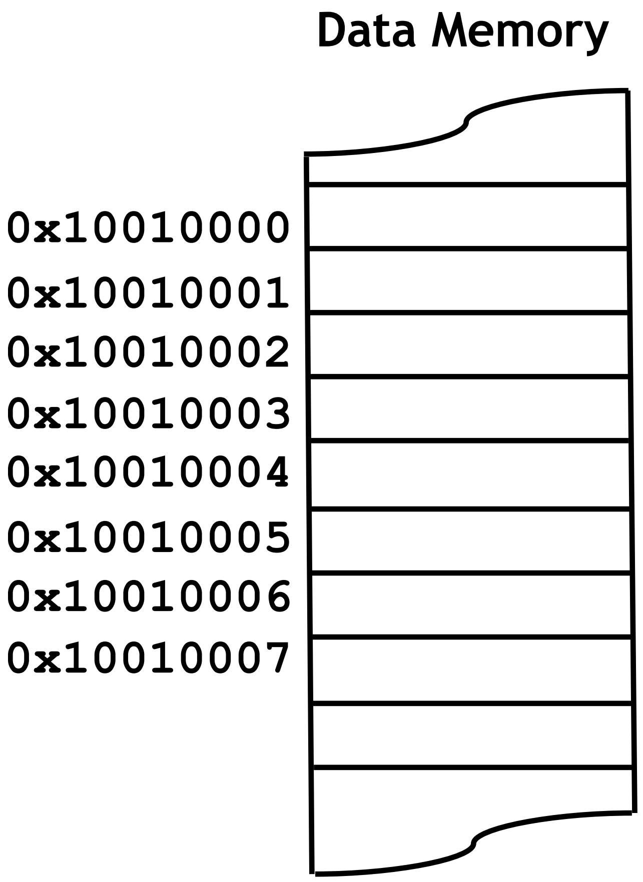
Data Memory



## Example Program that Uses Memory

```
int a = 10;
int b = 0;
void main() {
   b = a+7;
}
```

```
.data
a: .word 10
b: .word 0
.text
main:
    la   $4, a
    lw   $5, 0($4)
    addi $5, $5, 7
    sw   $5, 4($4)
```



## Enconding of loads and stores

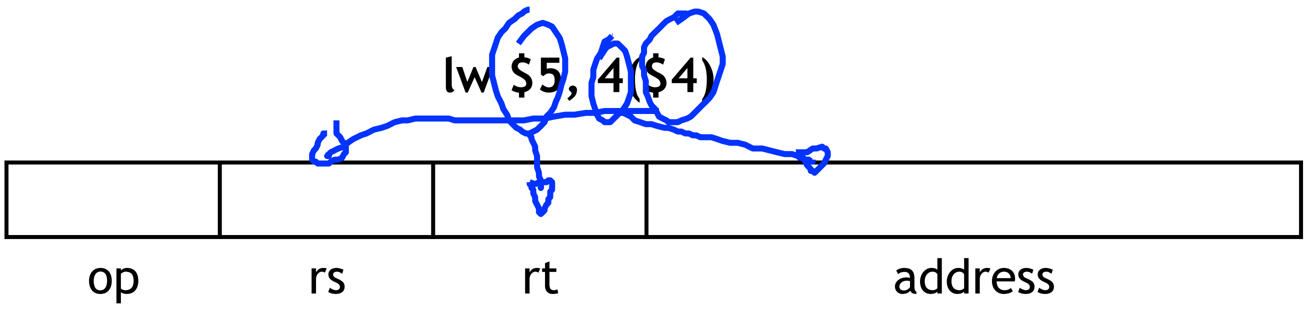
- **Loads and stores use the I-type format.**

