



MIPS assembly programming:

Today's lecture

- **Exam Structure**
- **Review the Datapath**
 - Trace a couple of instructions
- **Assembly programming**
 - Register names
 - Arrays
 - Pseudo-instructions (including pseudo-branches)
 - If/else

Exam Structure

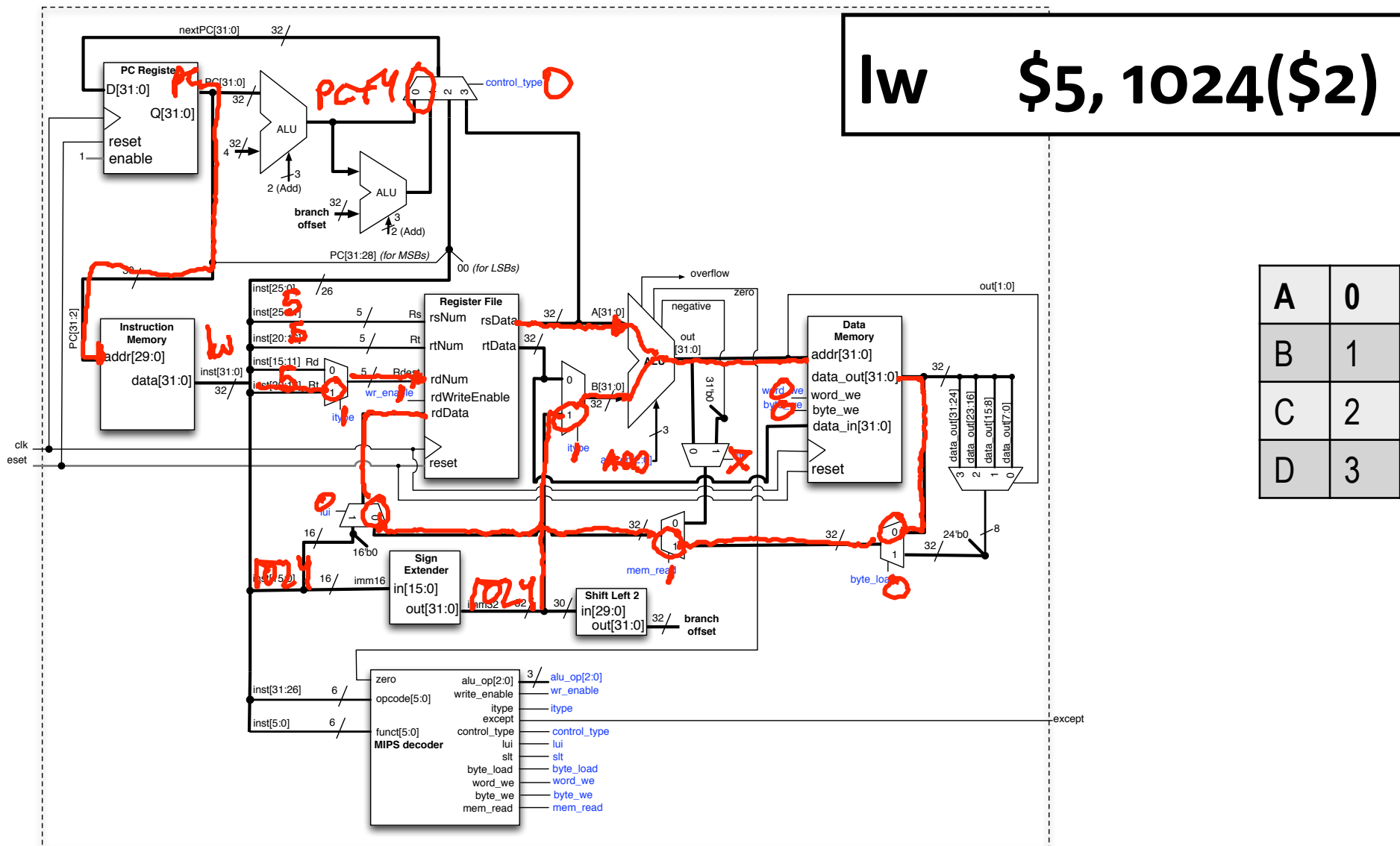
■ 3 main capabilities that you need to demonstrate:

