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.

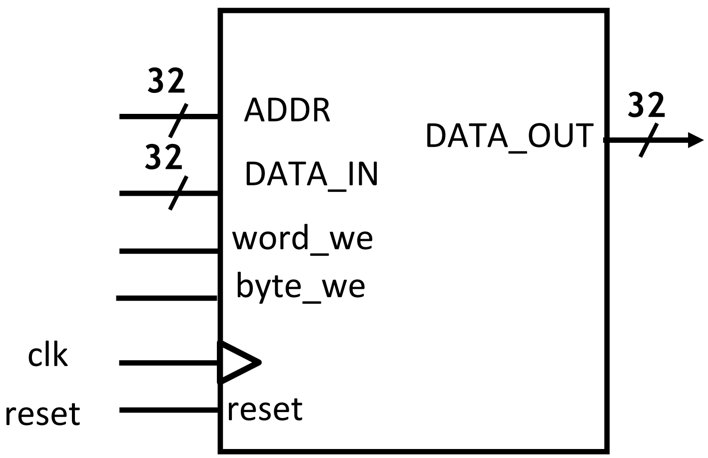
Harvard Architecture

- 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

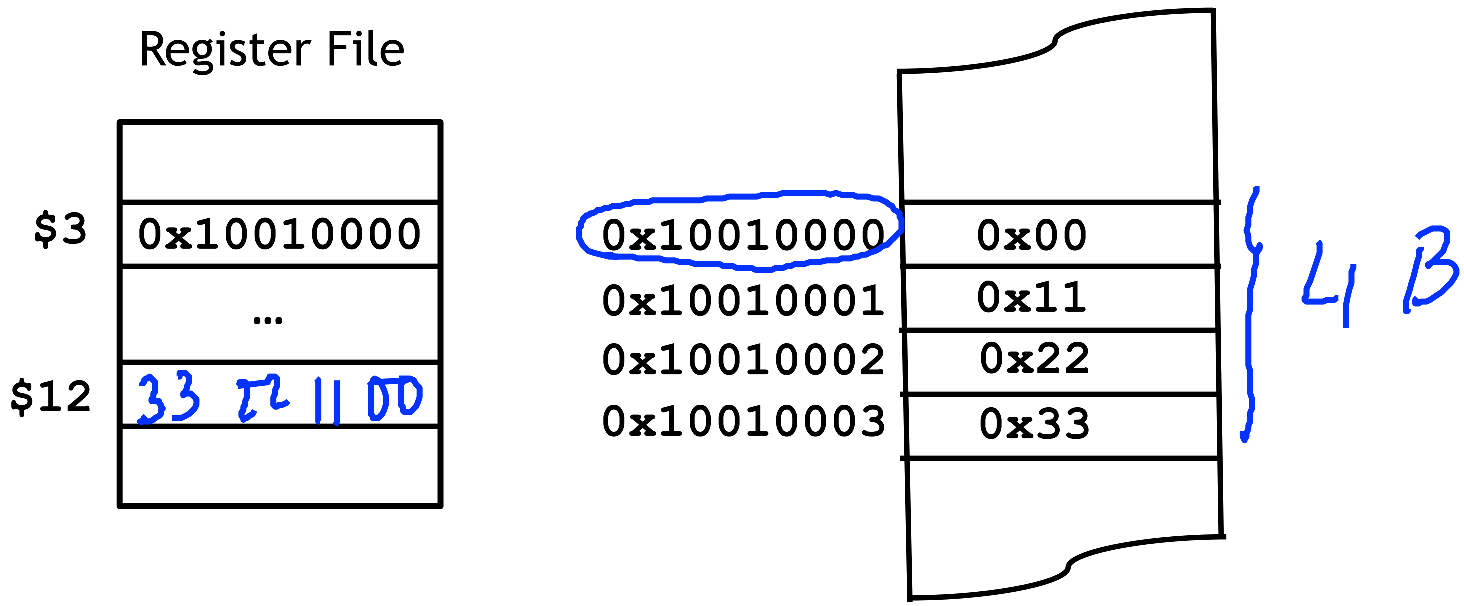
$12 = Memory[0+[$3]]