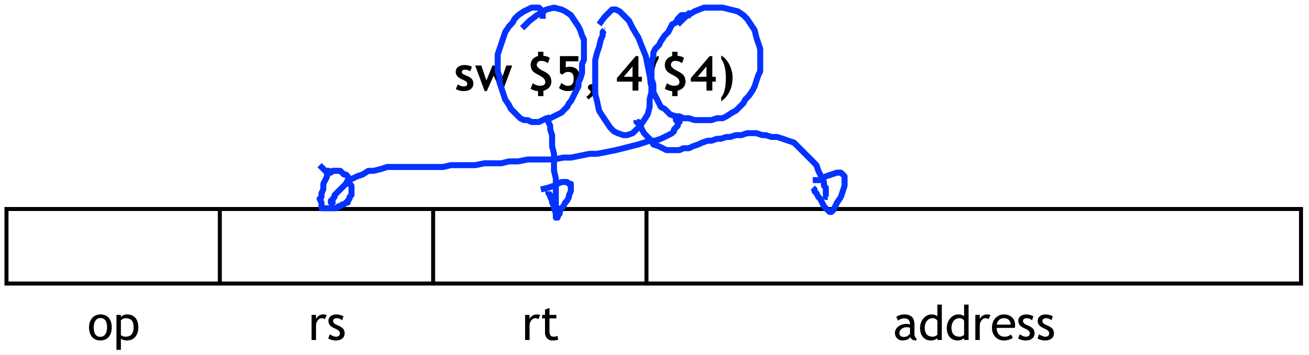
| op | rs | rt | address |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

- **The meaning of the register fields depends on the exact instruction.**
  — rs is a source register—an address for loads and stores
  — rt is the destination for load, but a source for store

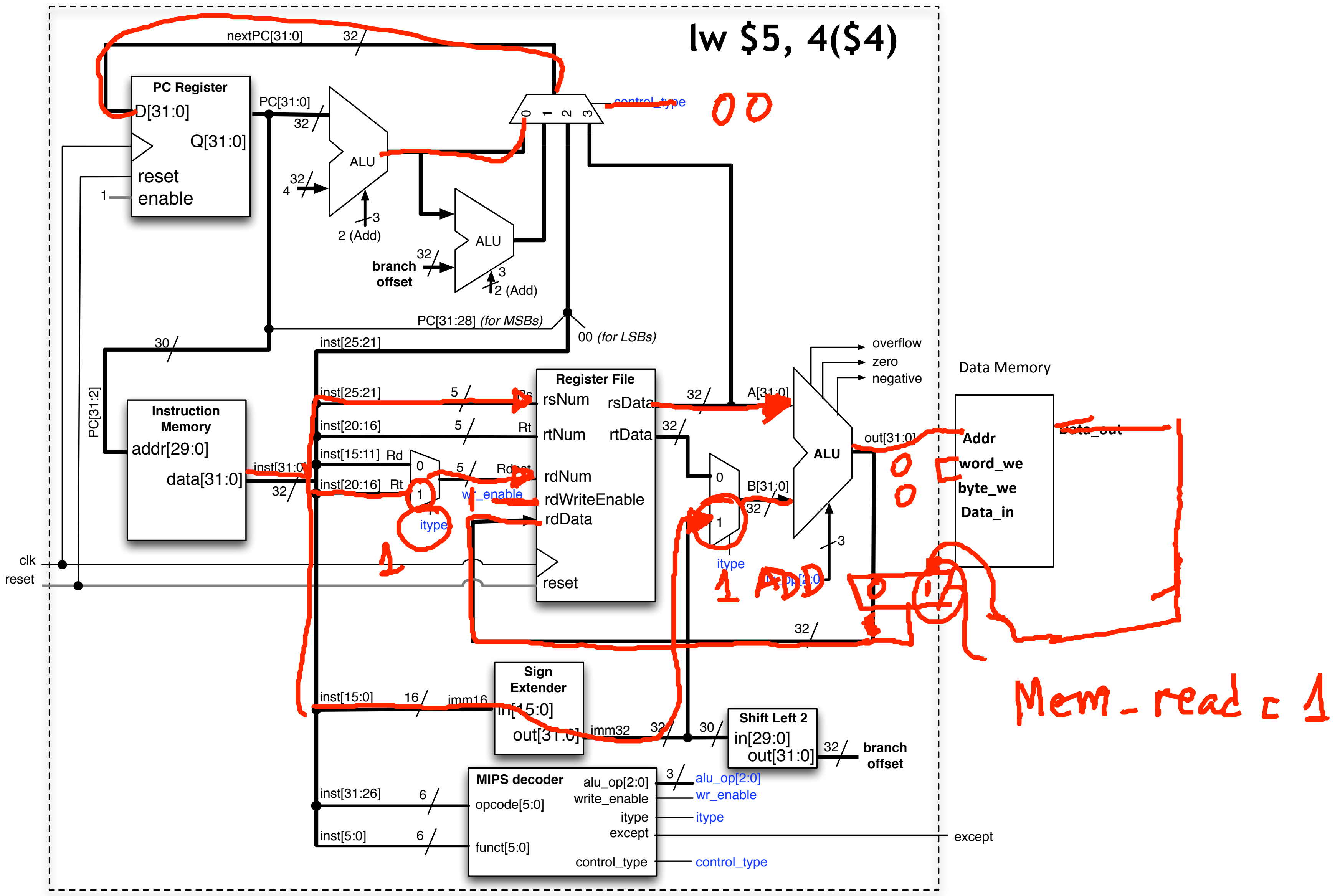
- **The address is a 16-bit signed two's-complement value.**
  - It can range from -32,768 to +32,767