- 76
1. Combinational Design (e.g., key pad) *gtrng*
 2. Sequential Design (e.g., score keeper) *word reader*
 3. Processor Datapath and Control (e.g., Lab 6)

(all of these will be available for the 2nd chance exam)

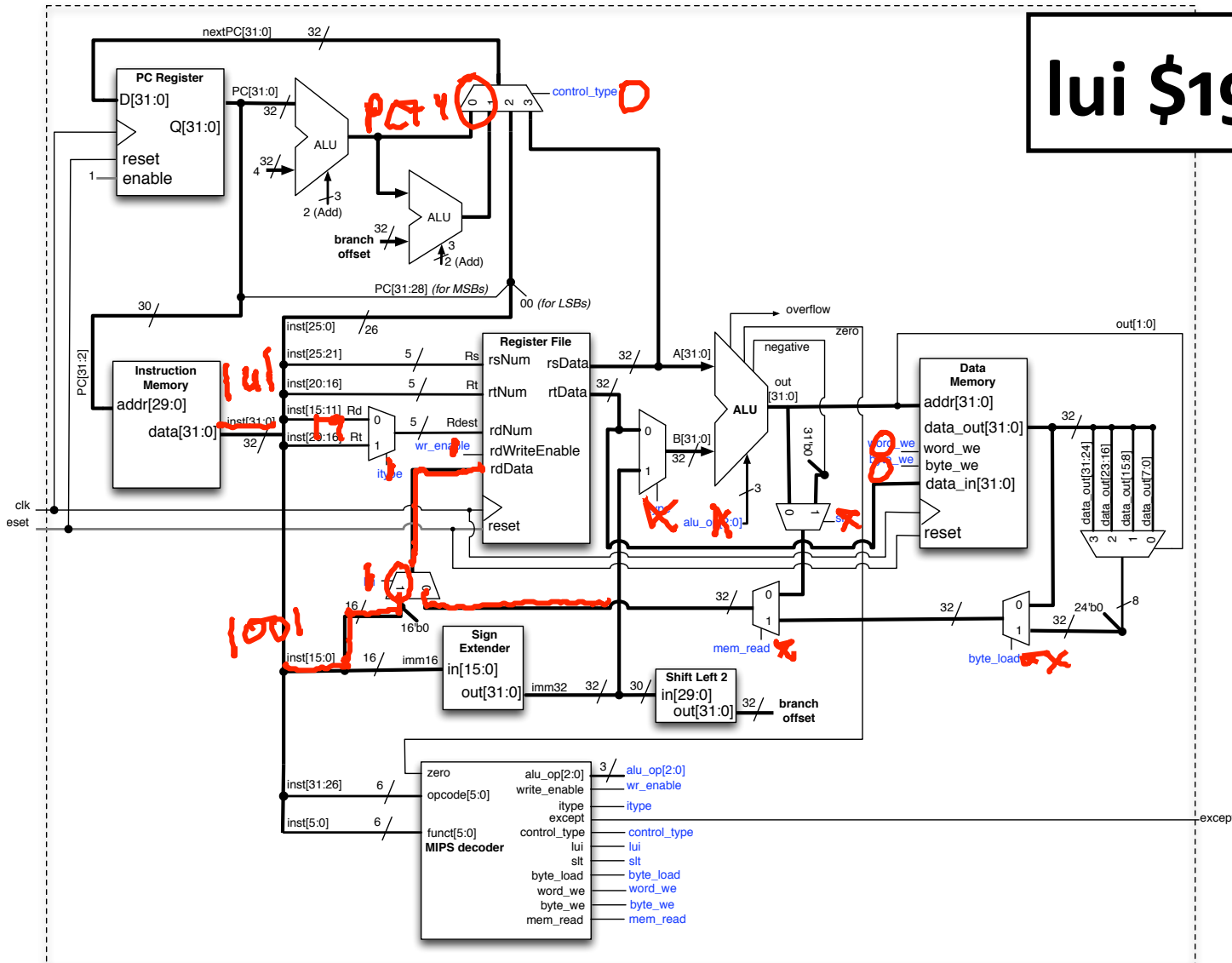
■ In addition there will be a number of short answer questions

- 45
- Covering topics selected from all of the lectures
 - (e.g., multiple choice, true/false, short answer)
 - The web problems are good examples of these questions.