Address = 0+ 0x10010000

`lw $12, 0($3)`

word = 32 bits = 4 B

Data Memory



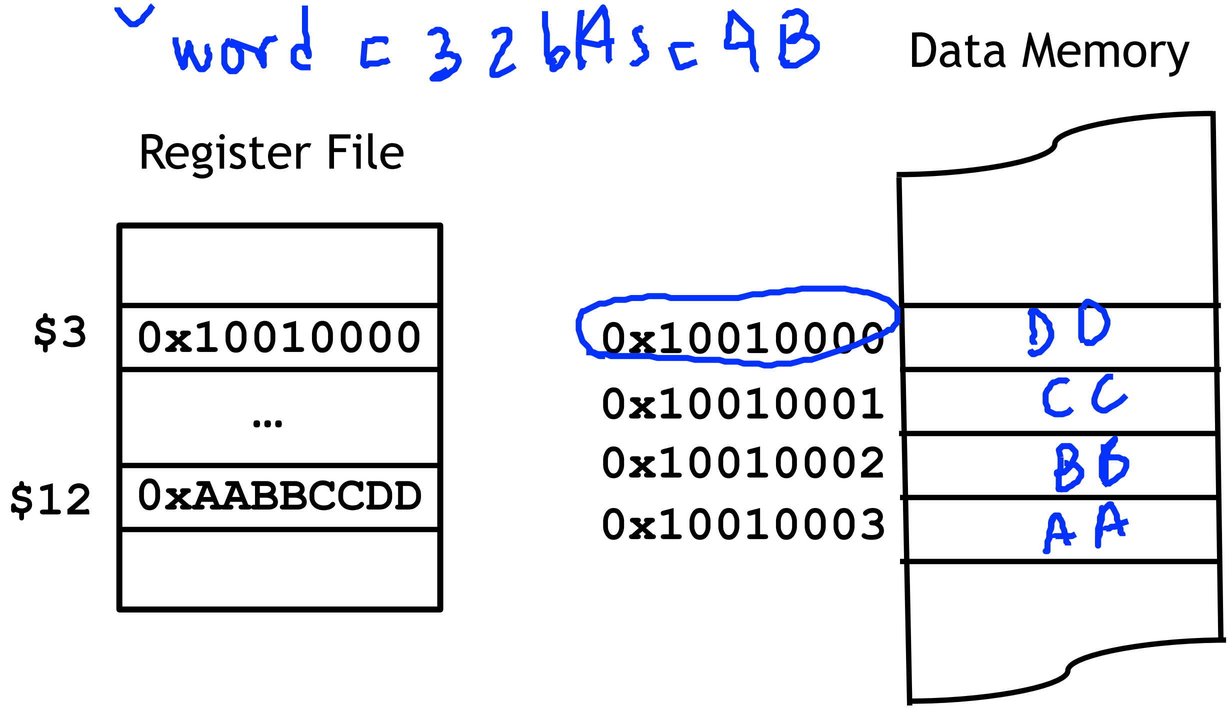
Example

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]]

