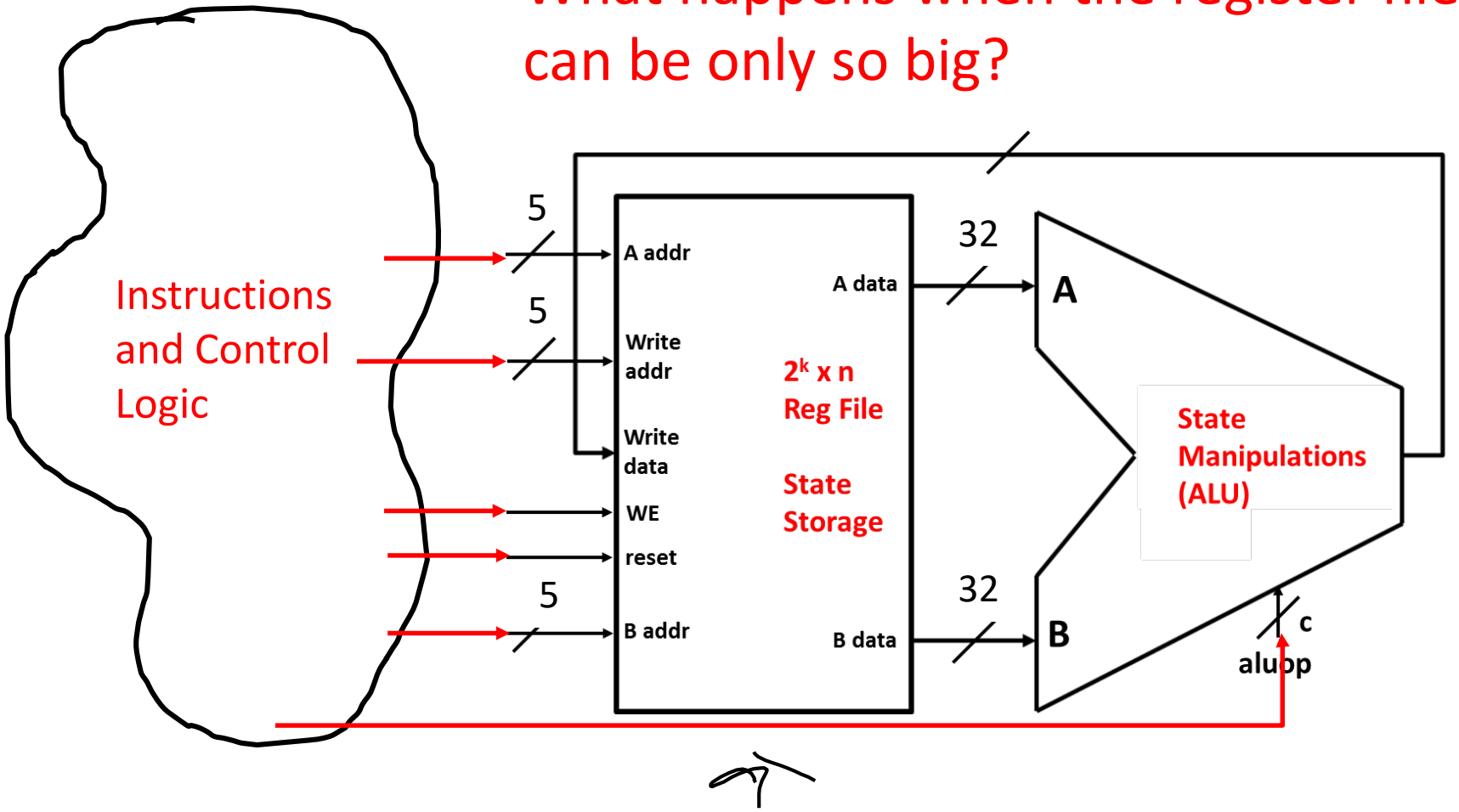
MIPS Load & Stores

Today's lecture

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

State – the central concept of computing

What happens when the register file can be only so big?



We need more space!

Registers	Main Memory
Fast	Slow
Synchronous	Not always synchronous
Small (32x32 bits)	Large (2^{32}B)
Expensive	Cheap-ish

Harvard Architecture stores programs and data in *separate* memories

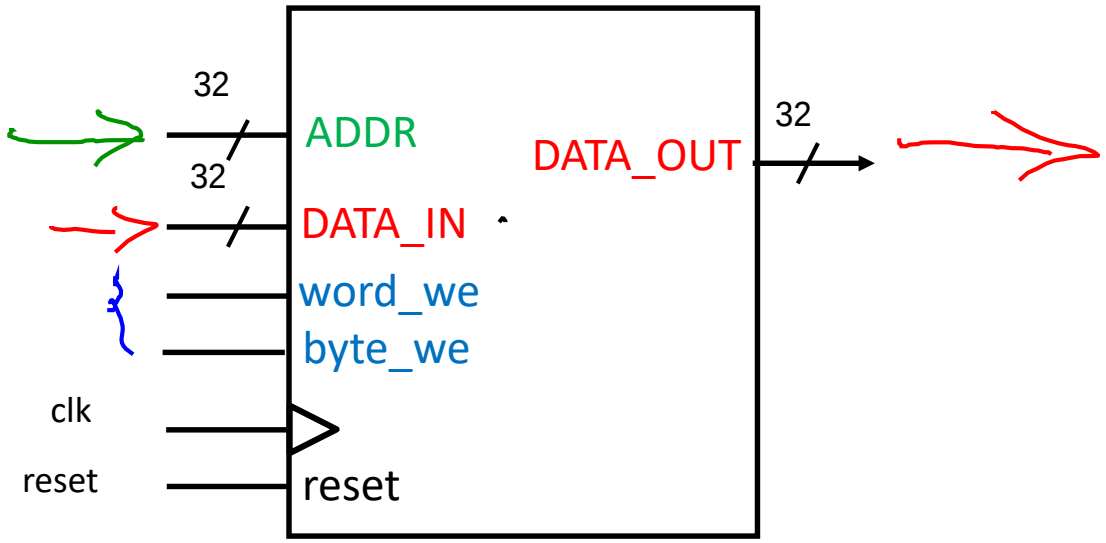
Instruction memory:

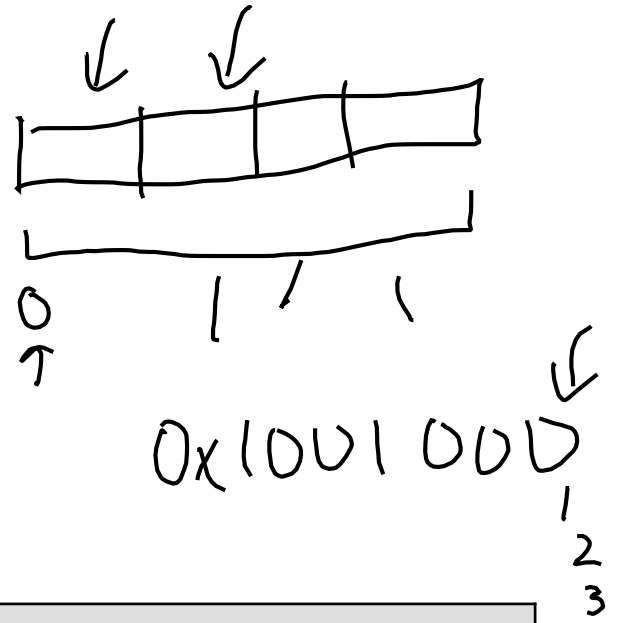
- Contains instructions to execute
- Treat as read-only

Data memory:

- Contains the data of the program
- Can be read/written

Data Memory is byte-addressable with 2^{32} bytes





word_we	byte_we	Operation	
0	0	Read (Load)	DATA_OUT = M[ADDR]
0	1	Write (Store) byte in ADDR	M[ADDR] = DATA_IN[7:0]
1	0	Write (Store) word in ADDR	M[ADDR]† = DATA_IN[31:0]

32-bit

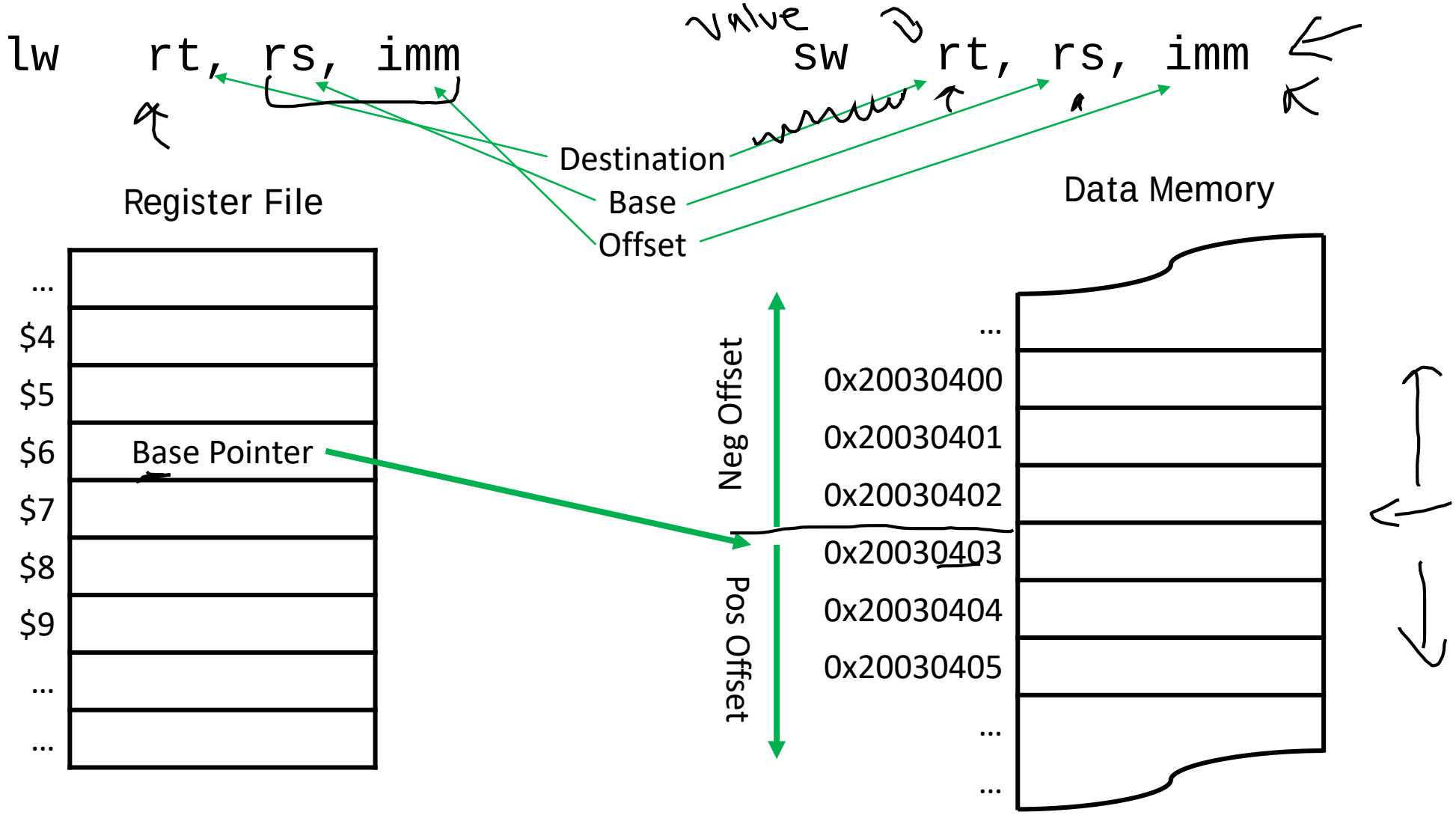
†(ADDR[1:0] must be word aligned)