A	0
B	1
C	2
D	3

CORE INSTRUCTION SET NAME, MNEMONIC	FOR- MAT	OPERATION (in Verilog)	OPCODE / FUNCT (Hex)
Load Word <code>lw</code>	I	$R[rt] = M[R[rs] + \text{SignExtImm}]$	(2) 23_{hex}



lui \$19, 0x1001

A	0
B	1
C	2
D	3

CORE INSTRUCTION SET
NAME, MNEMONIC

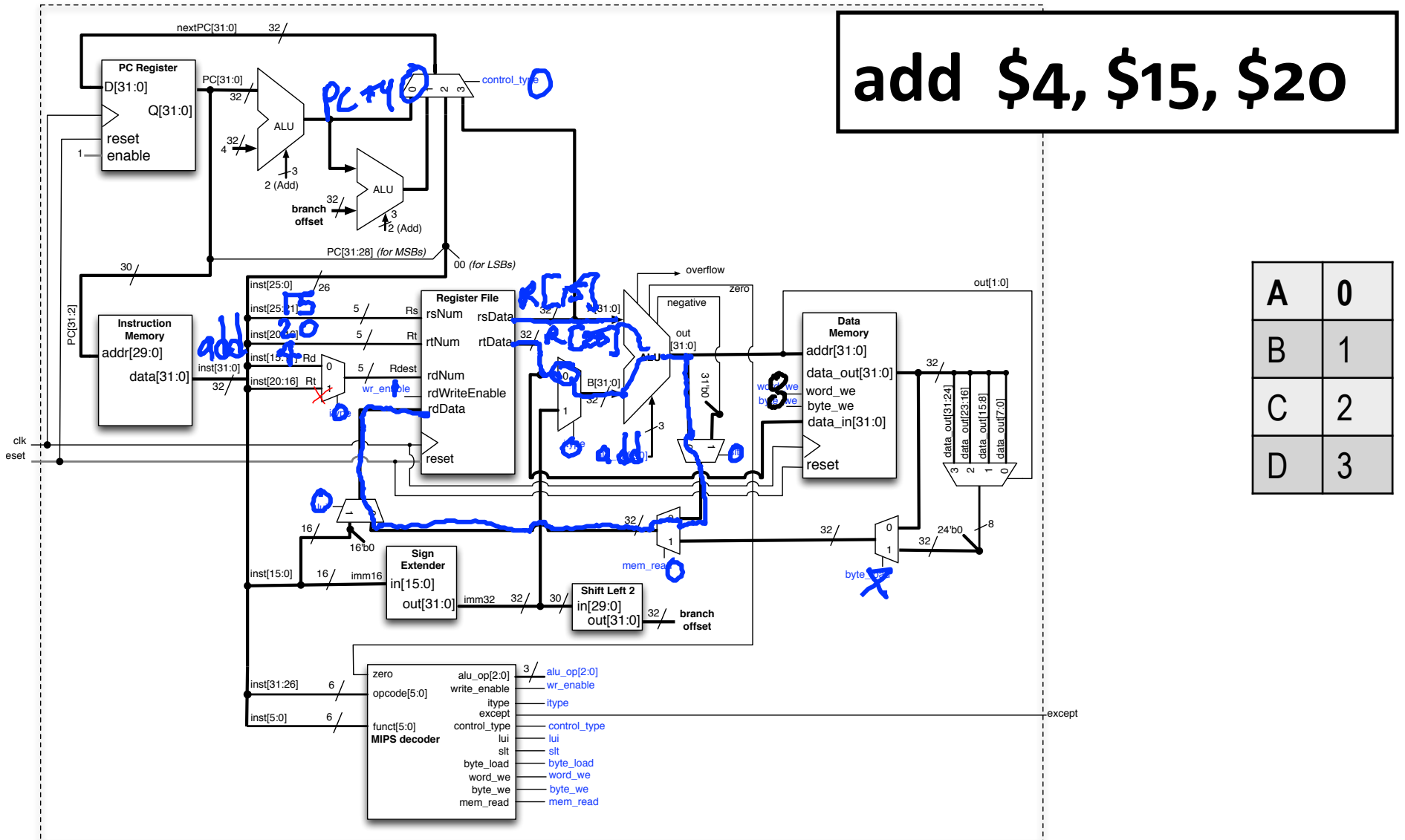
Load Upper Imm. lui

FOR-
MAT

I $R[rt] = \{imm, 16'b0\}$

OPERATION (in Verilog)
OPCODE / FUNCT
(Hex)

f_{hex}



add \$4, \$15, \$20

A	0
B	1
C	2
D	3

CORE INSTRUCTION SET
NAME, MNEMONIC

FOR-
MAT

OPERATION (in Verilog)
OPCODE / FUNCT
(Hex)