zero extend

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

LSB

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

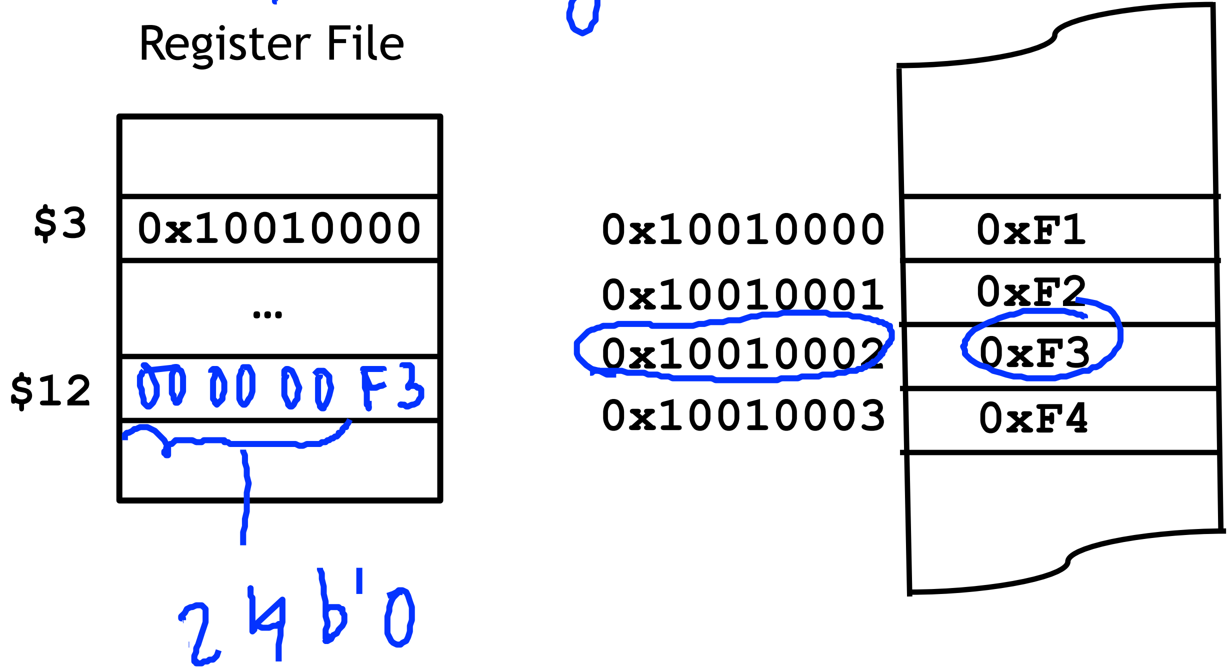
Memory [2 + R [$3]] = $12

Example

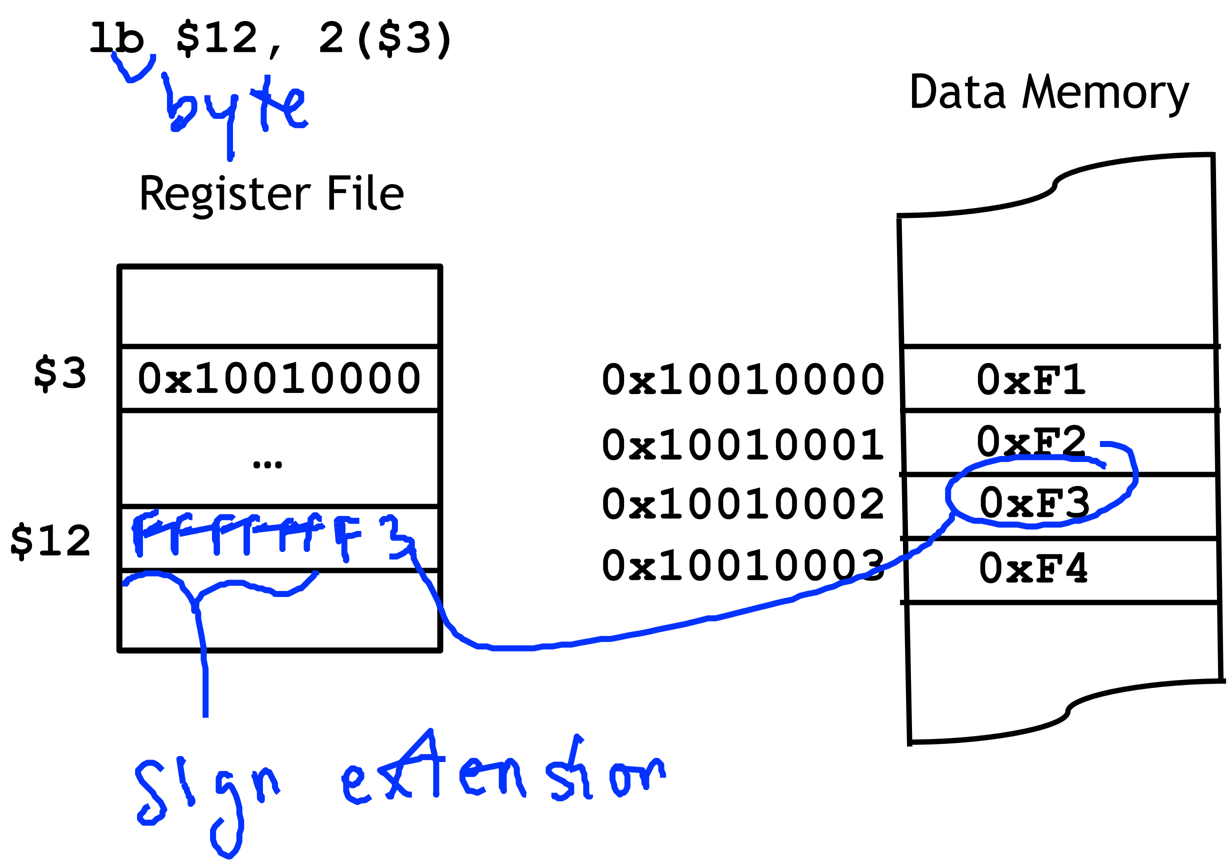
$\$12 = Memory[2 + R[\$3]]$

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

byte unsigned



Example



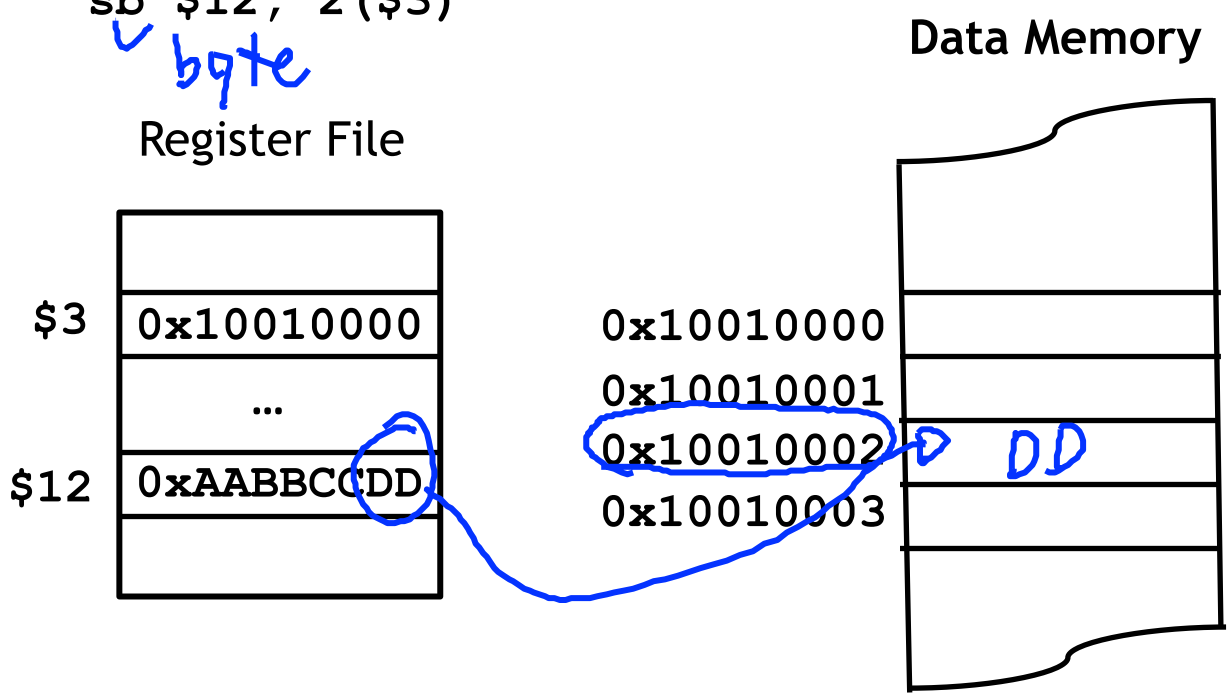
lb $12, 2($3)
byte
Register File
Data Memory
$3
0x10010000
…
$12
~~FFFFFF~~ FFF3
0x10010000
0x10010001
0x10010002
0x10010003
0xF1
0xF2
0xF3
0xF4
sign extension

