

Digital Logic Design + Computer Architecture

Sayandeep Saha

**Assistant Professor
Department of Computer
Science and Engineering
Indian Institute of Technology
Bombay**



Instruction Set Architecture

How to talk to a Computer?

- Computers can be given “instructions”
- We have a set of instructions for every computer — called **instruction set**
- When you write a program, you write instructions..
 - More details later...
 - Every instruction some hardware circuit implemented inside the processor to get its job done.
- **Instruction Set Architecture:** specifies the set of instructions a processor understands, their encoding, how they access memory etc...

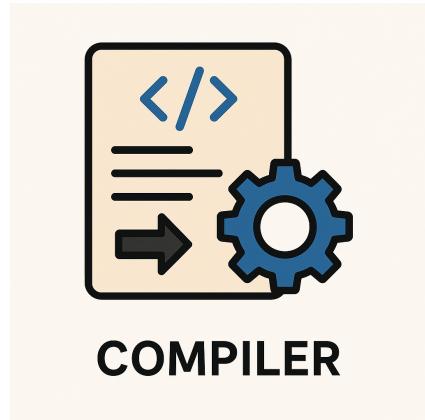


Image generated by ChatGPT

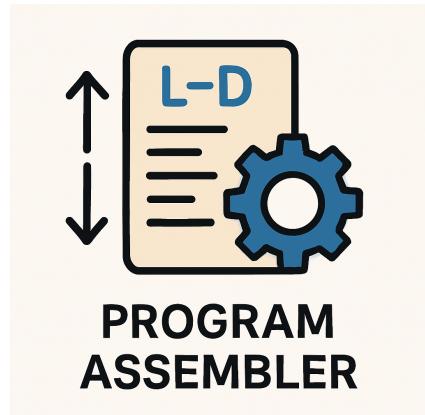
What happens when you write a program

- Say we write:

- $a = b + c;$



- There is a software program called **compiler**
 - Takes our code and encodes in terms of the instructions available for the computer
 - add reg1, reg2, reg3



- There is another program called **assembler** which converts the instruction (sequence) to bits
 - 0101110000110101



Image generated by ChatGPT

How to talk to a Computer?

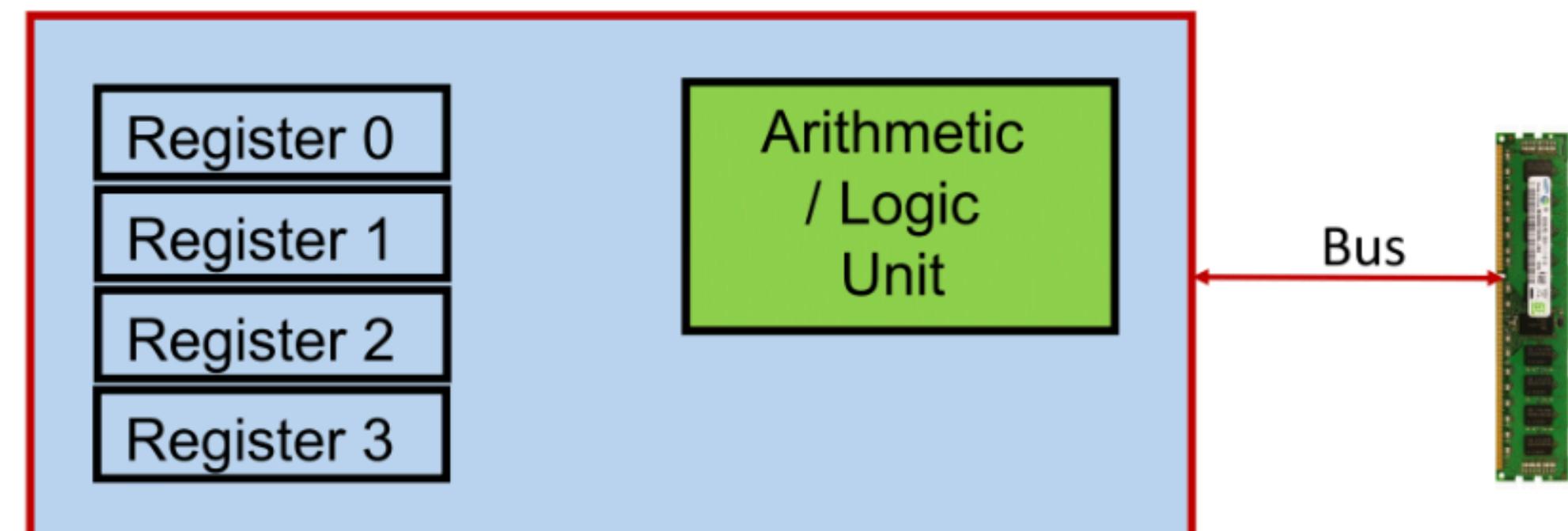
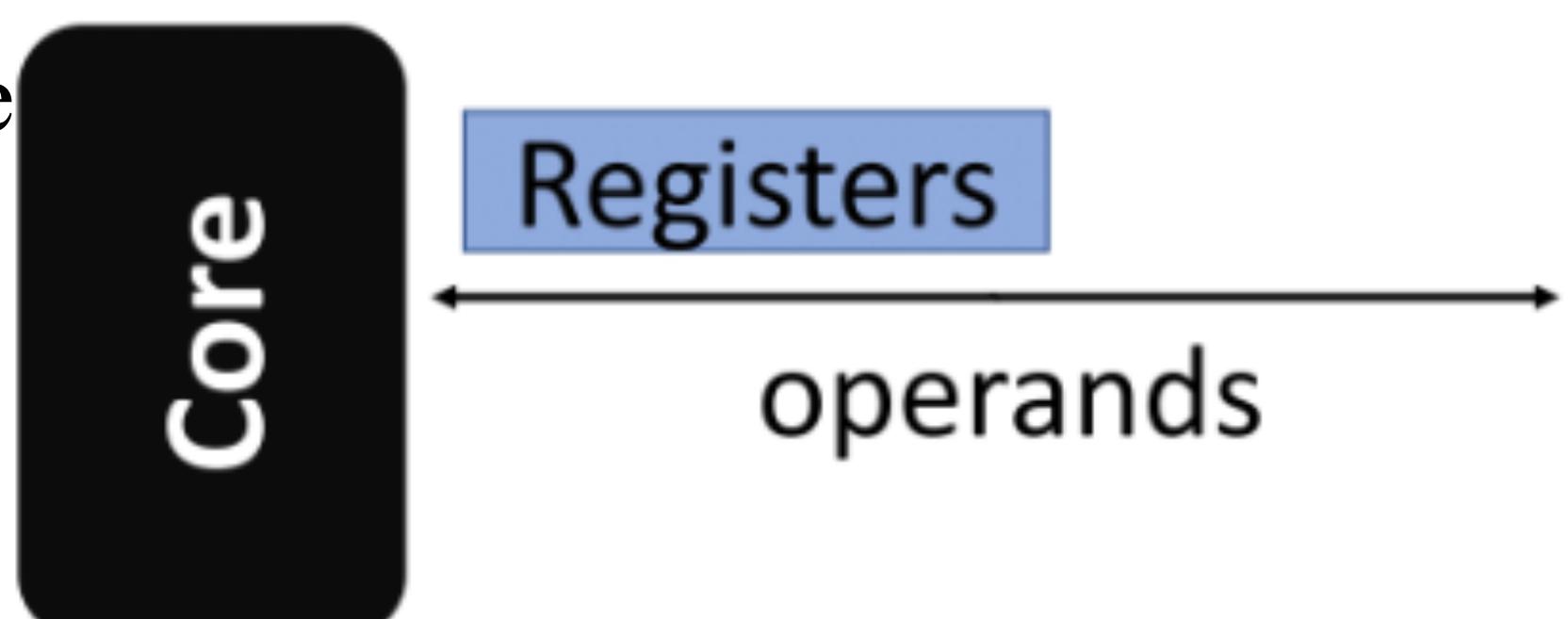
- **Instruction Set Architecture:** specifies the set of instructions a processor understands, their encoding, how they access memory etc...
 - **End of the day even your ChatGPT is a sequence of instructions** (many billions or trillions).
 - Instruction set is basically an **abstraction layer**
 - **Hides the complexity of hardware from the software designers,**
 - **Interfaces the software and hardware.**