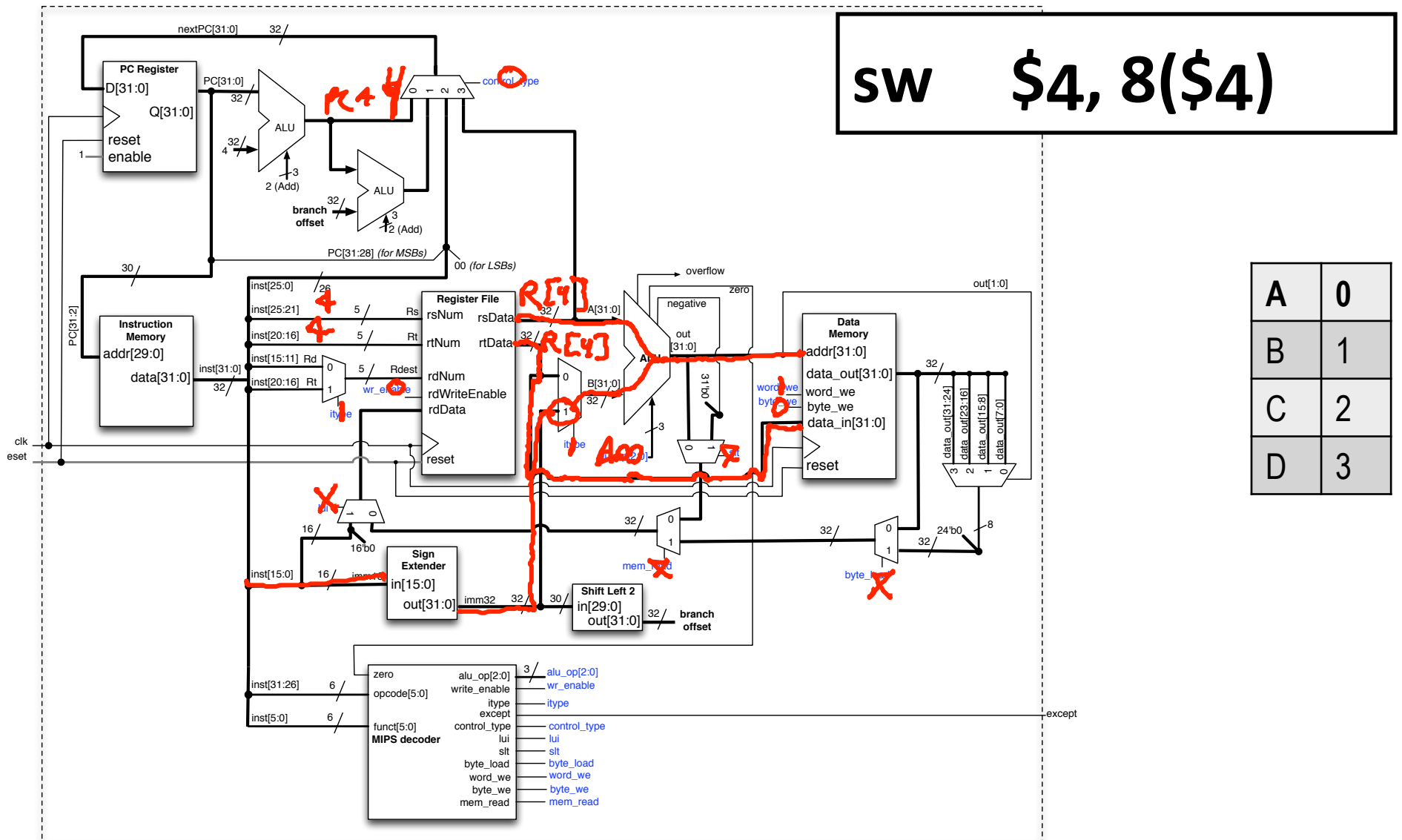
Add

add

R

$R[rd] = R[rs] + R[rt]$

(1) 0 / 20_{hex}



sw \$4, 8(\$4)

A	0
B	1
C	2
D	3

CORE INSTRUCTION SET	FOR-	OPERATION (in Verilog)	OPCODE / FUNCT
NAME, MNEMONIC	MAT		(Hex)
Store Word	sw	I $M[R[rs] + \text{SignExtImm}] = R[rt]$	(2) $2b_{\text{hex}}$

What you will need to learn for exam 2

- You must become “fluent” in MIPS assembly:
 - Translate from C to MIPS and MIPS to C
- Example problem from a 232 mid-term:

Question 3: Write a recursive function (30 points)

Here is a function `pow` that takes two arguments (`n` and `m`, both 32-bit numbers) and returns n^m (i.e., `n` raised to the m^{th} power).

```
int
pow(int n, int m) {
    if (m == 1)
        return n;
    return n * pow(n, m-1);
}
```

Translate this into a MIPS assembly language function.

MIPS register names

- In hardware, all the registers are equivalent:
 - Except register \$0, which is always zero
- In software, we'll use them for different purposes.
 - So we give them meaningful names
- For temporary values, we'll use the \$t registers

\$t0-\$t9

- If you have no reason for picking another register, then you should probably be using a \$t register.

Review: arithmetic expressions

- Complex arithmetic expressions may require multiple operations at the instruction set level.

$$t0 = (t1 + t2) \times (t3 - t4)$$

```
add $t0, $t1, $t2    # $t0 contains $t1 + $t2
sub $t5, $t3, $t4    # Temporary value $t5 = $t3 - $t4
mul $t0, $t0, $t5    # $t0 contains the final product
```

Computing with memory

- So, to compute with memory-based data, you must:
 1. Load the data from memory to the register file.
 2. Do the computation, leaving the result in a register.
 3. Store that value back to memory if needed.
- For example, let's say that you wanted to do the same addition, but the values were in memory. How can we do the following using MIPS assembly language?

```
char A[4] = {1, 2, 3, 4};
```

```
int result;
```

```
result = A[0] + A[1] + A[2] + A[3];
```

Allocating memory in the data segment

- Data define outside of functions is called global data
 - These variables can be accessed from any function
- This data is place in the program's global data segment
 - Allocated to memory addresses at compile time.
 - Amount of space allocated is based on variable type.

```
.data    // indicates the beginning of data segment
.word    // allocates space for 4-byte variable
.byte    // allocates space for 1-byte variable
.asciiz   // allocates space for an ASCII string
.space    // allocates a defined amount of space.
```

Review: Loading and storing bytes

- The MIPS “load byte” instruction **lb** transfers one byte of data from main memory to a register.

`lb $t0, 20($a0) # $t0 = Memory[$a0 + 20]`

- The “store byte” instruction **sb** transfers the lowest byte of data from a register into main memory.