We can load or store bytes or words

	Load	Store
Word	`lw` R[rt] = M[ADDR][31:0]	`sw` M[ADDR] = R[rs][31:0]

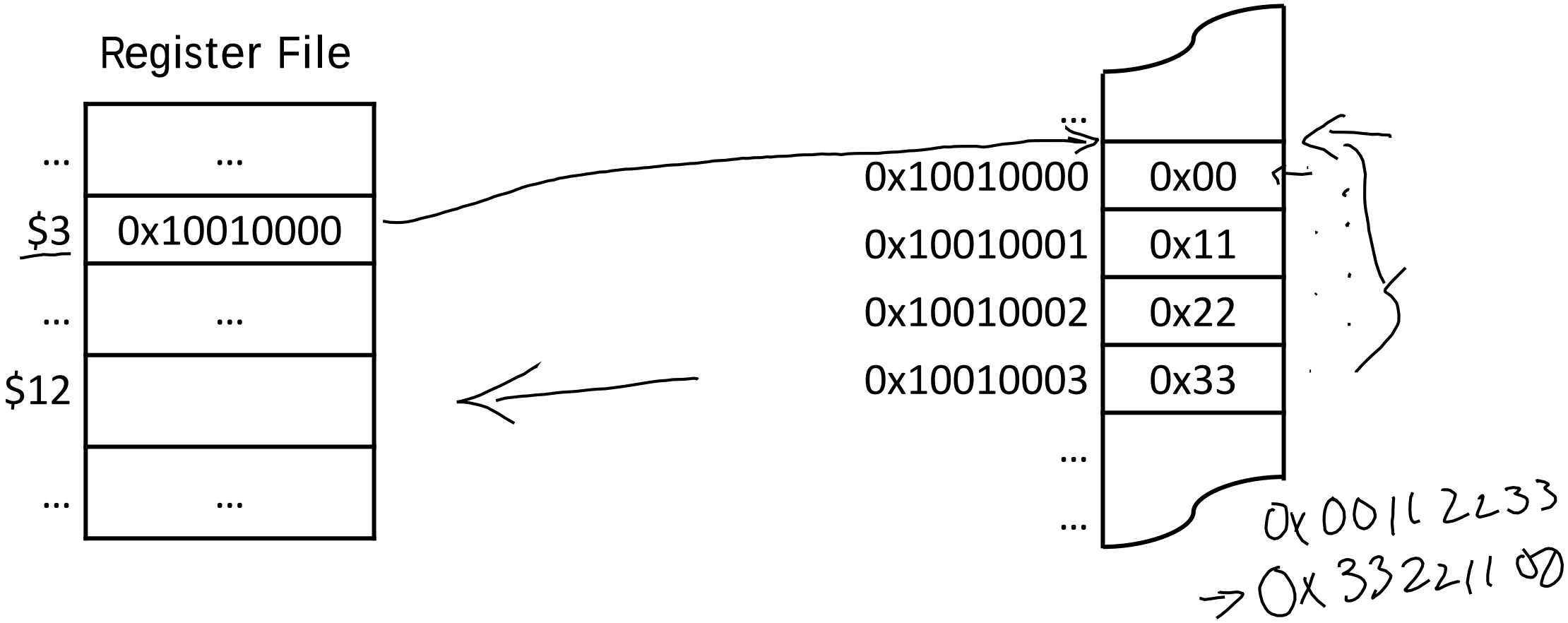
Byte	`lb`
	R[rt] = SEXT(M[ADDR][7:0])
	`lbu`
	R[rt] = ZEXT(M[ADDR][7:0])
		`sb`
		M[ADDR] = R[rs][7:0]