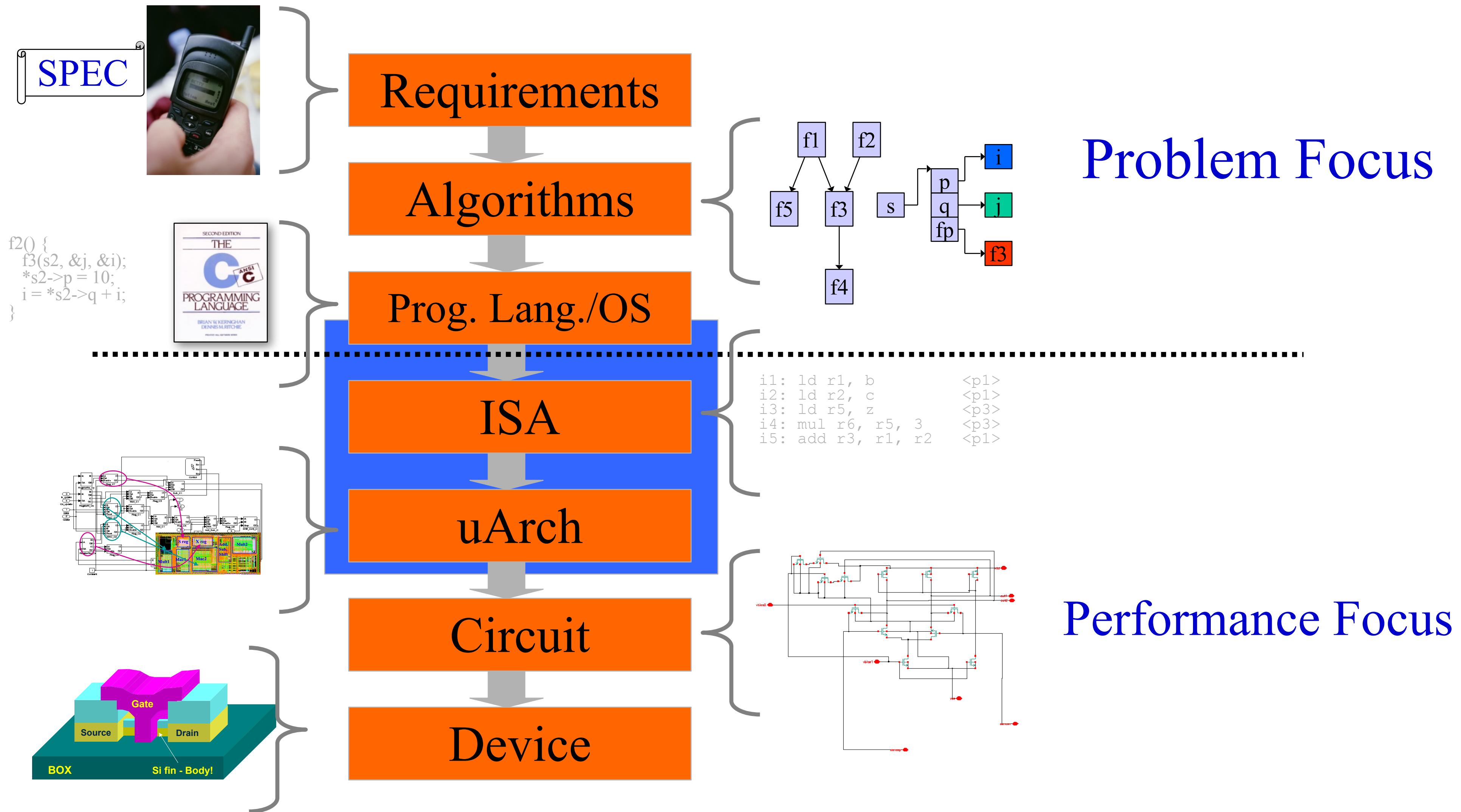
Image generated by ChatGPT

Let's get into the processor a bit

- It is a sequential circuit with a **limited** number of registers.
 - It interacts with an external “memory”.
 - Every instruction operates on some **operands** and generate results.
- Results and operands are stored in **registers**.
 - **But they can also be in memory as the number of registers are limited**
- Note that typically such memory (called **DRAM** or **Dynamic Random Access Memory**) is off chip —**outside the processor**
 - **To operate, you have to bring the data from memory and store the results back**

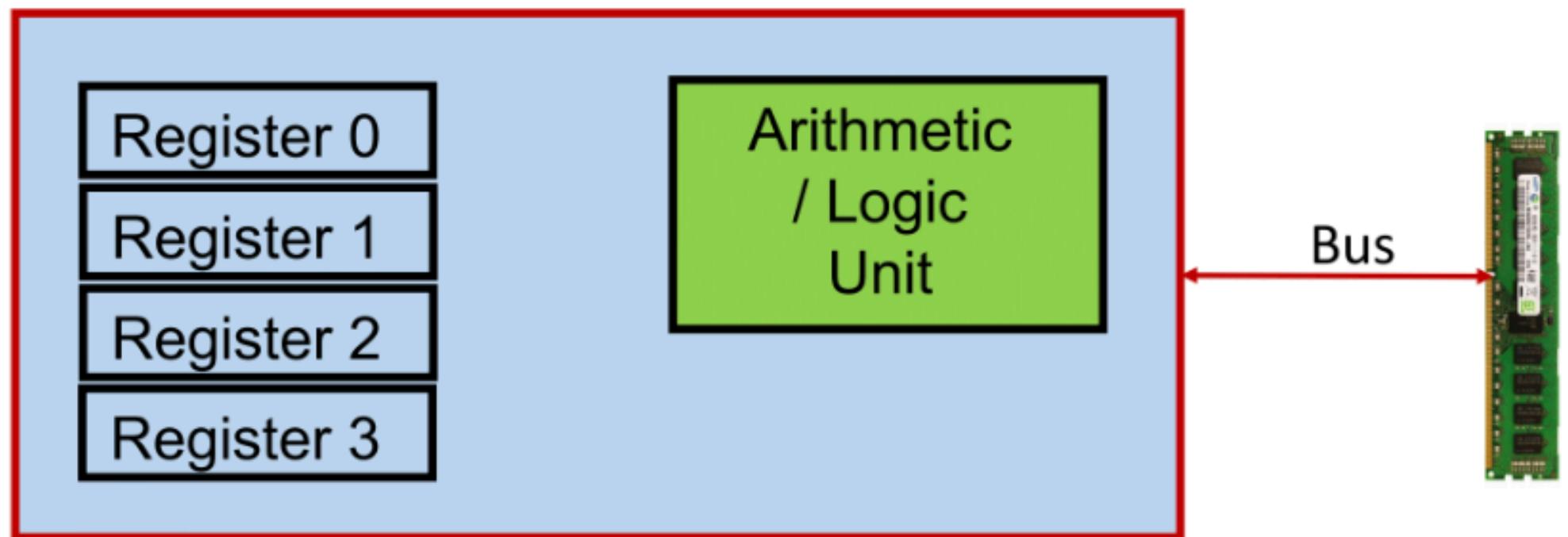
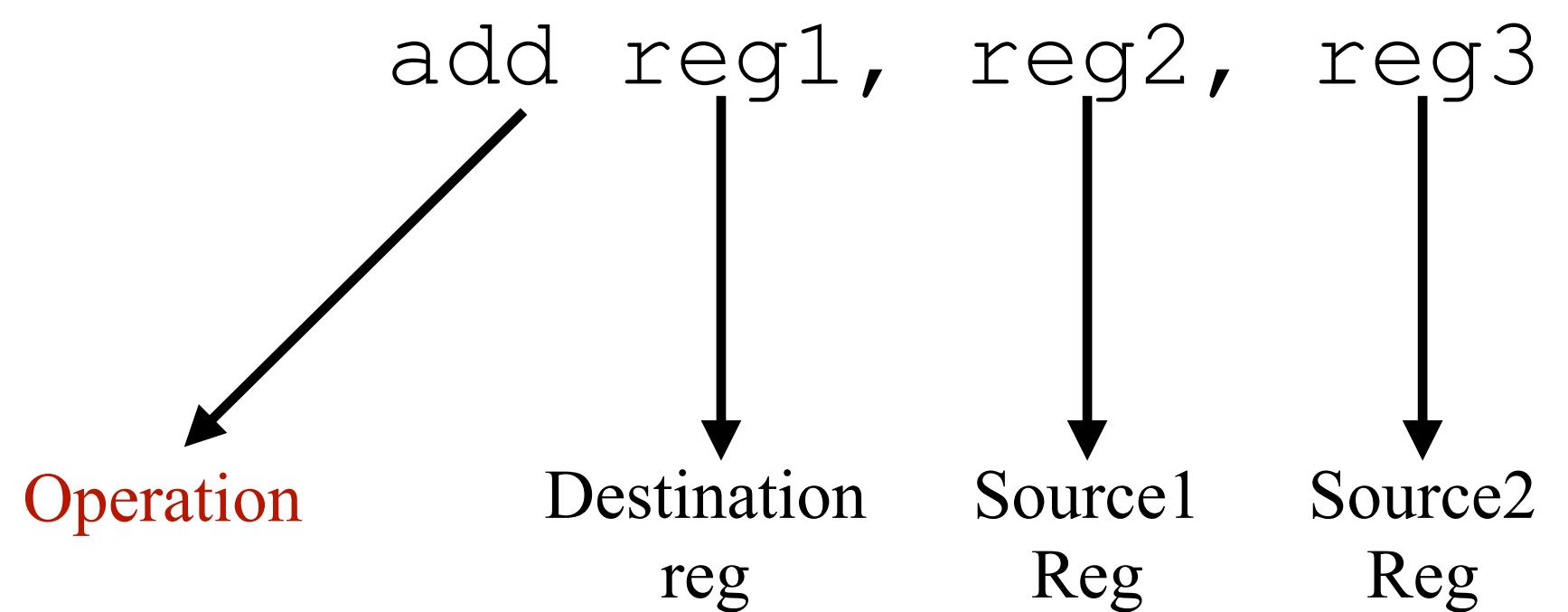