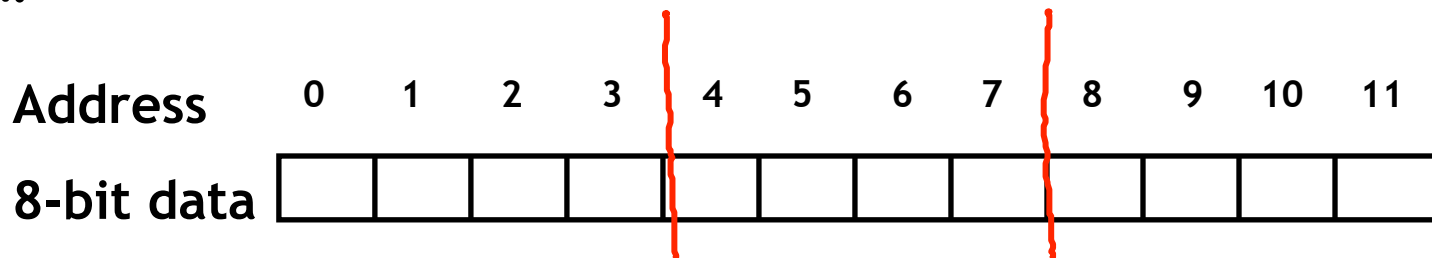
`sb $t0, 20($a0) # Memory[$a0 + 20] = $t0`

Review: Loading and storing words

- You can also load or store 32-bit quantities—a complete **word** instead of just a byte—with the **lw** and **sw** instructions.

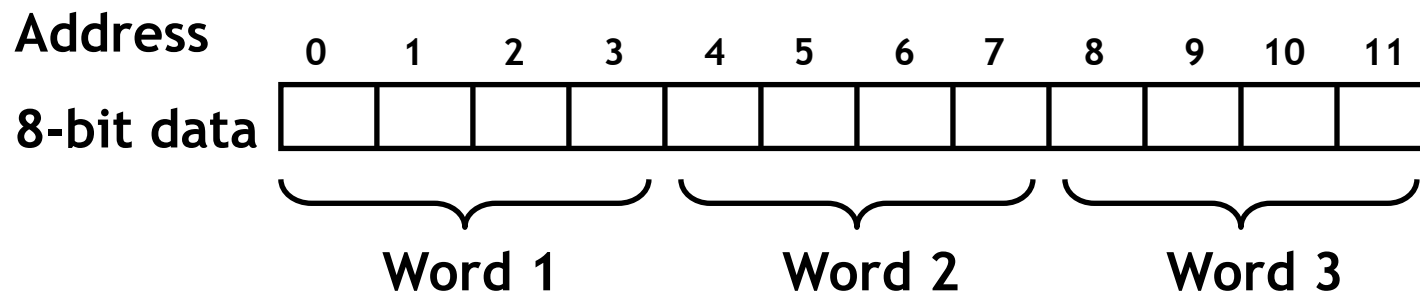
```
lw $t0, 20($a0)      # $t0 = Memory[$a0 + 20]
sw $t0, 20($a0)      # Memory[$a0 + 20] = $t0
```

- Most programming languages support several 32-bit data types.
 - Integers
 - Single-precision floating-point numbers
 - Memory addresses, or pointers
- Unless otherwise stated, we'll assume words are the basic unit of data.



Review: Memory alignment

- Since memory is byte-addressable, a 32-bit word actually occupies ~~four contiguous locations (bytes) of main memory.~~

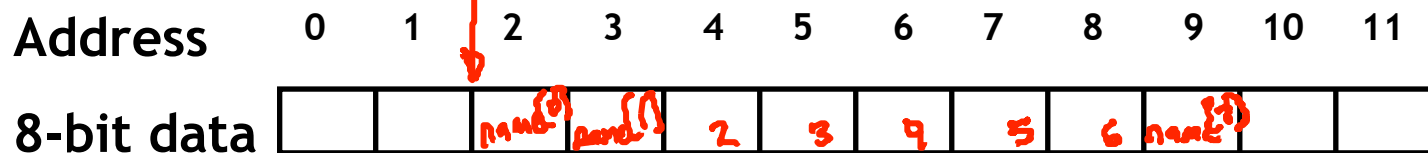


- The MIPS architecture requires words to be **aligned** in memory; 32-bit words must start at an address that is divisible by 4.
 - 0, 4, 8 and 12 are valid **word addresses**.
 - 1, 2, 3, 5, 6, 7, 9, 10 and 11 are *not* valid word addresses.
 - Unaligned memory accesses result in a **bus error**, which you may have unfortunately seen before.