Indexed addressing derives ADDR from a base "pointer" register and a constant

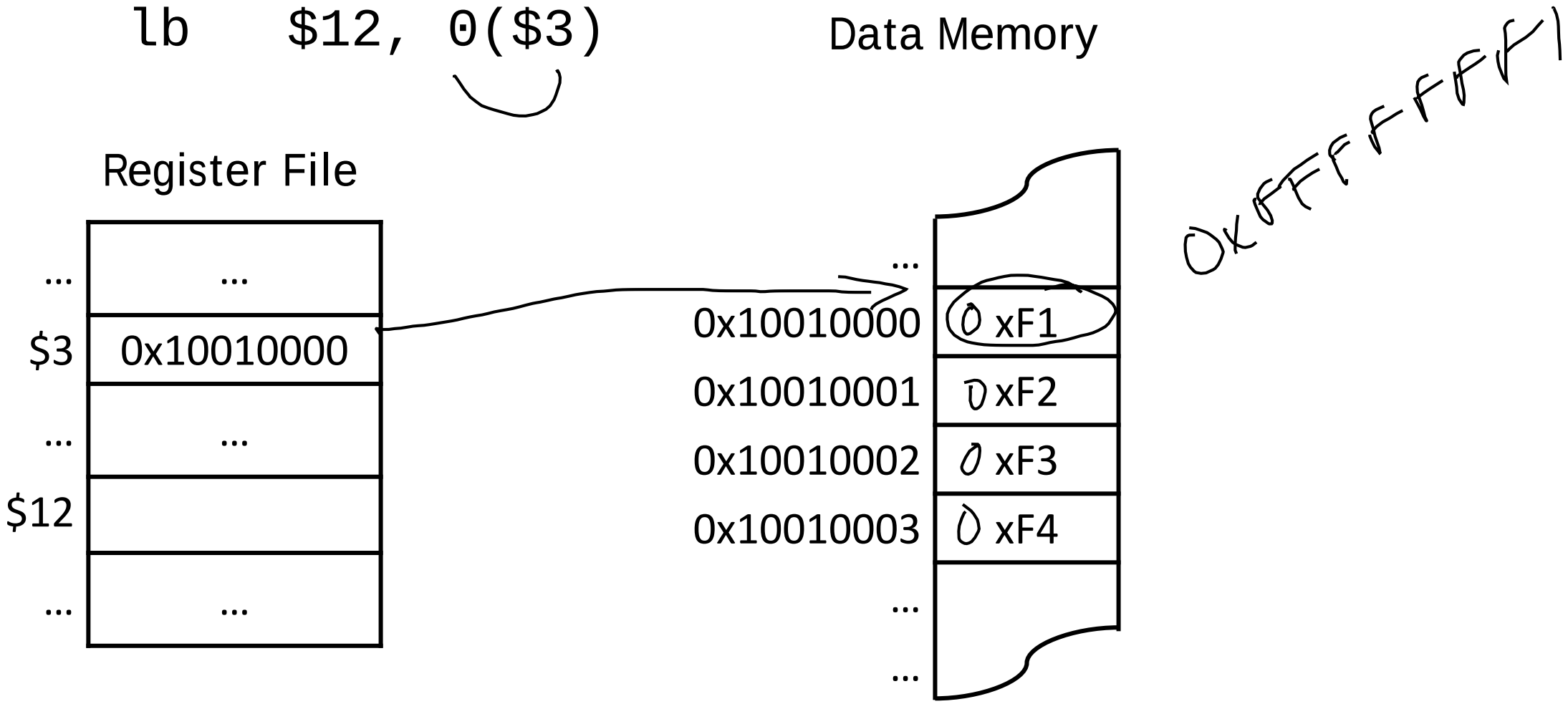


Load word (lw) is Little Endian

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



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


Register File
Data Memory
...
...
$3
0x10010000
...
...
$12
...
...
...
0x10010000
0x10010001
0x10010002
0x10010003
...
...
0 xF1
0 xF2
0 xF3
0 xF4
0xFFFFFFF1