The Big Picture



Dissection of an Instruction

- Let's focus on the simplistic view of the processor



- Most of the arithmetic/logical instructions can take this form — not all though

Instruction Set Architectures (ISA)

- There are many...
 - Intel uses **X86**
 - Apple uses a version of **AArch64** (ARM)
 - The entire world of embedded processors like ST-Microelectronics uses ARM
 - Now **RISC-V** is becoming a mainstream trend.
 - We shall study MIPS — a simple to understand ISA

Instruction Set Architectures (ISA)

- We shall study MIPS — a simple to understand ISA
 - Great for beginning...
 - Similar to ARM
 - Still in use in the embedded devices
 - Your smart card
 - Modems
 - Bitcoin-wallets

Now let's write some MIPS

- We shall name the registers as \$0, \$1, or \$a0, \$g1 etc...
- Now we shall try something a bit more complex...

add reg1, reg2, reg3
↓
add \$0, \$1, \$2

Now let's write some MIPS

- Let's compute: $a = b + c - d$
- No idea? — get idea :P

add reg1, reg2, reg3
↓
add \$0, \$1, \$2

Now let's write some MIPS

- Let's compute: $a = b + c - d$
add \$0, \$1, \$2
- Assume we have add and sub instructions taking two
sources and one destination register
sub \$0, \$1, \$2

Now let's write some MIPS

- Let's compute: $a = b+c-d$
add \$0, \$1, \$2
- Assume we have add and sub instructions taking two sources and one destination register
sub \$0, \$1, \$2
- First' let's simplify :
 - $t = b+c$
 - $a = t-d$
- Now, I can map to instructions..
 - add \$r0, \$r1, \$r2 // $t = b+c$
 - sub \$d0, \$r0, \$r3 // $a = t-d$

• **Observe:** I use a temporary register...

Now let's write some MIPS

- Let's try: $f = (g+h) - (i+j)$

Now let's write some MIPS

- Let's try: $f = (g+h) - (i+j)$

- add \$r0, \$r1, \$r2 // $x = g+h$
- add \$r3, \$r4, \$r5 // $y = i+j$
- Sub \$r0, \$r0, \$r3 // $f = x-y$

- **Food of thought:** Well, do I really need to reuse registers???



Ok...A Few MIPS Details...

- We have 32 registers in the processor
 - So we have to reuse registers, no other option...
 - Typically, registers are 32-bits...
- But why don't we have infinite number of registers
 - Well, every piece of register is a real hardware...



- But: Why 32??

Ok...A Few MIPS Details...

- We have 32 registers in the processor
 - So we have to reuse registers, no other option...
 - Typically, registers are 32-bits...
 - Each instruction also encoded in 32 bits



- But: Why 32??

- But why don't we have infinite number of registers
 - Well, every piece of register is a real hardware...

The choice depends on several factors, like the speed of the execution, the usage and size of memory, the size of code, the encoding and decoding of instructions....**It's not a random choice...**

Immediate Instructions...

- $b = a + 7$
`addi $r0, $r1, 7`
 - We don't need a register for the constant...
 - Can you tell me why?? Just guess...



Immediate Instructions...

- $b = a + 7$

`addi $r0, $r1, 7`

- We don't need a register for the constant...
- Can you tell me why?? Just guess...



- i stands for ‘immediate’
- The constant is in **2's complement form and of 16 bits.**
- Question: Do I need a subi instruction??

Zero Is Very Special in Our Life...

- MIPS has a register which is called \$zero
 - It stores 0
 - What is the purpose?
 - Well, a lot...you will see
 - A simple use of \$zero

```
add $r1, $r0, $zero // a = b
```

- But again, why???



Zero Is Very Special in Our Life...

- MIPS has a register which is called \$zero
 - It stores 0
 - What is the purpose?
 - Well, a lot...you will see
 - A simple use of \$zero

```
add $r1, $r0, $zero // a = b
```

- But again, why??? — just not needed



a=b...The Pseudo-Instructions

- You can still write...

```
move $r1, $r0 // a = b
```

- But it is a pseudo-instruction
- Internally it converts to add
- Once again an engineering choice
- There are many such pseudo-instructions. See:

https://en.wikibooks.org/wiki/MIPS_Assembly/Pseudoinstructions

Logical Instructions

- Your good old Boolean algebra

sll, srl, and, or, nor, andi, ori etc

No **not** instruction ☺, well not is nor with one operand=0

- Remember: These are **bitwise operations...**
 - Treats the operands as bit strings instead of numbers

Logical Instructions

- Your good old Boolean algebra

sll, srl, and, or, nor, andi, ori etc

No **not** instruction ☺, well not is nor with one operand=0

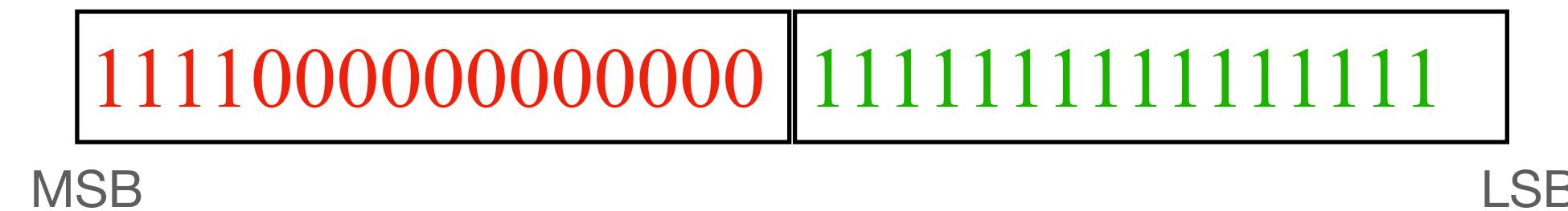
- Remember: These are **bitwise operations...**
 - Treats the operands as bit strings instead of numbers

Critical Thinking...