# Enconding of loads and stores



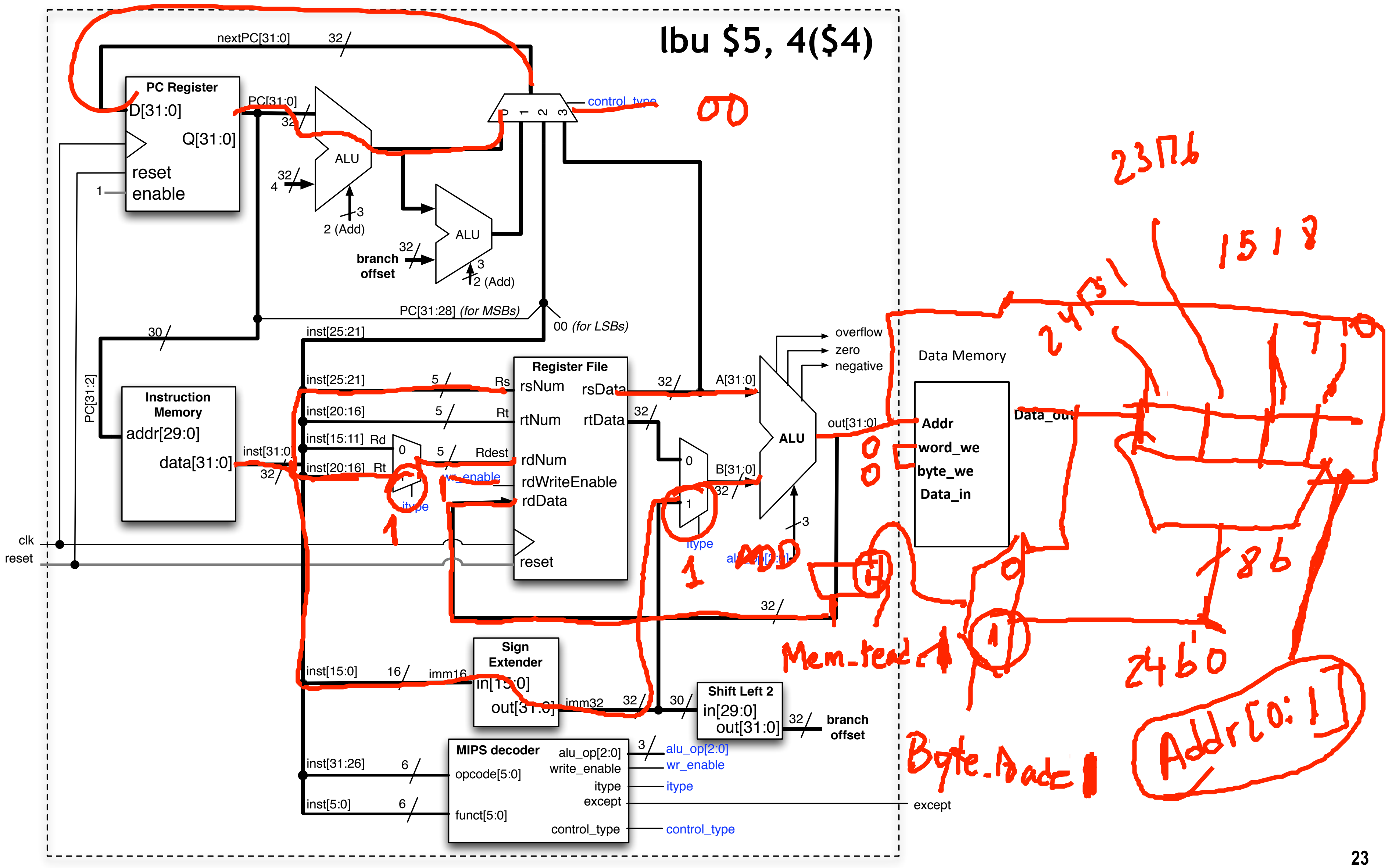