Example

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

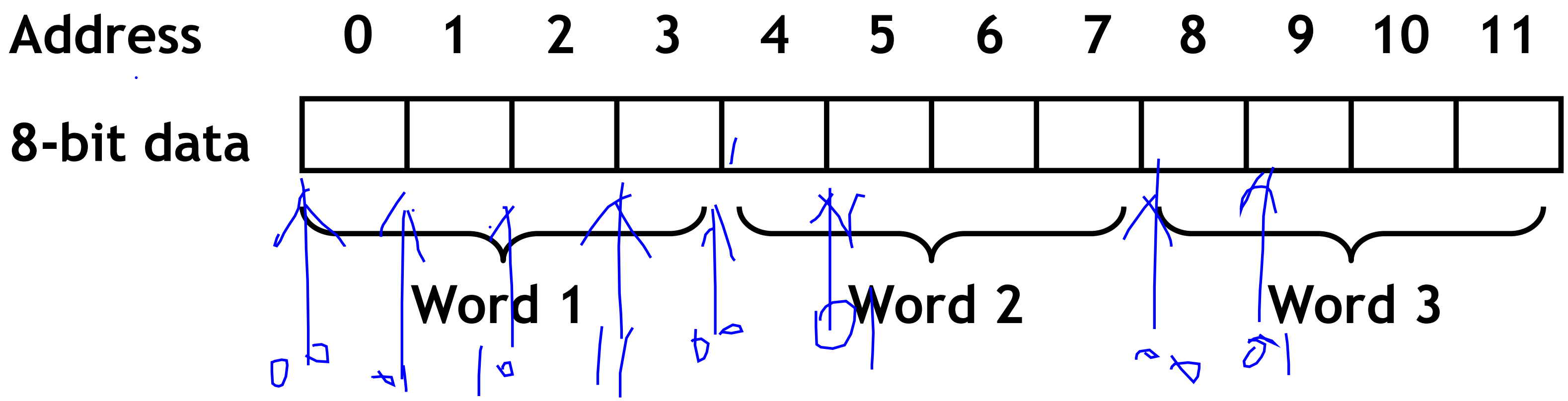
```
sb $12, 2($3)
```

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

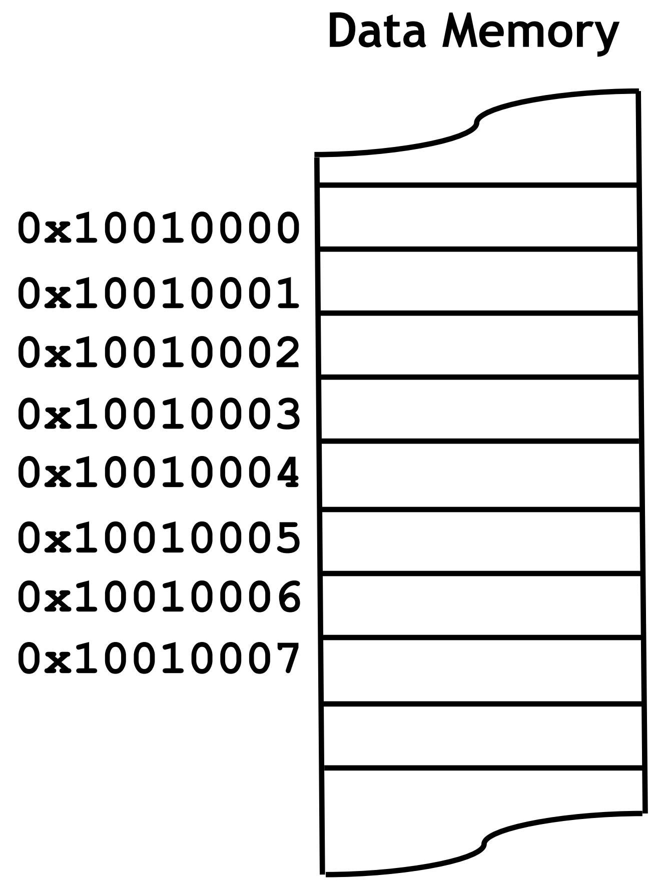
```
int a = 10;
int b = 0;
void main() {
   b = a+7;
}
```