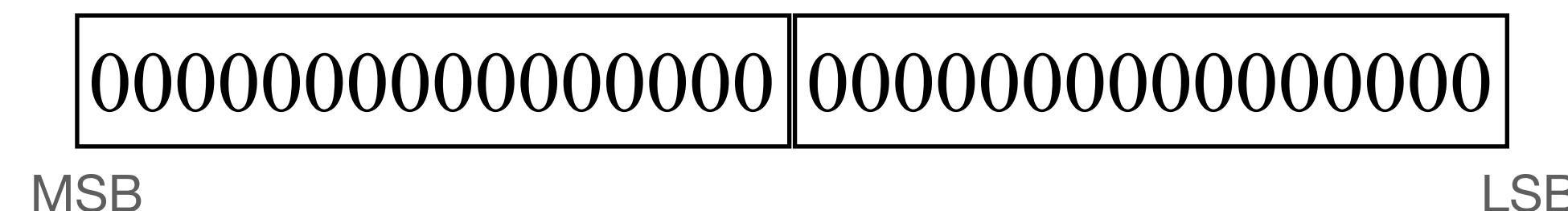
- We have seen that constants are 16 bits...
- But registers are 32-bits...
- How to store a 32-bit constant in a register???
 - Let's say the constant is:
 - 11110000000000001111111111111111
 - In Hex: 0xF000FFFF
- Info: You have the following new instruction:
 - **lui \$r0, const** // loads const in the upper 16 bits of the register \$r0

Critical Thinking...

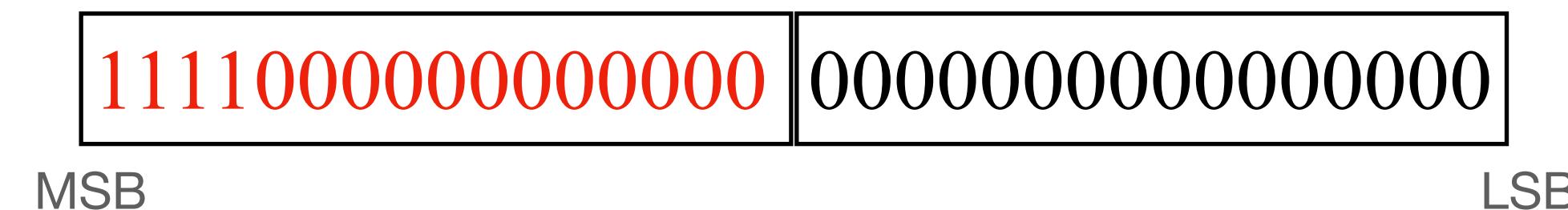
- Think, how the data will be represented inside your register...



- Initially The register \$r0 is at (simplifying assumption...does not matter)

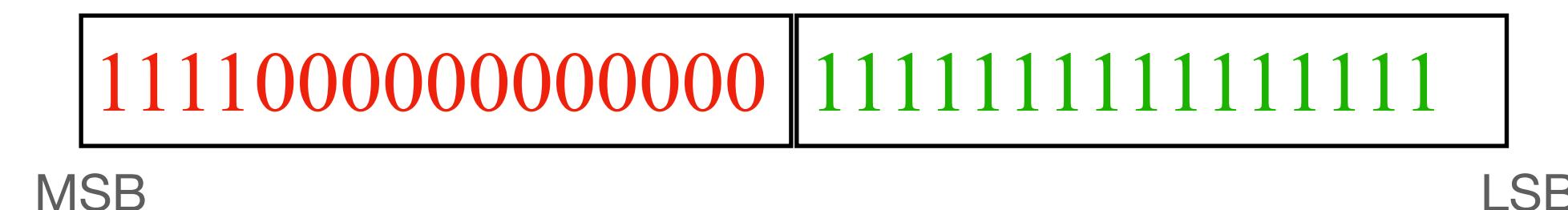


- Now do: lui \$r0, 0xF000



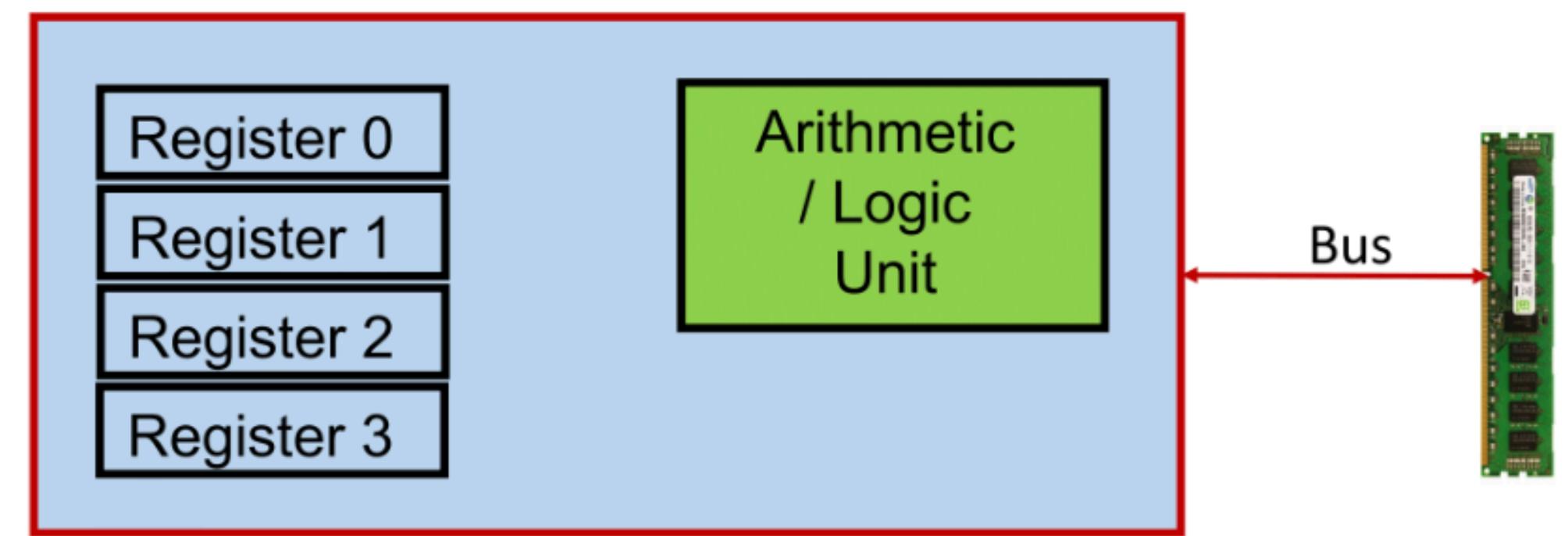
- Now do, addi \$r0, 0xFFFF

• You can also do ori



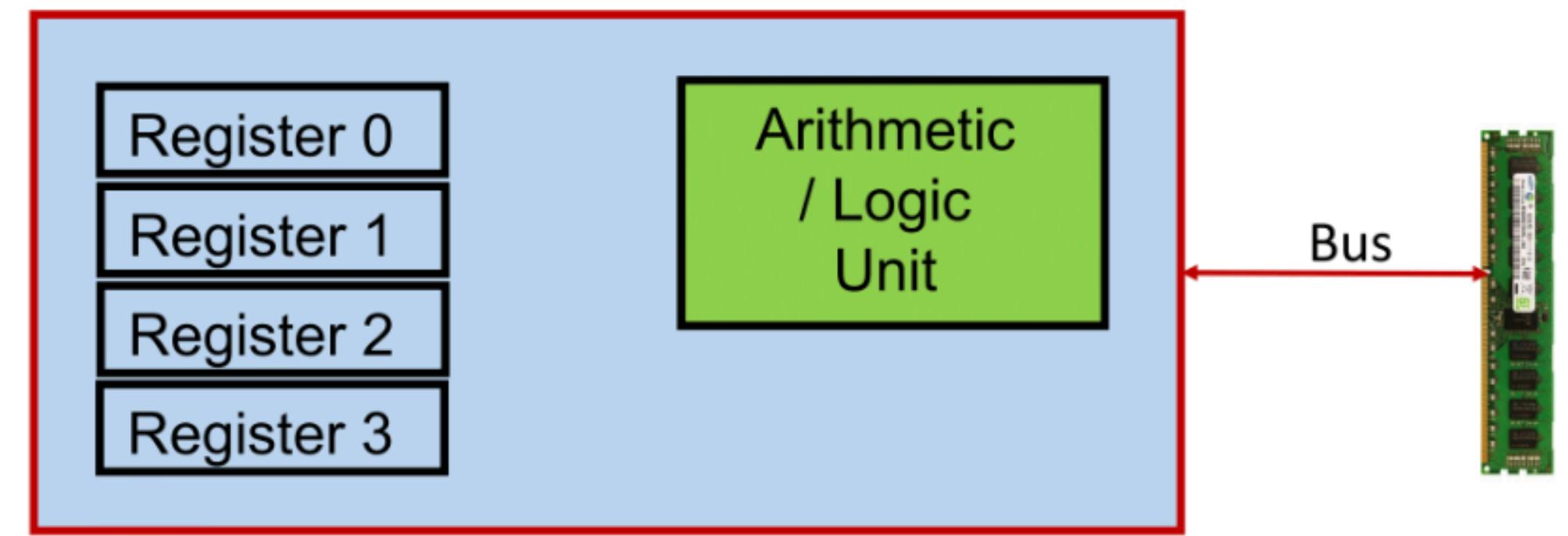
How to Use Your Memory??