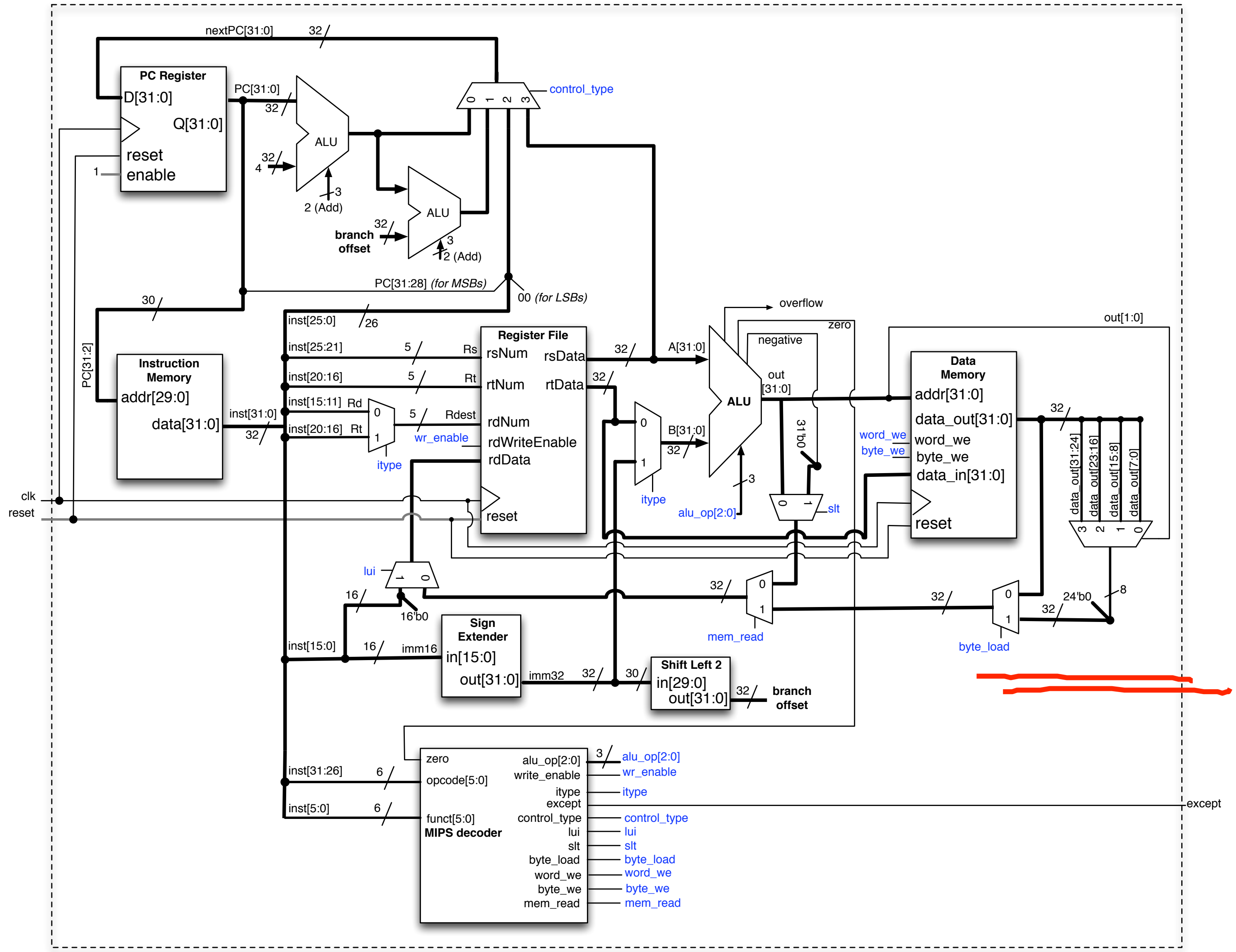
# load word implemented