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

Example Program that Uses Memory

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

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



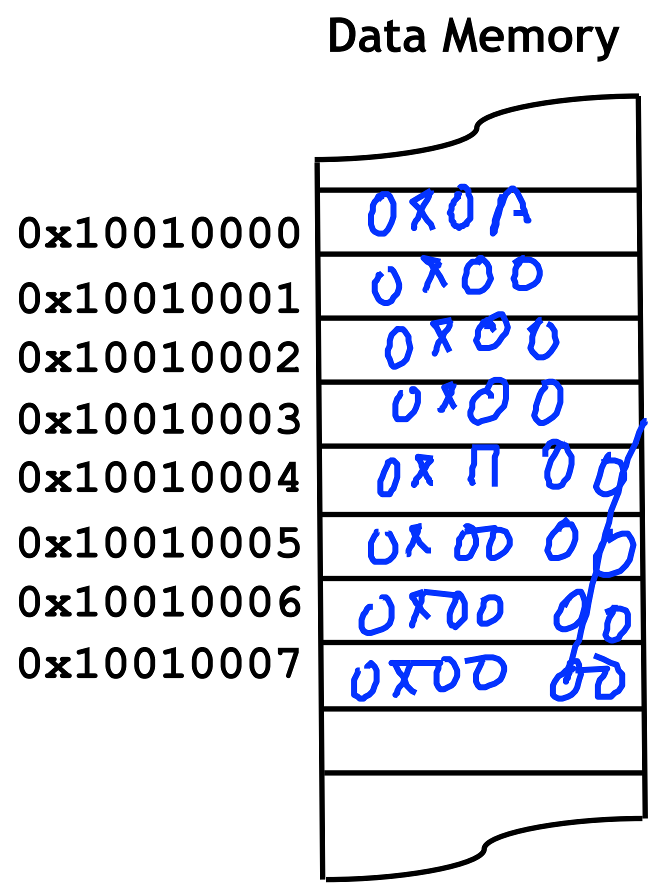
Example Program that Uses Memory

iclicker

```
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)
```

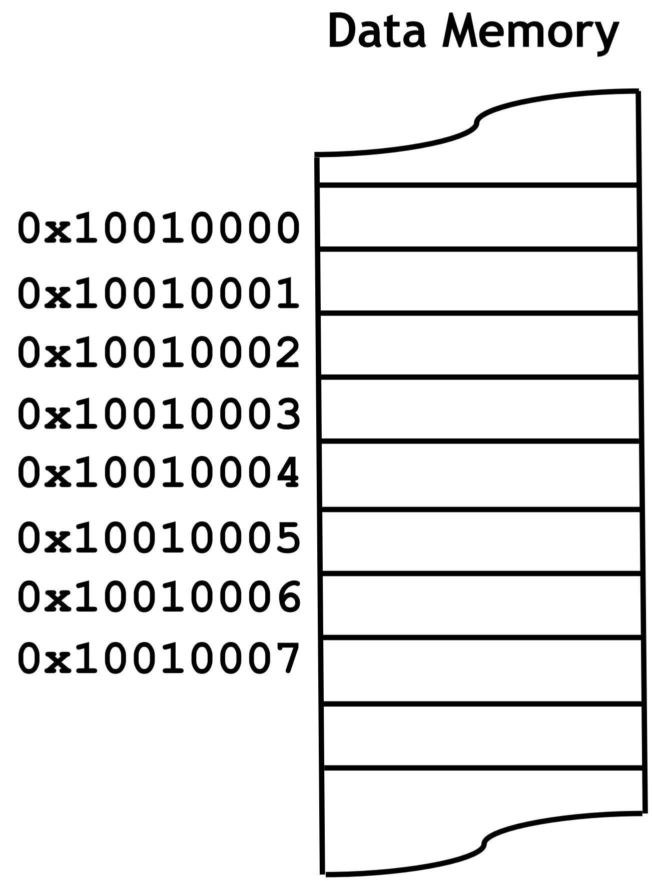
C

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

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

sw $rt off( )

- 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

lw $5, 4($4)