- Recall, that MIPS only have 32 registers.
- Have you ever cared about counts while declaring variables in your program? — No way...
- Then how things work?
 - How can every program fits itself in 32 registers?



How to Use Your Memory??

- Solution:
 - Just store things in an external memory
 - Fetch the data to registers whenever it is required
 - Store the results after processing.
 - But still something is missing here...What is that??



How to Use Your Memory??

- Name this person?



How to Use Your Memory??

- Name this person?
 - John Luis von Neumann



How to Use Your Memory??

- In the old days, “programming” involved actually changing a machine’s physical configuration:
 - by flipping switches or connecting wires.
 - Memory only stored data that was being operated on.
- Then around 1944, **John von Neumann and others got the idea to encode instructions in a format that could be stored in memory just like data. — Stored program paradigm**
 - The processor interprets and executes instructions from memory



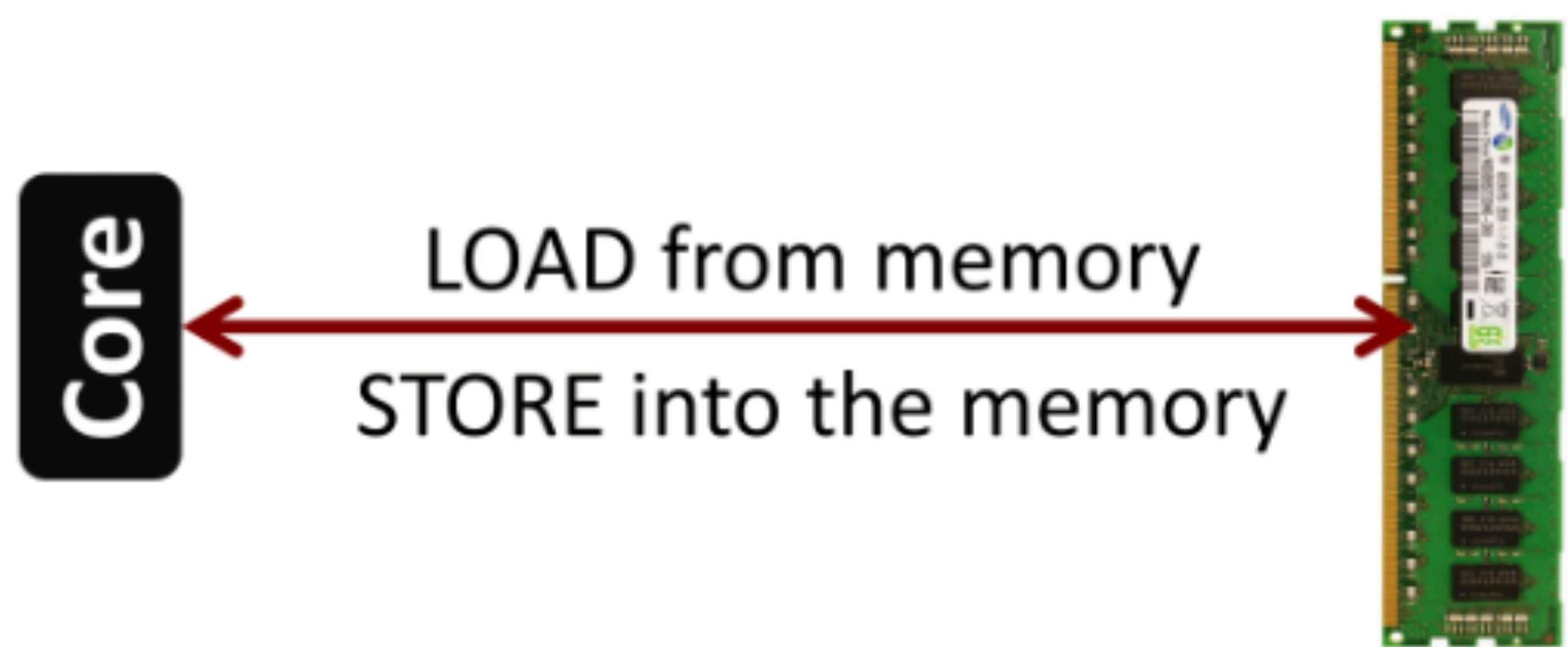
How to Use Your Memory??

- In the old days, “programming” involved actually changing a machine’s physical configuration:
 - by flipping switches or connecting wires.
 - Memory only stored data that was being operated on.
- Then around 1944, **John von Neumann and others got the idea to encode instructions in a format that could be stored in memory just like data. — Stored program paradigm**
 - The processor interprets and executes instructions from memory



Memory Instructions

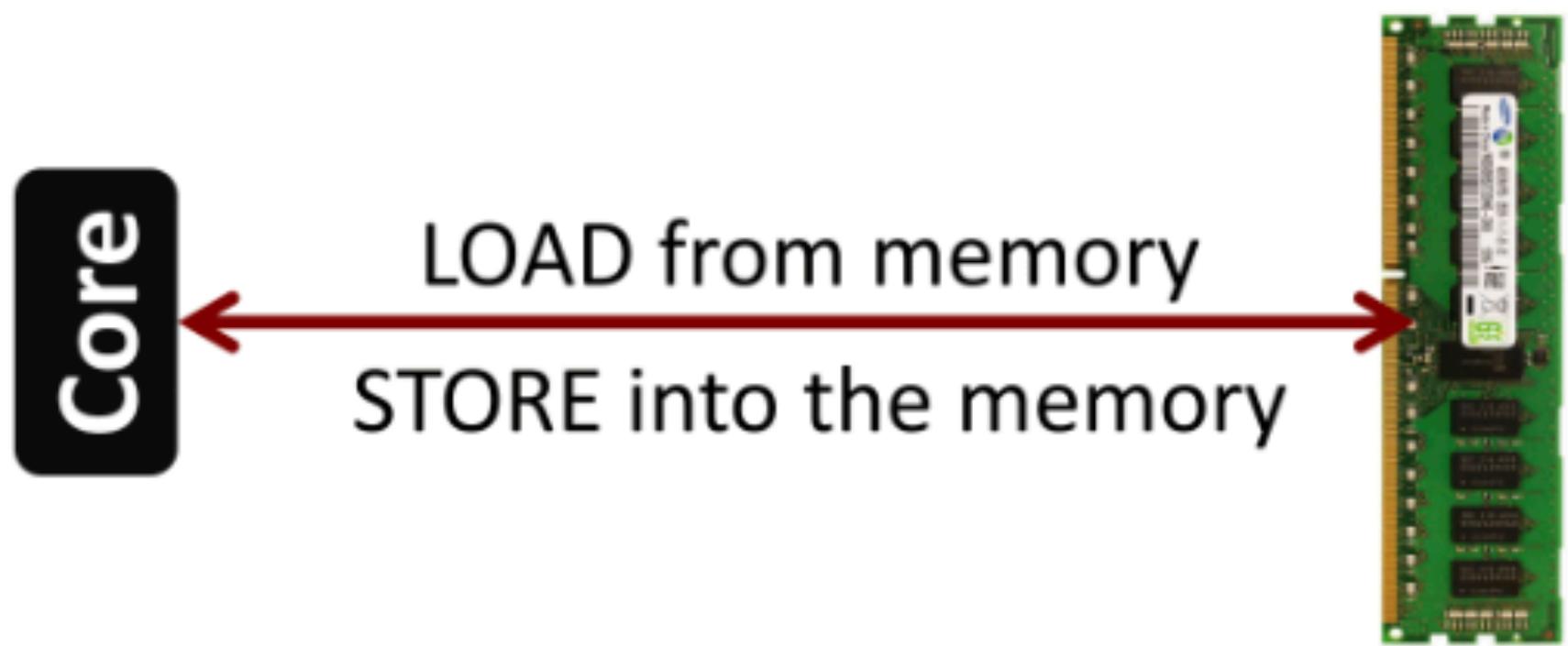
- Load-Store Architecture:
 - Load your data to process
 - Store it back...
- Instructions are handled in a slightly different manner....will come to that...