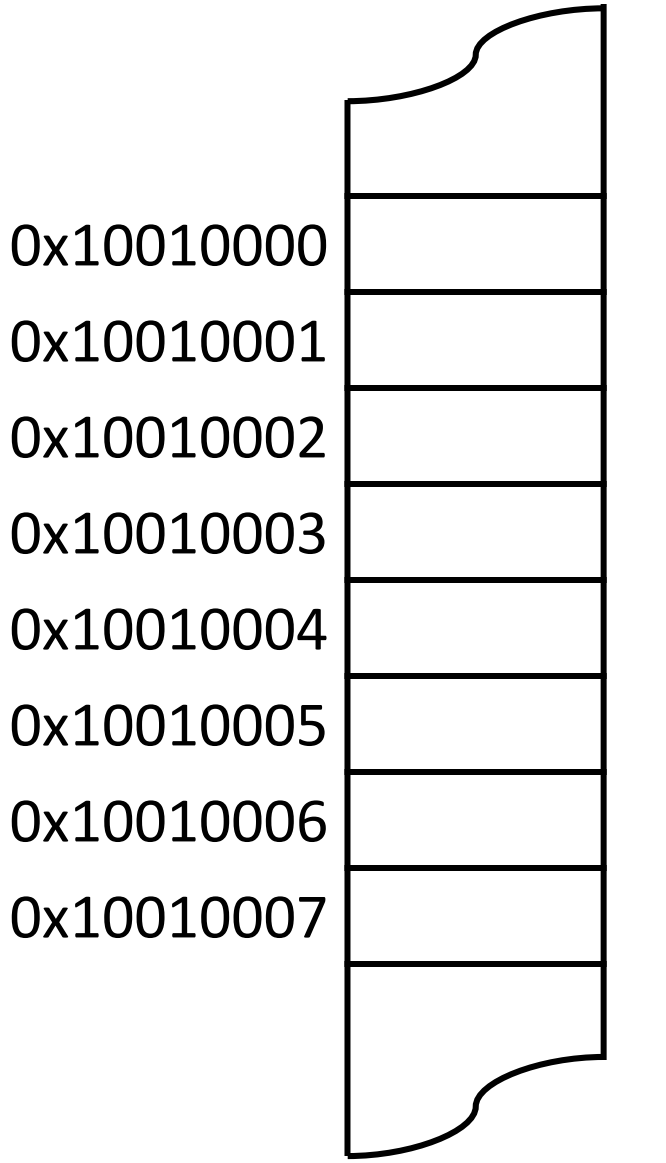
Convert C code into MIPS assembly

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

Convert C code into MIPS assembly

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

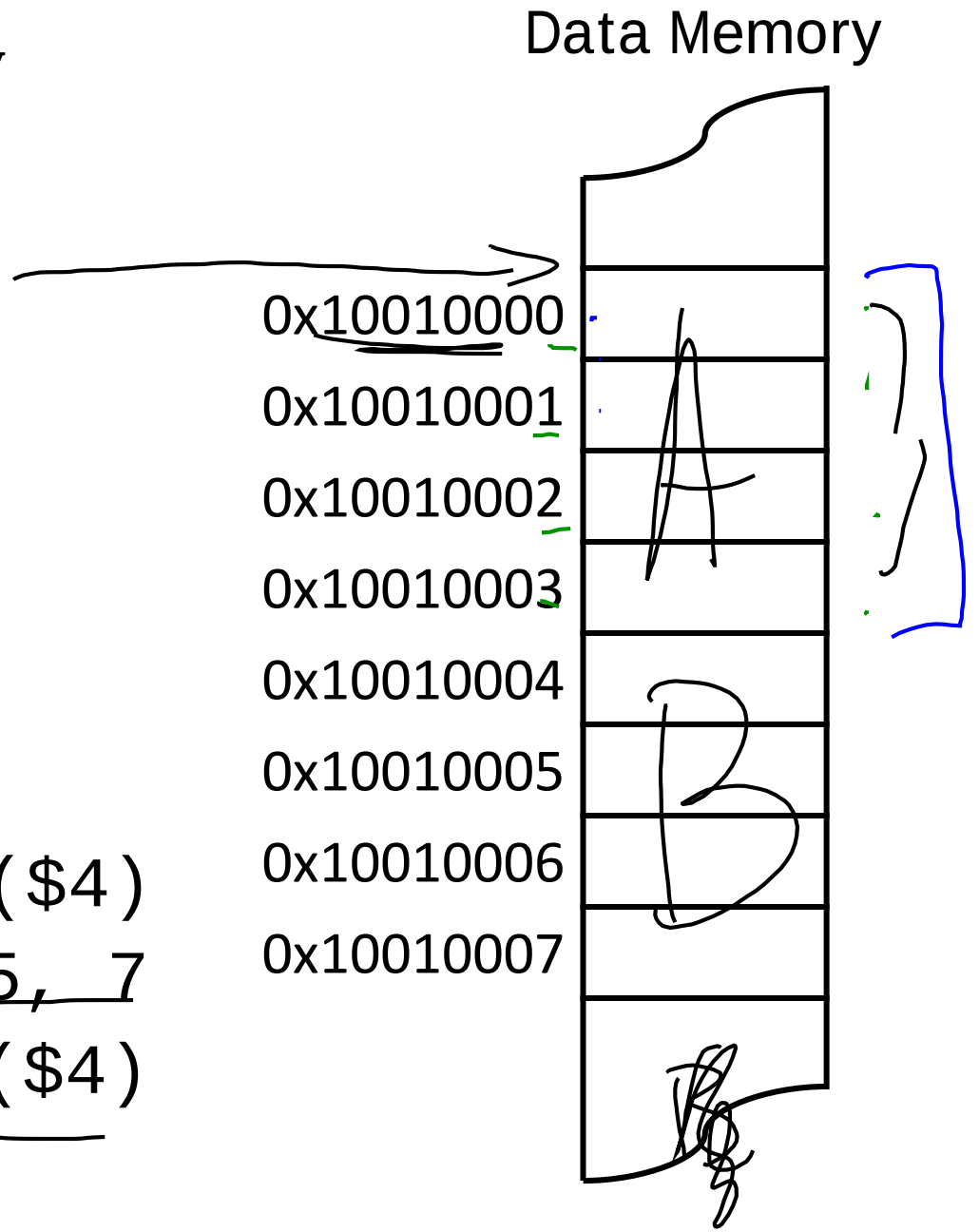
```
.data
a: .word 10
b: .word 0
.text
main:
    la   $4, a
```