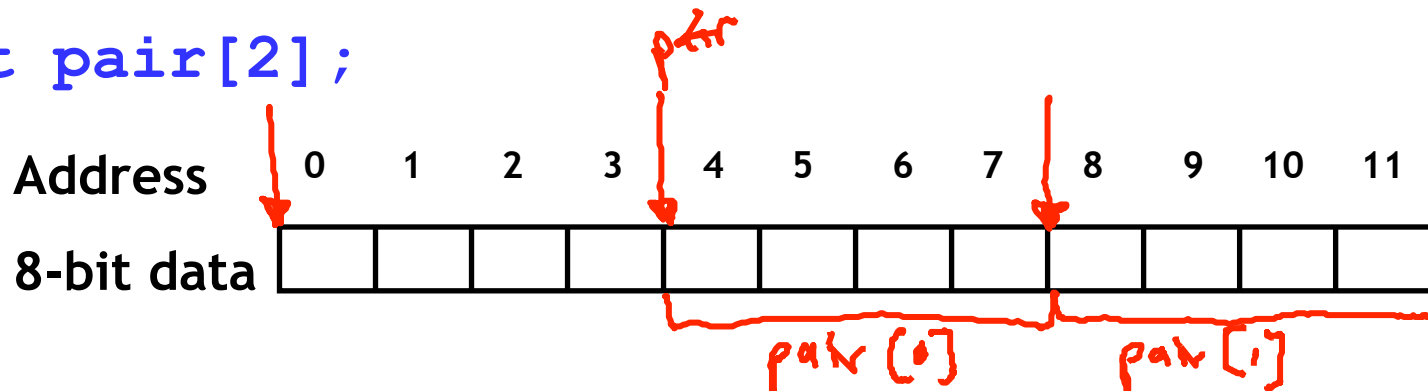
Arrays

- Arrays are just groups of variables contiguous in memory
 - Contiguous = laid out one after another in memory.

`char name[8];`



`int pair[2];`



An array of words

- Remember to be careful with memory addresses when accessing words.
- For instance, assume an array of words begins at address 2000.
 - The first array element is at address 2000.
 - The second word is at address 2004, not 2001.
- Revisiting the earlier example, if `$a0` contains 2000, then

```
lw $t0, 0($a0)
```

accesses the 0th word of the array, but

```
lw $t0, 8($a0)
```

would access the *2nd* word of the array, at address 2008.

Pseudo-instructions

- MIPS assemblers support **pseudo-instructions** that give the illusion of a more expressive instruction set, but are actually translated into one or more simpler, “real” instructions.
- In addition to the **la** (load address) we saw in section, you can use the **li** and **move** pseudo-instructions:

```
li    $a0, 2000      # Load immediate 2000 into $a0  
move  $a1, $t0      # Copy $t0 into $a1
```

- They are probably clearer than their corresponding MIPS instructions:

```
addi  $a0, $0, 2000   # Initialize $a0 to 2000  
add   $a1, $t0, $0    # Copy $t0 into $a1
```

- We'll see lots more pseudo-instructions this semester.
 - A complete list of instructions is given in [Appendix A](#) of the text.
 - Unless otherwise stated, you can always use pseudo-instructions in your assignments and on exams.

Pseudo-branches (motivation)

- Do you remember how much of a pain it to implement:

```
if (x < 10) {  
    ...  
}
```

bge \$t4, 10, skip

skip

- Need a slt and a beq... (or is it a bne?)

```
slti $t0, $t4, 10          # immediate version of slt  
beq $t0, $zero, skip_if_body # beq(a) or bne(b)?
```

- This is extremely error prone; bad design for humans

Pseudo-branches

- The MIPS processor only supports two branch instructions, **beq** and **bne**, but to simplify your life the assembler provides the following other branches:

```
blt    $t0, $t1, L1 // Branch if $t0 < $t1
ble    $t0, $t1, L2 // Branch if $t0 <= $t1
bgt    $t0, $t1, L3 // Branch if $t0 > $t1
bge    $t0, $t1, L4 // Branch if $t0 >= $t1
```

- There are also immediate versions of these branches, where the second source is a constant instead of a register.