lw \$t0, 1(\$a0) # \$t0 = Memory[\$a0 + 1]
sw \$t0, 1(\$a0) # Memory[\$a0 + 1] = \$t0

Memory Instructions

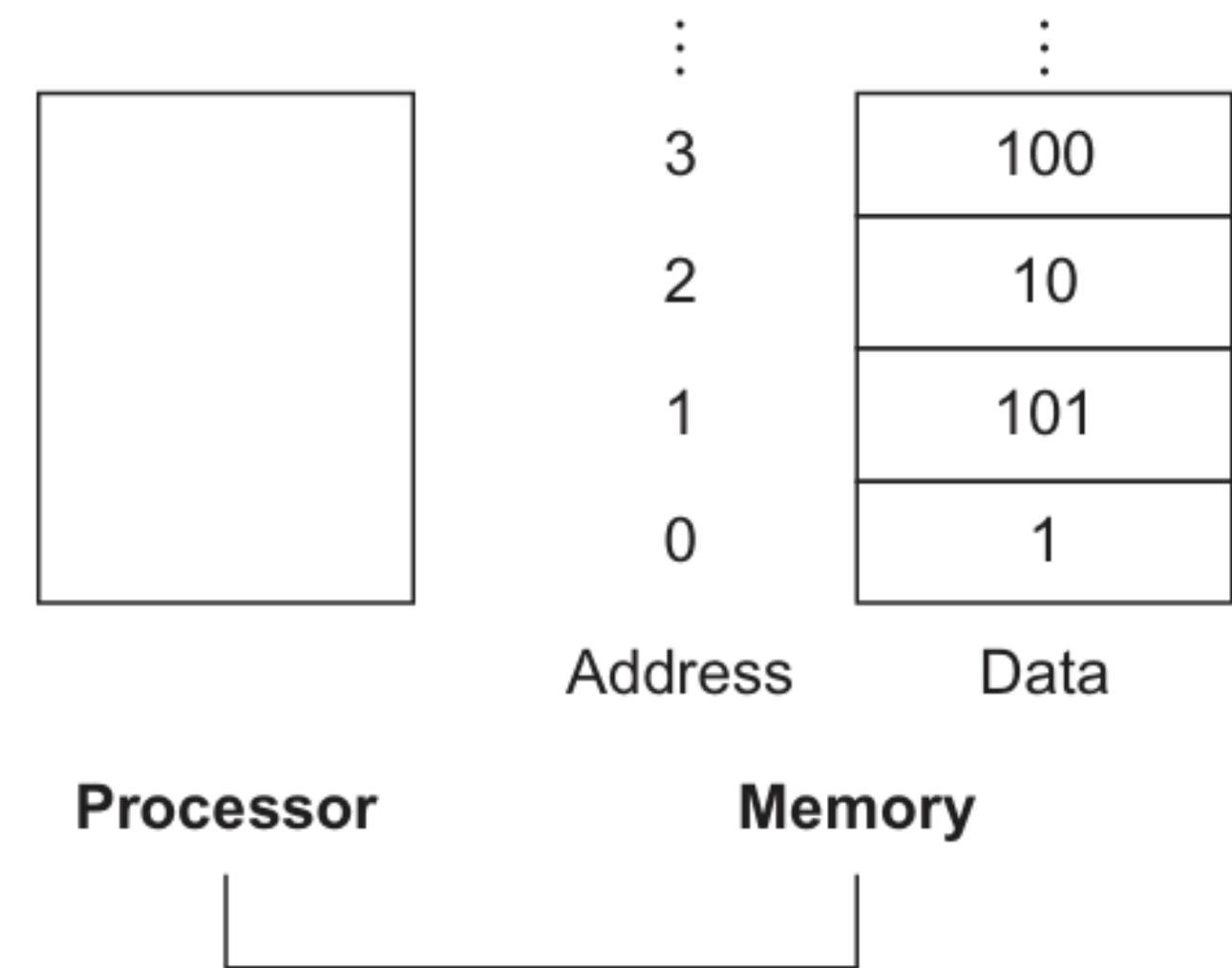
- Load-Store Architecture:
 - Load your data to process
 - Store it back...
- Instructions are handled in a slightly different manner....will come to that...
- But, a critical question:
 - How do you know where to find the data inside memory?



lw \$t0, 1(\$a0) # \$t0 = Memory[\$a0 + 1]
sw \$t0, 1(\$a0) # Memory[\$a0 + 1] = \$t0

Memory Instructions

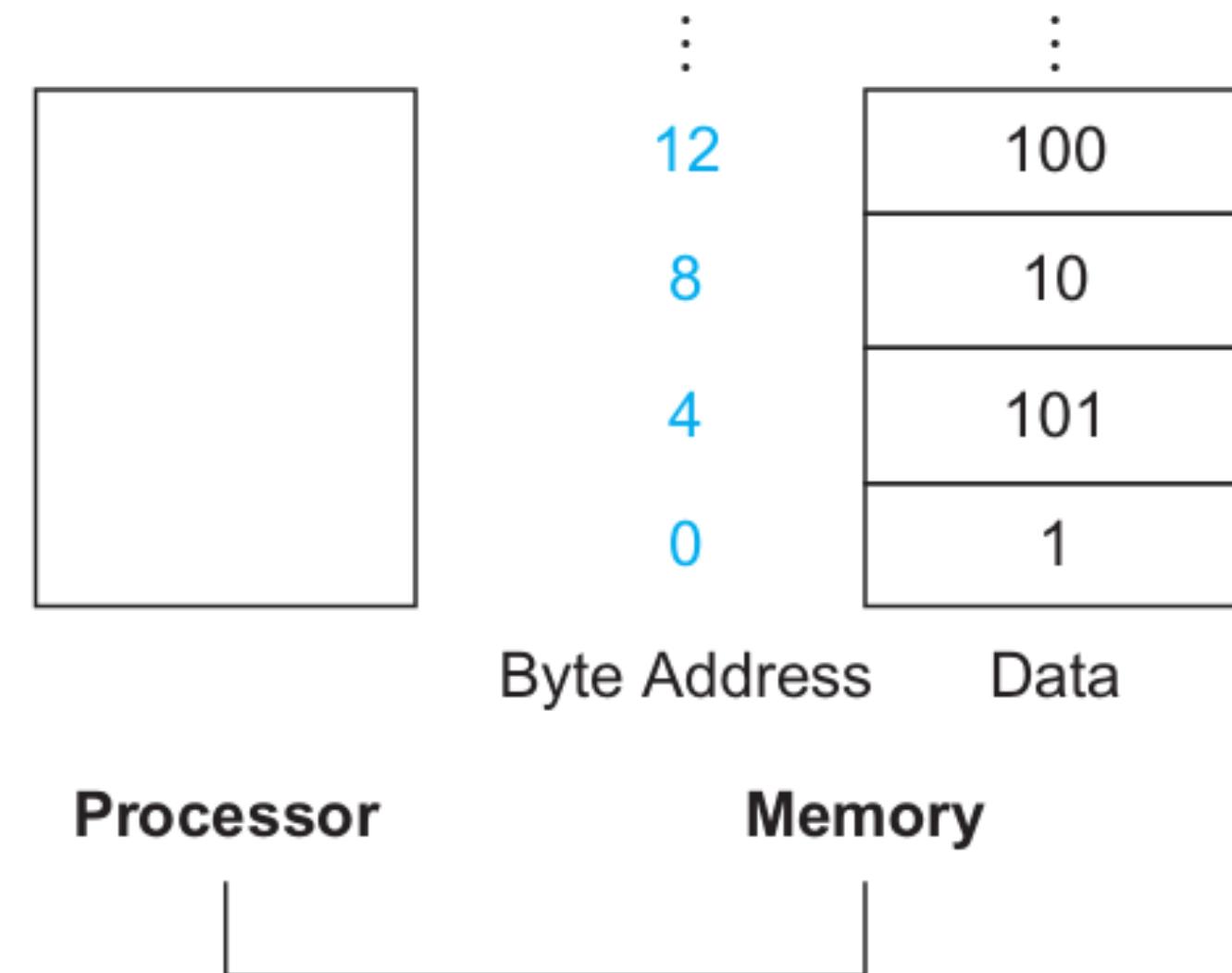
- But, a critical question:
 - How do you know where to find the data inside memory?
 - Memory has addresses
 - Think it like a large contiguous array...
 - **Every byte in memory has an unique address**
 - **Byte-addressable**
 - **BTW, each address is 32-bit in MIPS**