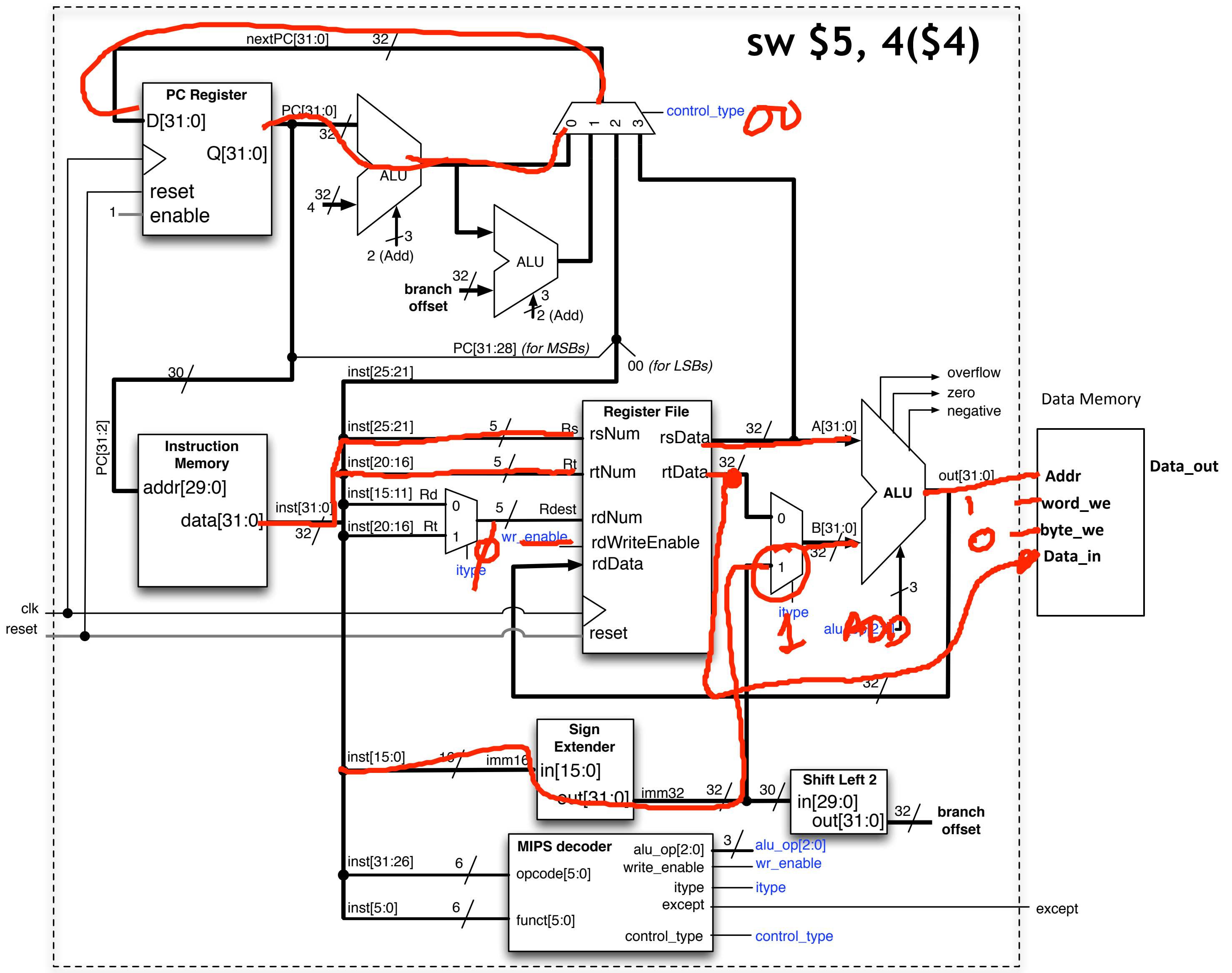
# load byte implemented

lbu $5, 4($4)