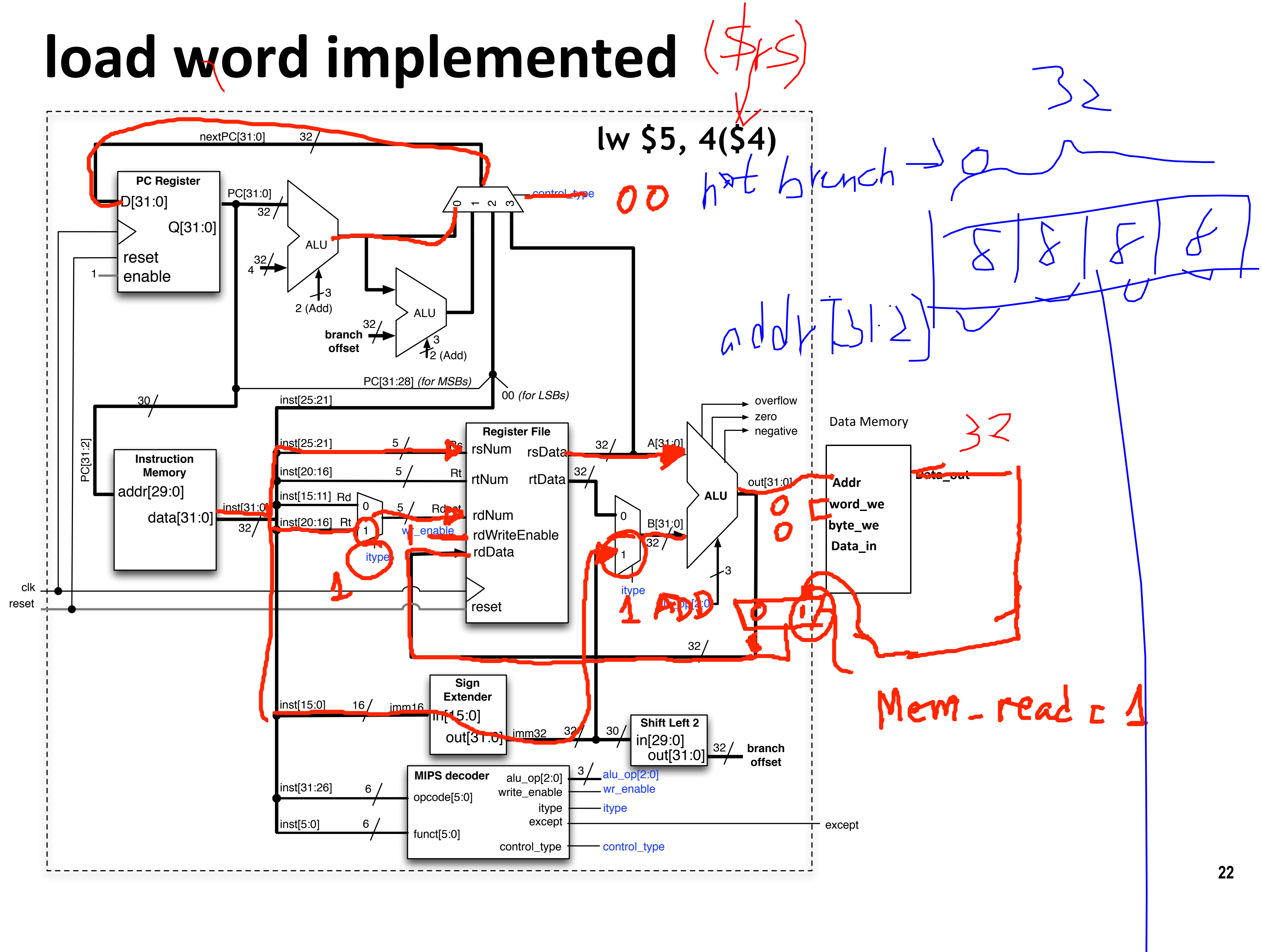
op	rs	rt	address

sw $5, 4($4)