Memory Instructions

```
lw $t0, 1($a0)    # $t0 = Memory[$a0 + 1]  
sw $t0, 1($a0)    # Memory[$a0 + 1] = $t0
```

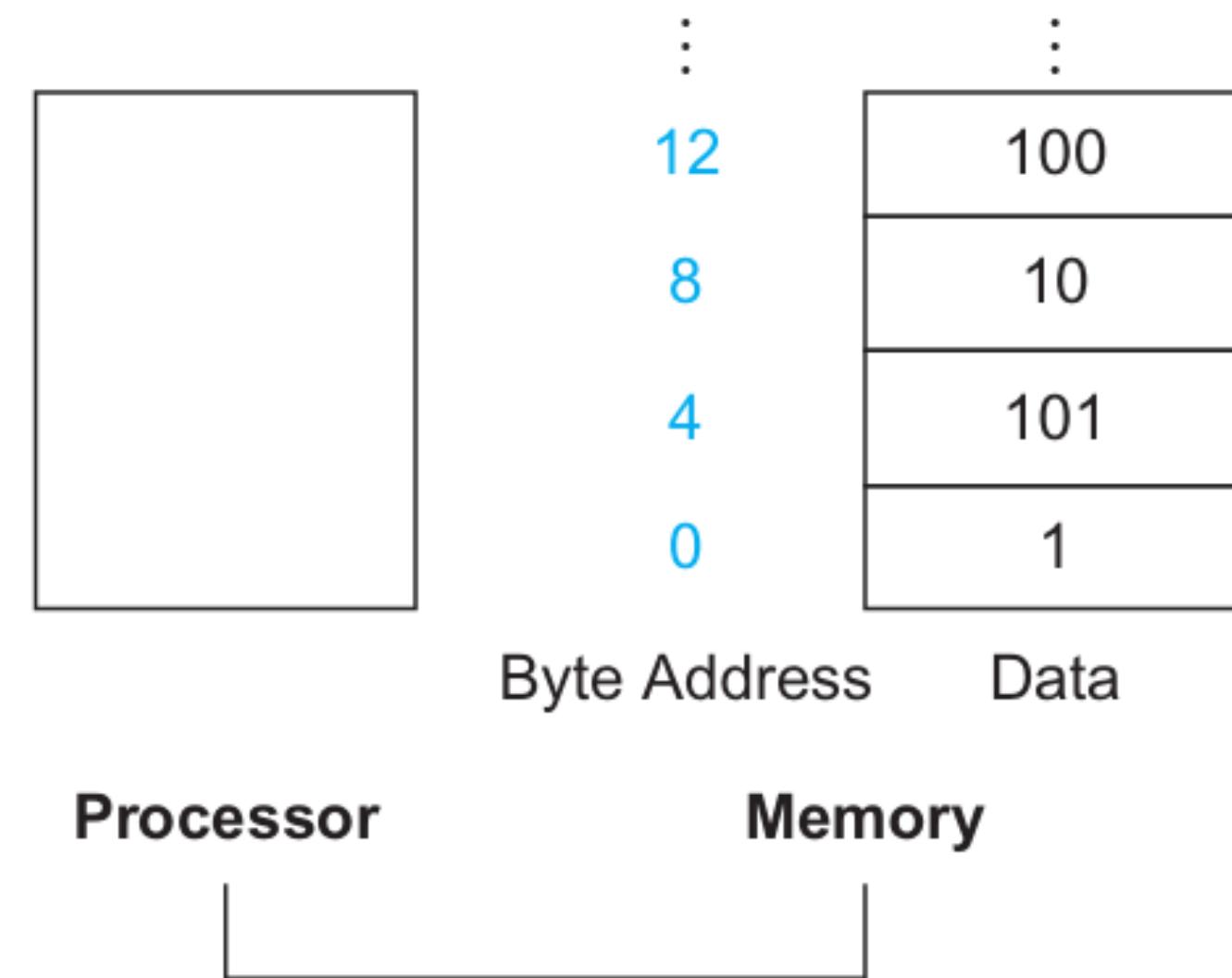
- The **lw** is interpreted as “load word”
 - MIPS also have other variants like “load byte” (**lb**)
- Data comes in **\$t0**.
- But what is the **1(\$a0)** part signify?
 - **\$a0** is the *base address* of the location you want to read from memory
 - **1** is called the *offset*.
- But why don’t you read directly?



Memory Instructions

```
lw $t0, 1($a0)    # $t0 = Memory[$a0 + 1]  
sw $t0, 1($a0)    # Memory[$a0 + 1] = $t0
```

- The **lw** is interpreted as “load word”
 - MIPS also have other variants like “load byte” (**lb**)
- Data comes in **\$t0**.
- But what is the **1(\$a0)** part signify?
 - **\$a0** is the *base address* of the location you want to read from memory
 - **1** is called the *offset*.
- But why don’t you read directly?
 - Again a design choice, to ease compilation, programming, and hardware design...