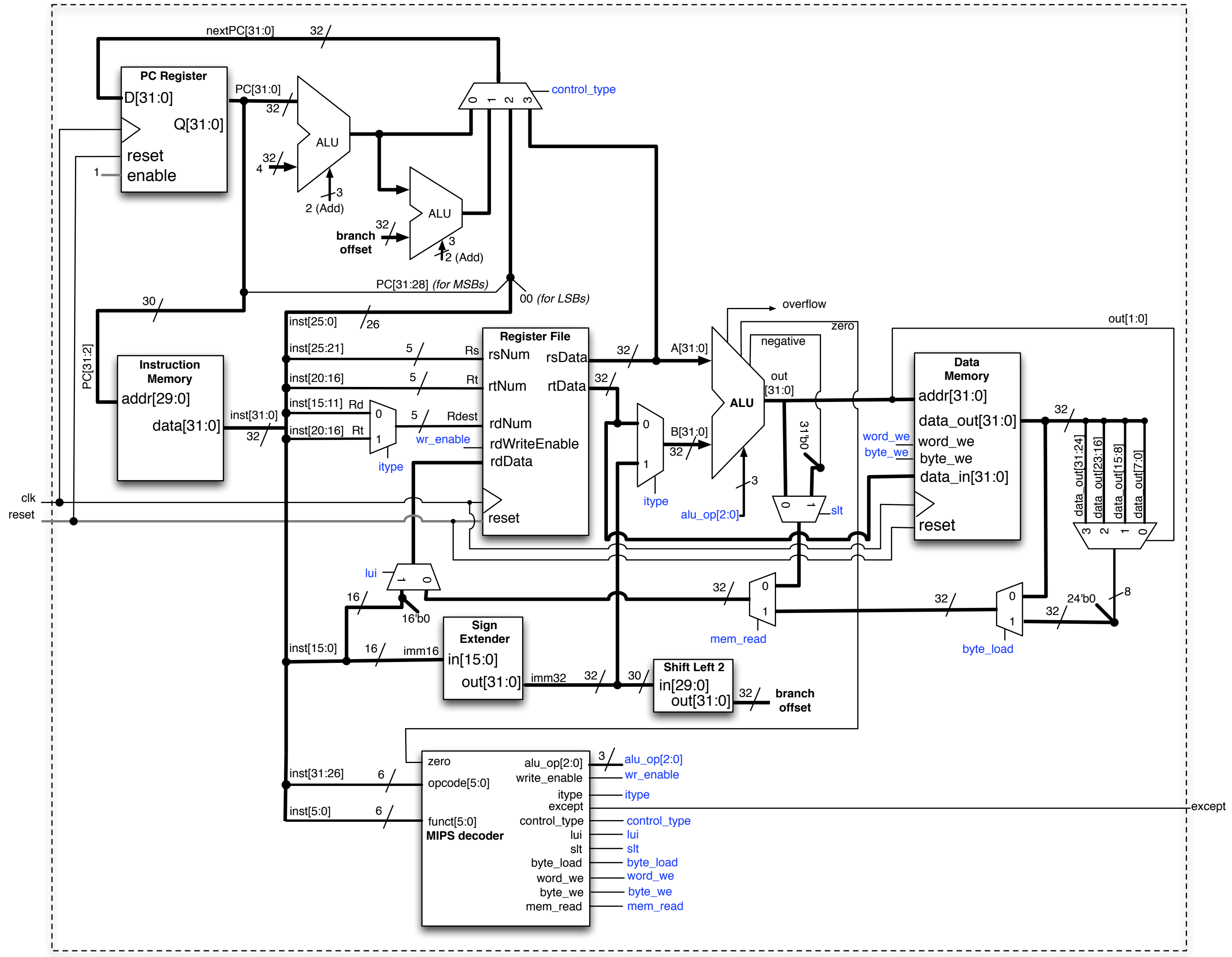



nextPC[31:0]
32
PC Register
D[31:0]
Q[31:0]
reset
enable
1
PC[31:0]
32
ALU
32
4
3
2 (Add)
control_type
0 1 2 3
ALU
32
branch offset
3
2 (Add)
PC[31:28] (for MSBs)
00 (for LSBs)
30
PC[31:2]
Instruction Memory
addr[29:0]
data[31:0]
inst[31:0]
32
inst[25:0]
26
inst[25:21]
5
Rs
inst[20:16]
5
Rt
inst[15:11]
Rd
inst[20:16]
Rt
0
1
5
Rdest
wr_enable
itype
Register File
rsNum
rsData
rtNum
rtData
rdNum
rdWriteEnable
rdData
reset
32
32
A[31:0]
B[31:0]
32
itype
ALU
overflow
zero
negative
out [31:0]
31'b0
3
alu_op[2:0]
slt
word_we
byte_we
out[1:0]
Data Memory
addr[31:0]
data_out[31:0]
word_we
byte_we
data_in[31:0]
reset
32
data_out[31:24]
data_out[23:16]
data_out[15:8]
data_out[7:0]
3 2 1 0
8
24'b0
32
32
byte_load
32
mem_read
clk
reset
lui
16
16'b0
inst[15:0]
16
imm16
Sign Extender
in[15:0]
out[31:0]
imm32
32
30
Shift Left 2
in[29:0]
out[31:0]
32
branch offset
inst[31:26]
6
inst[5:0]
6
zero
opcode[5:0]
funct[5:0]
MIPS decoder
alu_op[2:0]
write_enable
itype
except
control_type
lui
slt
byte_load
word_we
byte_we
mem_read
3
alu_op[2:0]
wr_enable