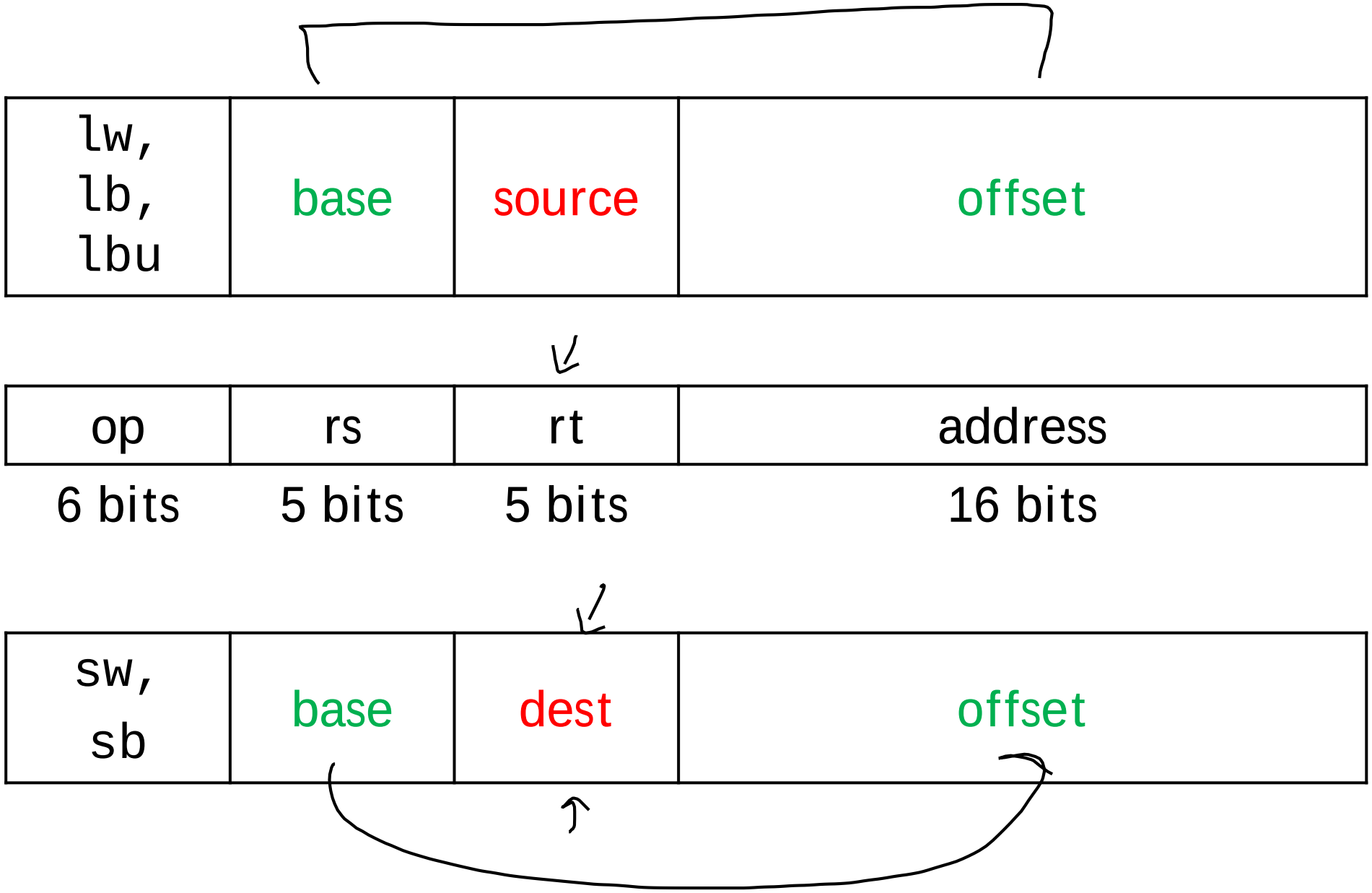
Convert C code into MIPS assembly

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

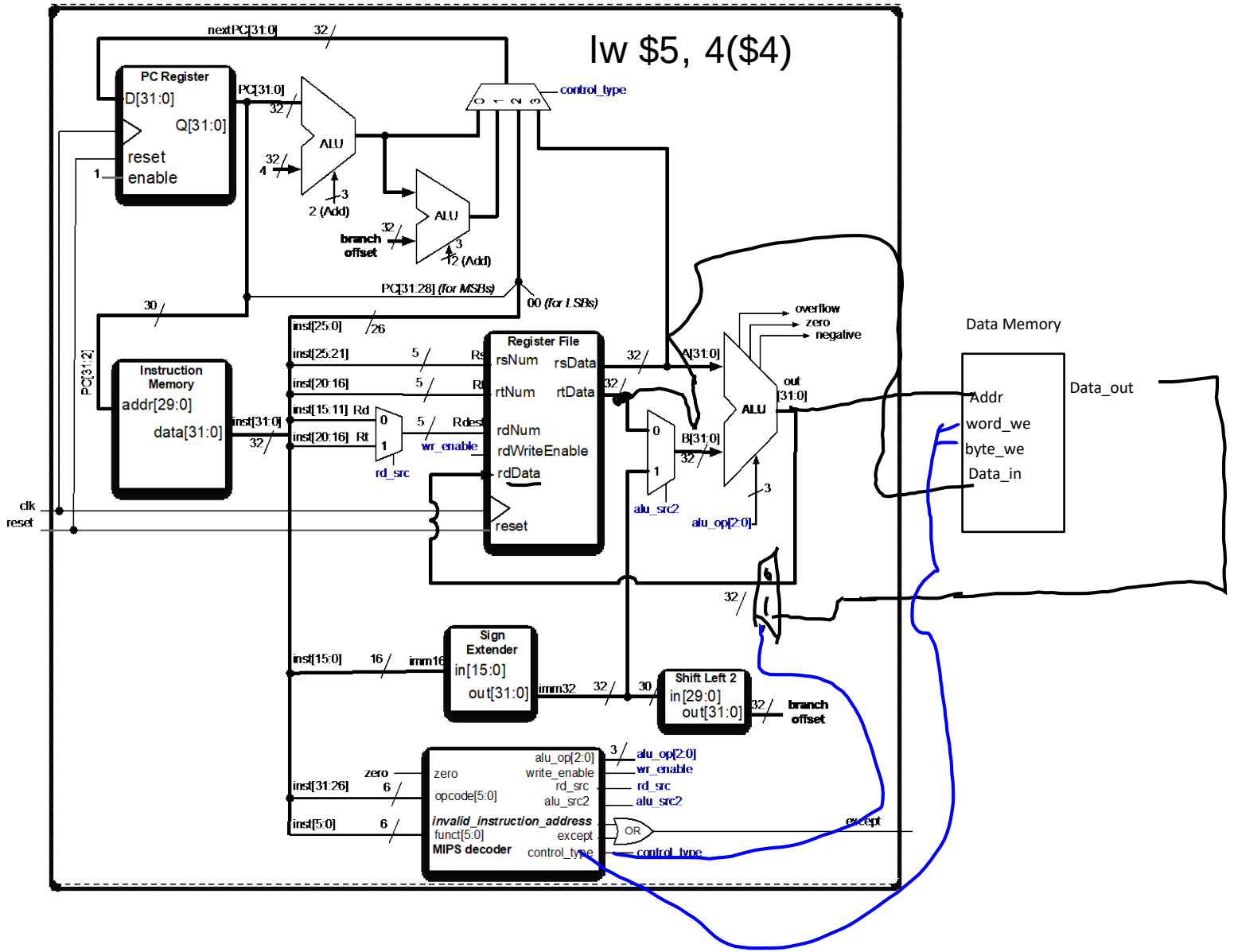
```
.data
a: .word 10
b: .word 0
.text
main:
    la   $4, a
    lw   $5, 0($4)
    addi $5, $5, 7
    sw   $5, 4($4)
```