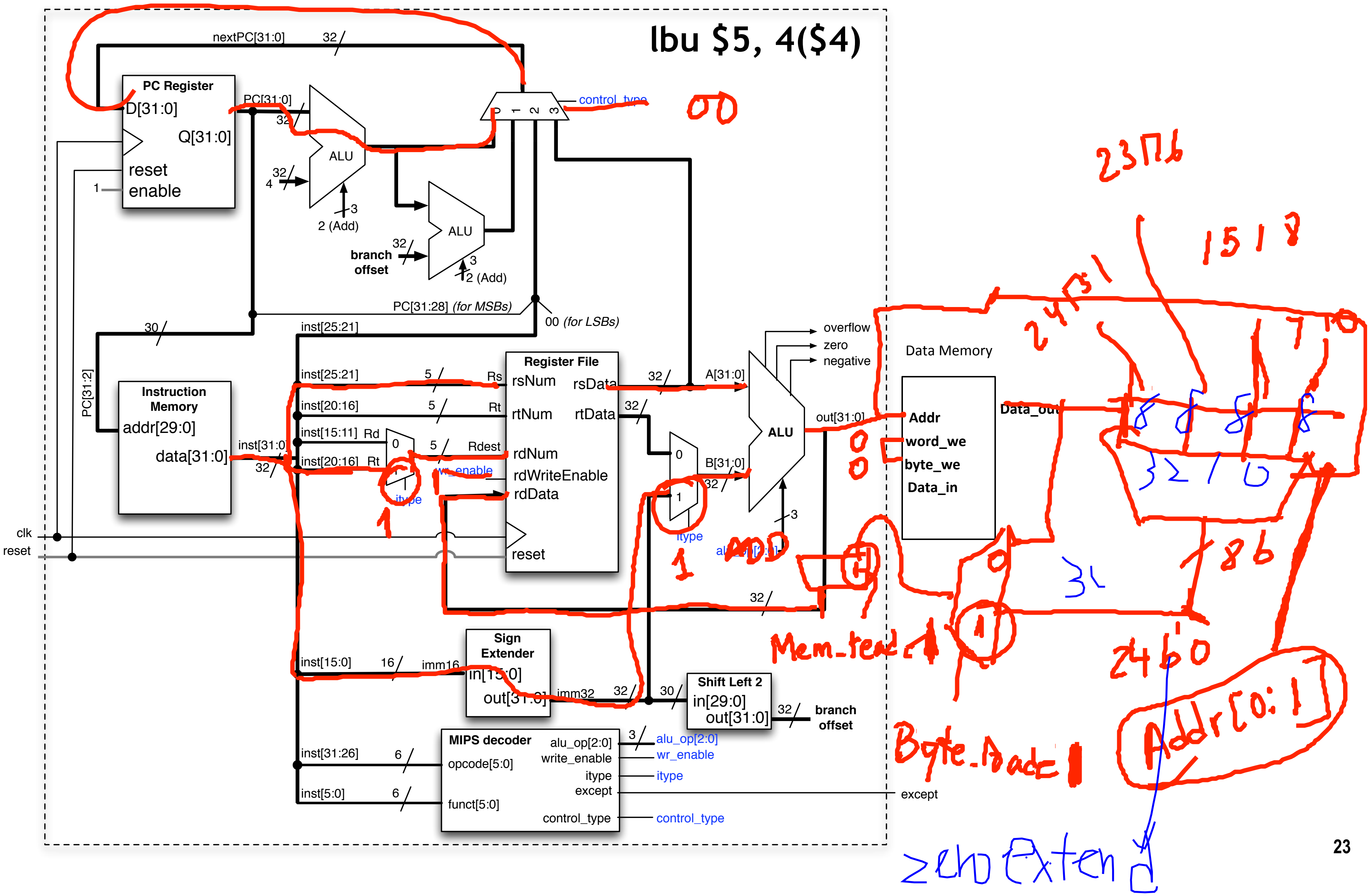
op	rs	rt	address

load word implemented

lw $5, 4($4)