Implementing pseudo-branches

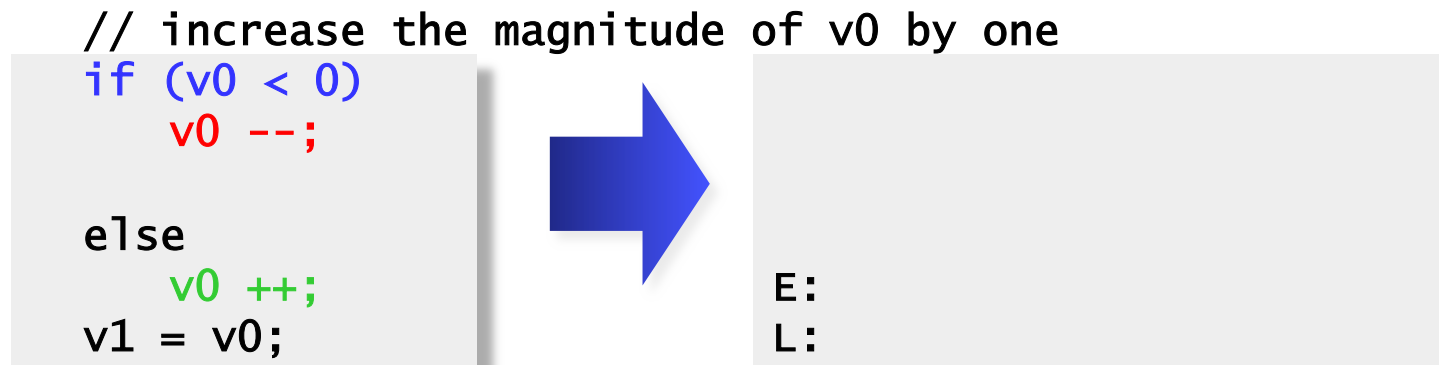
- Most pseudo-branches are implemented using `slt`. For example, a branch-if-less-than instruction `blt $a0, $a1, Label` is translated into the following.

```
slt  $at, $a0, $a1    // $at = 1 if $a0 < $a1
bne  $at, $0, Label    // Branch if $at != 0
```

- All of the pseudo-branches need a register to save the result of `slt`, even though it's not needed afterwards.
 - MIPS assemblers use register `$1`, or `$at`, for temporary storage.
 - You should probably avoid using `$at` in your own programs, as it may be overwritten by assembler-generated code.

Translating an if-then-else statements


- If there is an **else** clause, it is the target of the conditional branch
 - And the **then** clause needs a jump over the **else** clause



Dealing with else-if code is similar, but the target of the first branch will be another if statement.

Translating an if-then-else statements

- If there is an **else** clause, it is the target of the conditional branch
 - And the **then** clause needs a jump over the **else** clause

<pre>// increase the magnitude of v0 by one if (v0 < 0) v0 --; else v0 ++; v1 = v0;</pre>		<pre>bge \$v0, \$0, E sub \$v0, \$v0, 1 j L E: add \$v0, \$v0, 1 L: move \$v1, \$v0</pre>
---	---	---

- Dealing with else-if code is similar, but the target of the first branch will be another if statement.
 - Drawing the control-flow graph can help you out.

Case/Switch Statement

- Many high-level languages support **multi-way branches**, e.g.

```
switch (two_bits) {  
    case 0:    break;  
    case 1:    /* fall through */  
    case 2:    count ++;    break;  
    case 3:    count += 2;    break;  
}
```

- We could just translate the code to if, then, and
elses:

```
if ((two_bits == 1) || (two_bits == 2)) {  
    count ++;  
} else if (two_bits == 3) {  
    count += 2;  
}
```


Case/Switch Statement

```
switch (two_bits) {  
    case 0:    break;  
    case 1:    /* fall through */  
    case 2:    count ++;    break;  
    case 3:    count += 2;    break;  
}
```

- Alternatively, we can:

1. Create an array of jump targets
2. Load the entry indexed by the variable two_bits
3. Jump to that address using the jump register, or **jr**, instruction

- This is much easier to show than to tell.