itype
except
control_type
lui
slt
byte_load
word_we
byte_we
mem_read

# store implemented



# Full Machine Datapath – Lab 6


nextPC[31:0]
32
PC Register
D[31:0]
Q[31:0]
reset
enable
1
PC[31:0]
32
4
ALU
3
2 (Add)
ALU
branch offset
3
2 (Add)
0 1 2 3
control_type
PC[31:28] (for MSBs)
00 (for LSBs)
30
PC[31:2]
Instruction Memory
addr[29:0]
data[31:0]
inst[31:0]
inst[25:0]
26
inst[25:21]
inst[20:16]
inst[15:11]
inst[20:16]
Rs
Rt
Rd
Rt
Rdest
5
wr_enable
itype
Register File
rsNum
rtNum
rdNum
rdWriteEnable
rdData
rsData
rtData
reset
A[31:0]
B[31:0]
ALU
alu_op[2:0]
overflow
zero
negative
out
[31:0]
31'b0
slt
word_we
byte_we
out[1:0]
Data Memory
addr[31:0]
data_out[31:0]
word_we
byte_we
data_in[31:0]
reset
data_out[31:24]
data_out[23:16]
data_out[15:8]
data_out[7:0]
24'b0
8
byte_load
mem_read
lui
16
16'b0
inst[15:0]
Sign Extender
in[15:0]
out[31:0]
imm16
imm32
Shift Left 2
in[29:0]
out[31:0]
branch offset
clk
reset
inst[31:26]
inst[5:0]
6
zero
opcode[5:0]
funct[5:0]
MIPS decoder
alu_op[2:0]
write_enable
itype
except
control_type
lui
slt
byte_load
word_we
byte_we
mem_read
except