load byte implemented

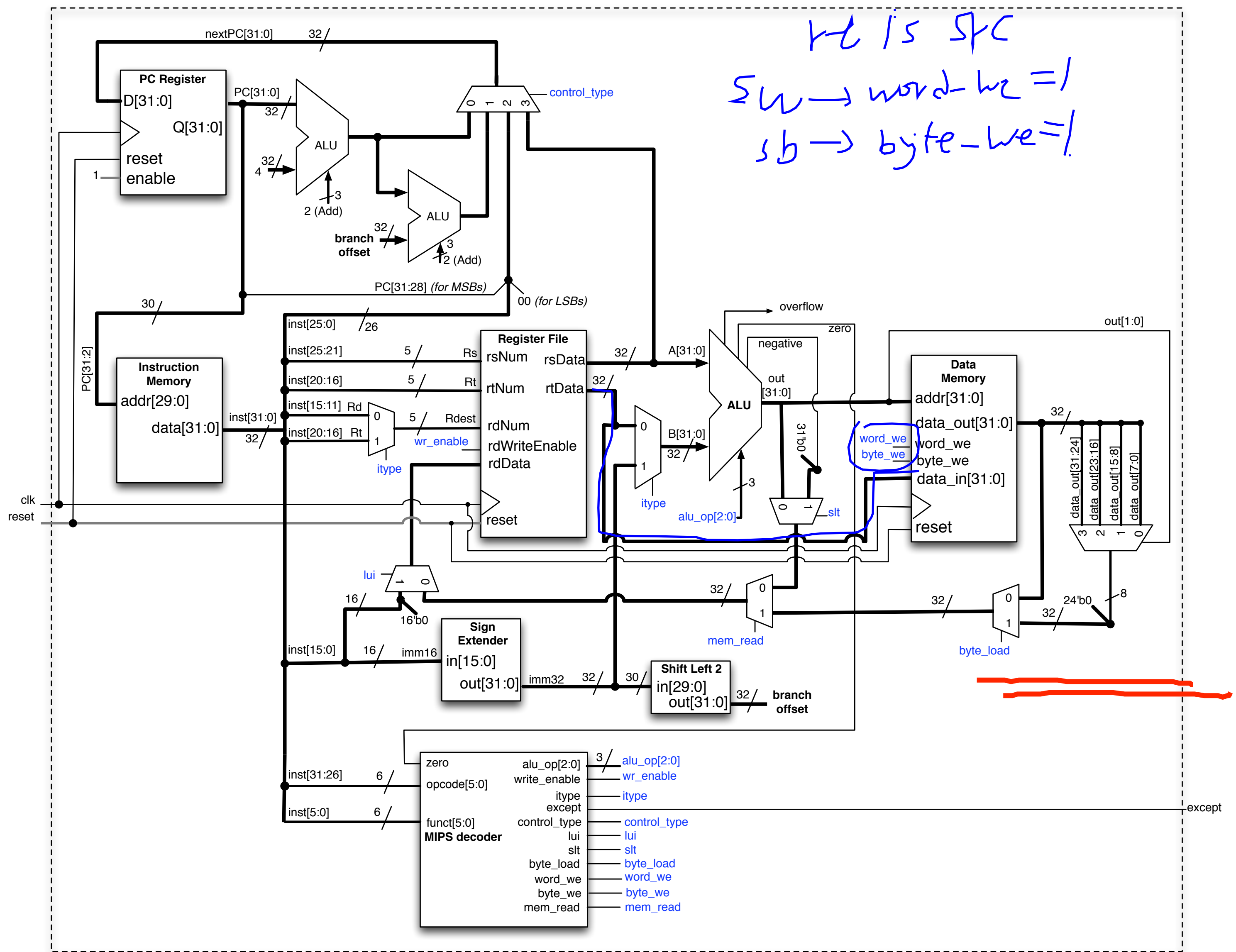


sw/J.

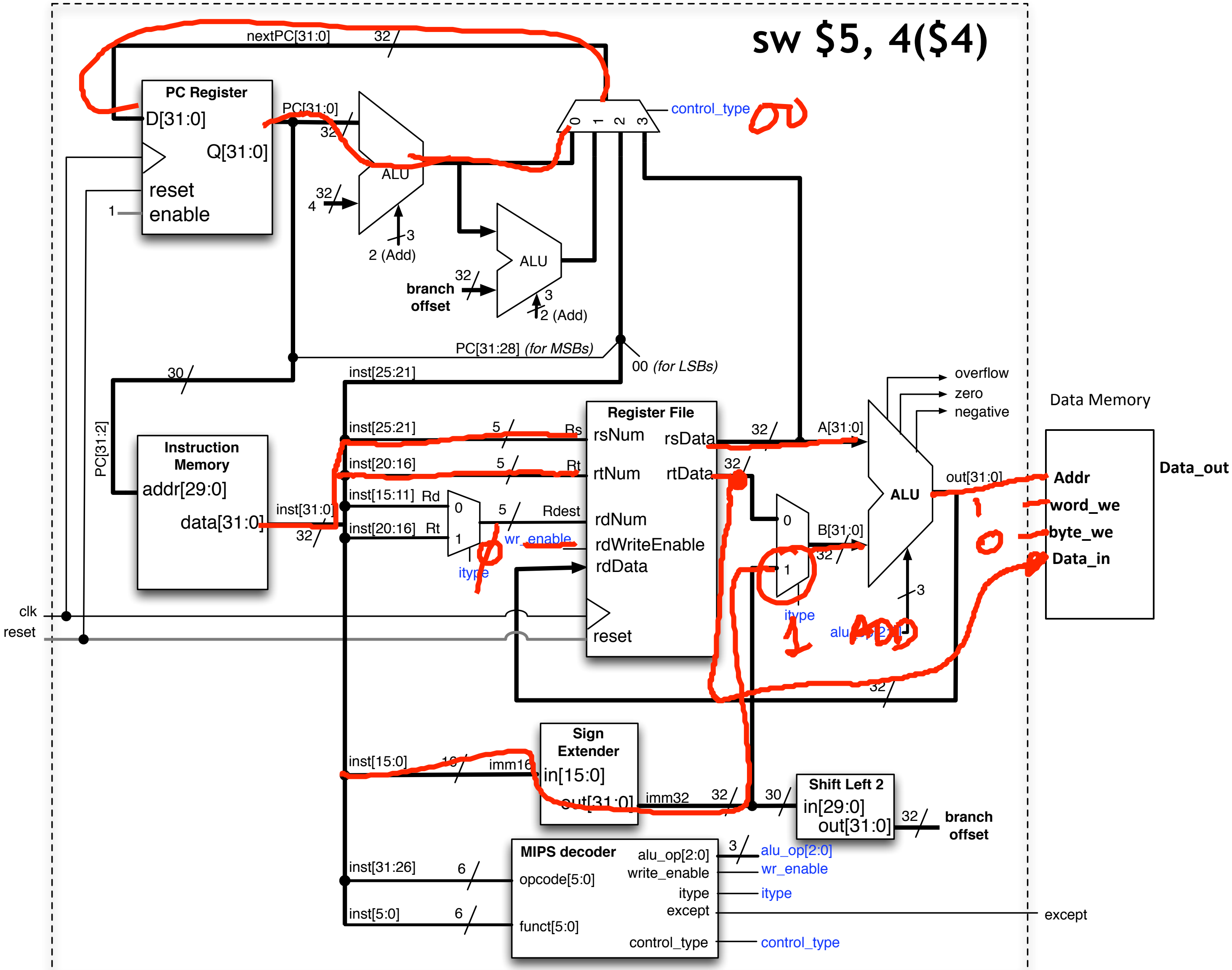
rt is src

sw → word_we = 1

sb → byte_we = 1



store implemented



Full Machine Datapath – Lab 6