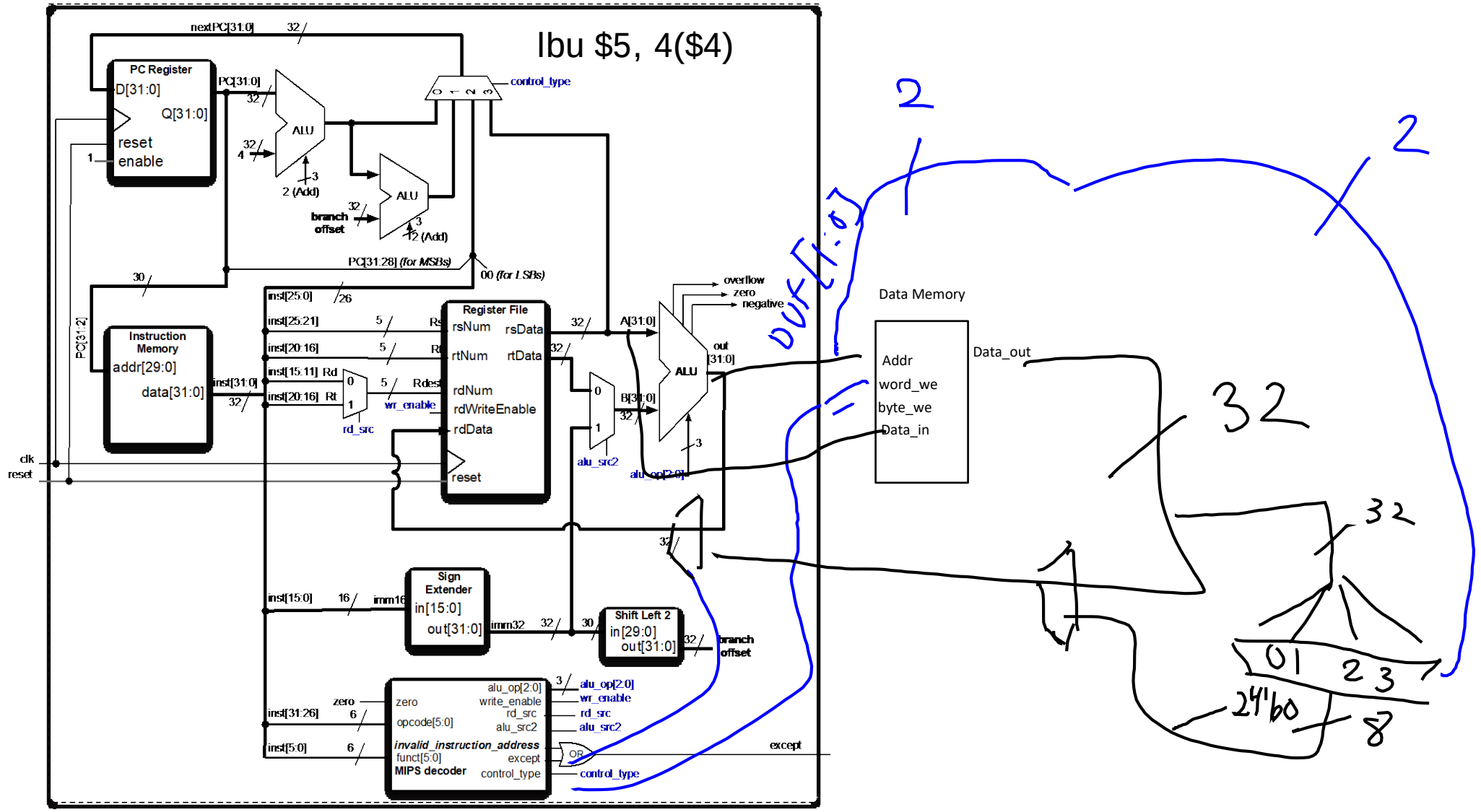
Loads and stores use the I-type format

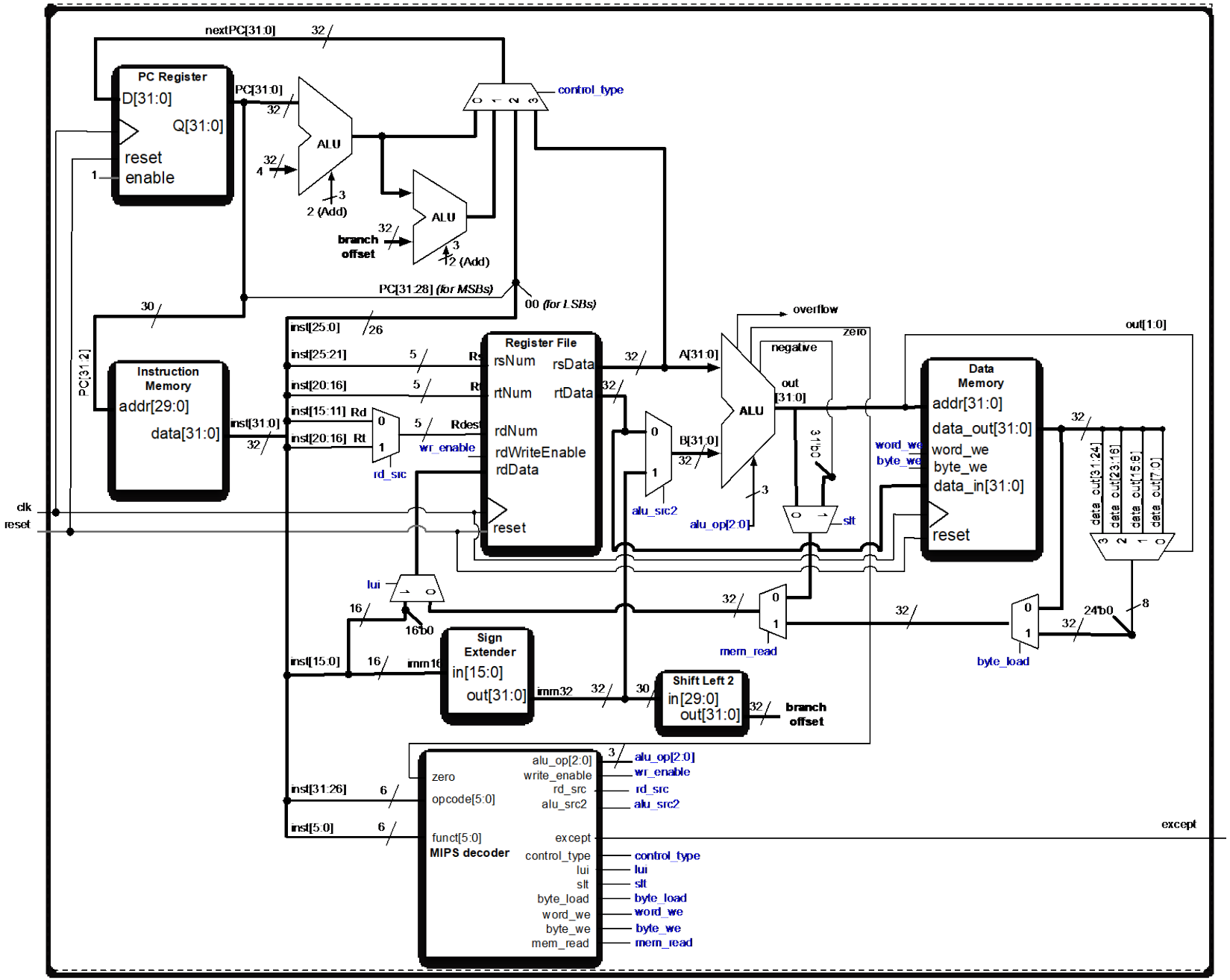


load word **implemented**

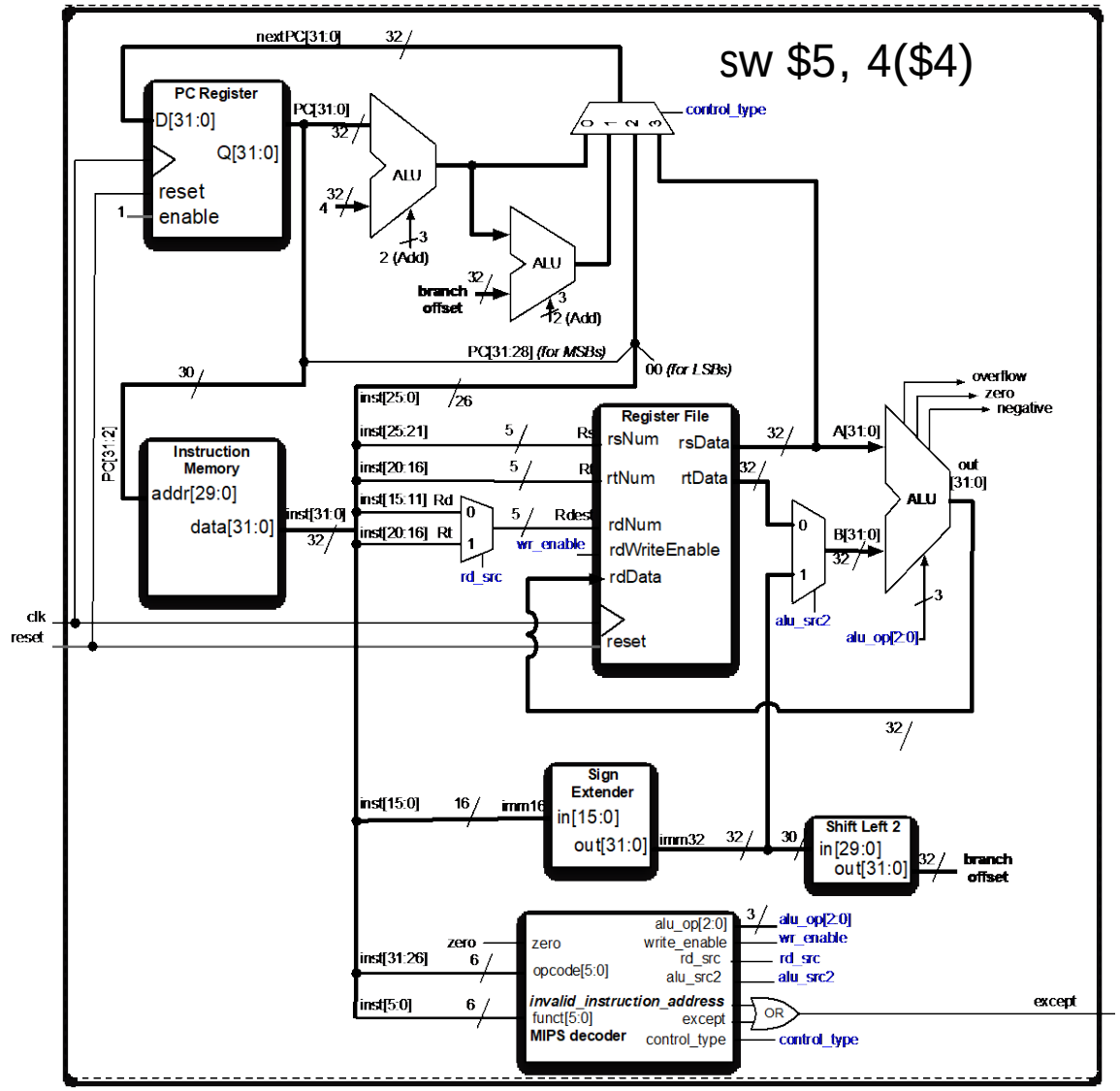


load byte unsigned **implemented**




nextPC[31:0]
32
PC Register
D[31:0]
Q[31:0]
reset
enable
1
PC[31:0]
32
ALU
4
32
3
2 (Add)
ALU
branch offset
32
3
12 (Add)
control_type
0 1 2 3
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
rd_src
Register File
rsNum
rsData
rtNum
rtData
rdNum
rdWriteEnable
rdData
reset
32
32
A[31:0]
B[31:0]
32
0
1
alu_src2
ALU
overflow
zero
negative
out [31:0]
3
alu_op[2:0]
31'b0
slt
0 1
Data Memory
addr[31:0]
data_out[31:0]
word_we
byte_we
data_in[31:0]
reset
word_we
byte_we
out[1:0]
32
data_out[31:24]
data_out[23:16]
data_out[15:8]
data_out[7:0]
3 2 1 0
clk
reset
lui
1 0
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
32
0
1
mem_read
32
32
0
1
byte_load
32
24'b0
8
inst[31:26]
6
inst[5:0]
6
zero
opcode[5:0]
funct[5:0]
MIPS decoder
alu_op[2:0]
write_enable
rd_src
alu_src2
except
control_type
lui
slt
byte_load
word_we
byte_we
mem_read
3
alu_op[2:0]
wr_enable
rd_src
alu_src2
except
control_type
lui
slt
byte_load
word_we
byte_we
mem_read

`store` implemented



Full Machine Datapath – Lab 6