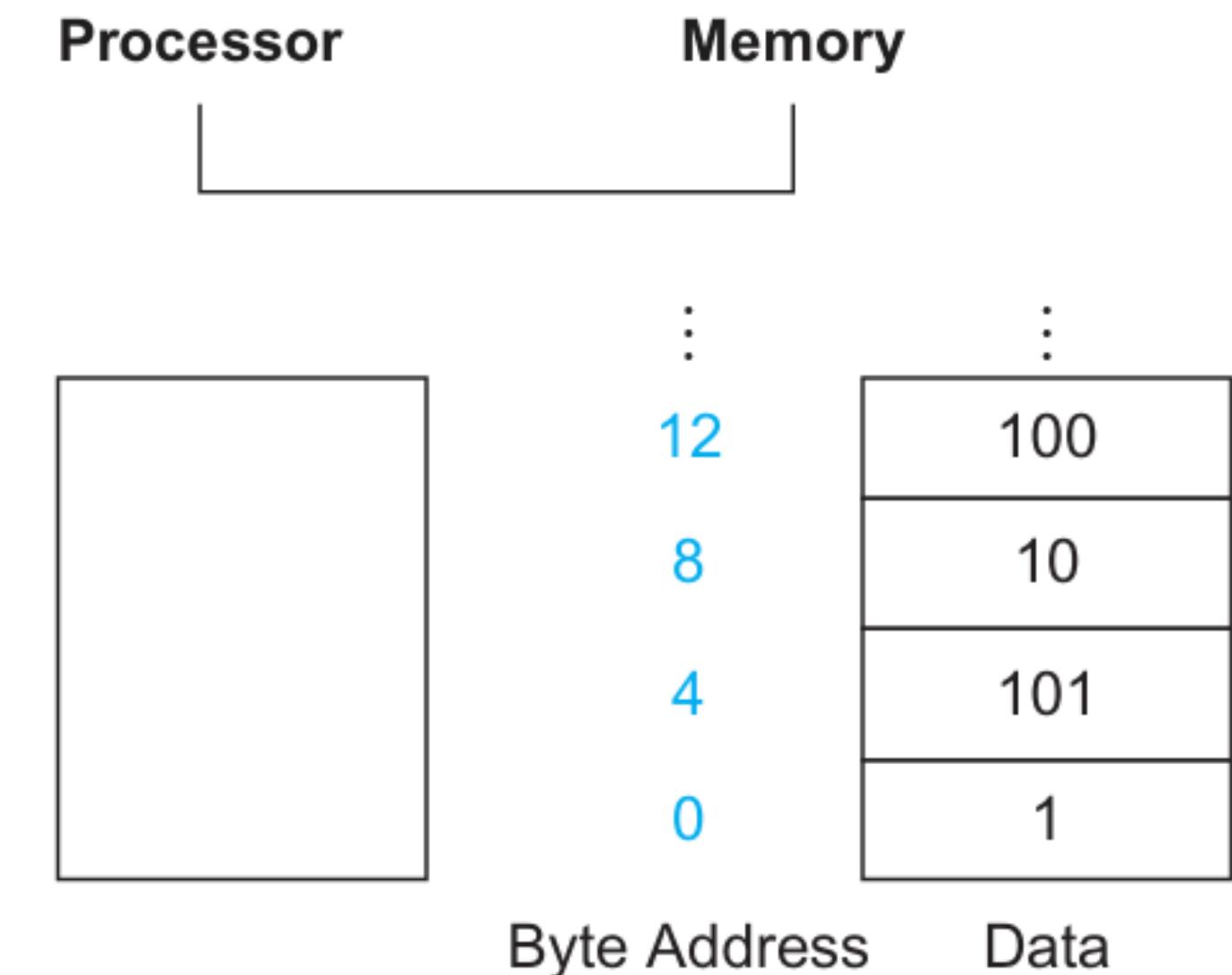
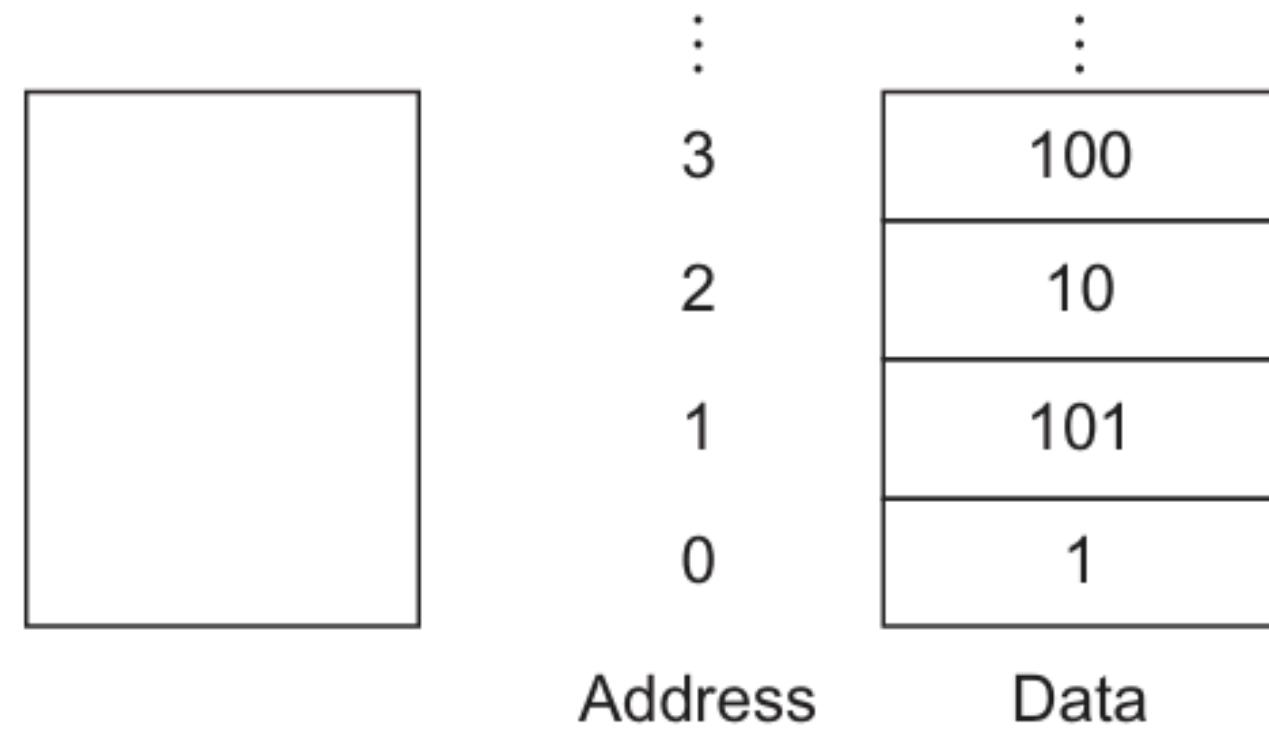


Memory Instructions: Word vs. Byte

lw \$t0, 1(\$s0)

lb \$t0, 1(\$s0)

- `lw` is interpreted as “load word”
- `lb` is “load byte”
- For the `lw`, we need the base+offset ($\$s0 + 1$) to be **always divisible by 4 — word alignment**
- Why?
- Nothing such for `lb`
- **What Lies Beneath?**
 - `lb` just read the byte in the calculated address
 - `lw` reads four consecutive bytes starting from the calculated address.
- Why word alignment — again, it simplifies hardware
OS, compiler....

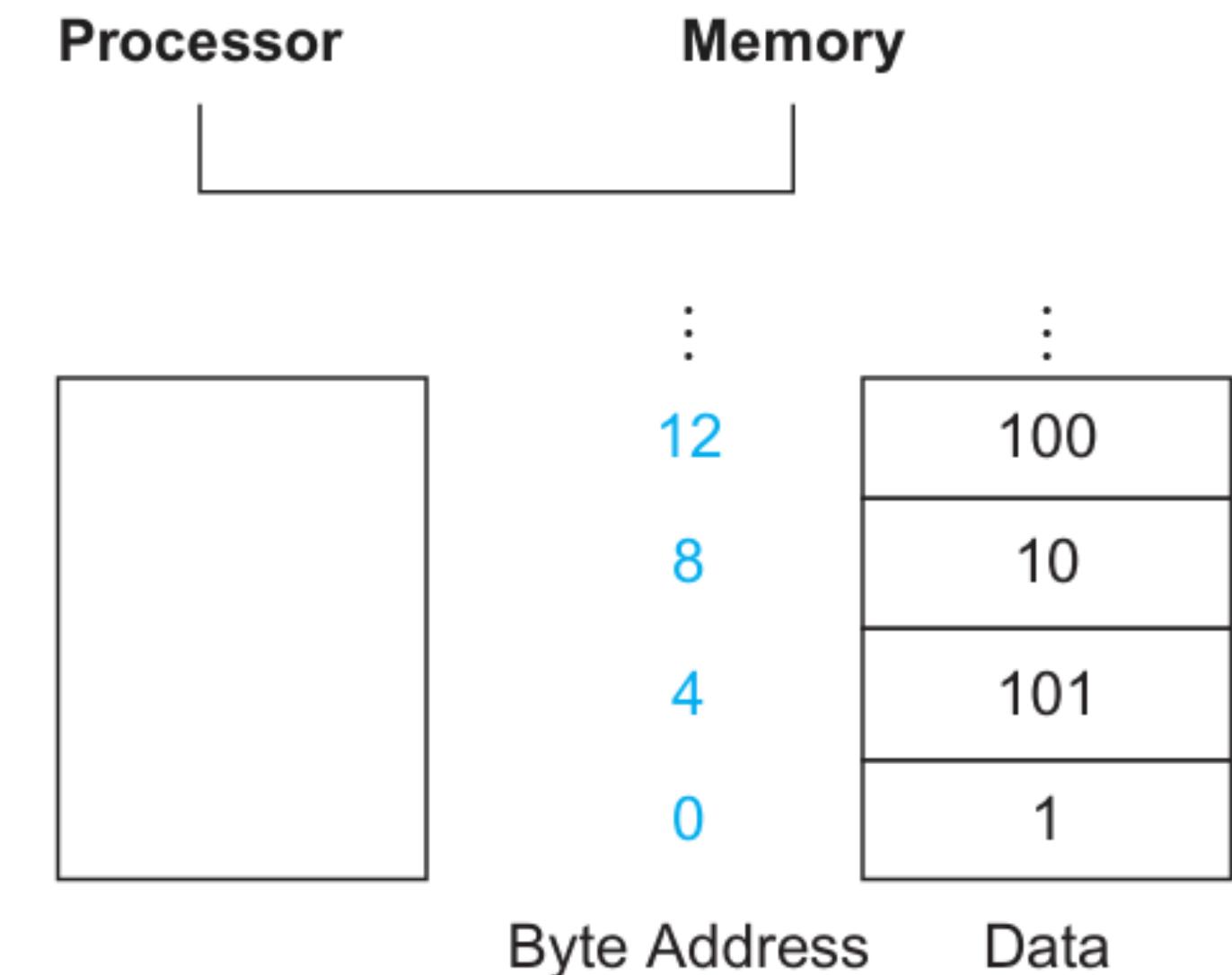
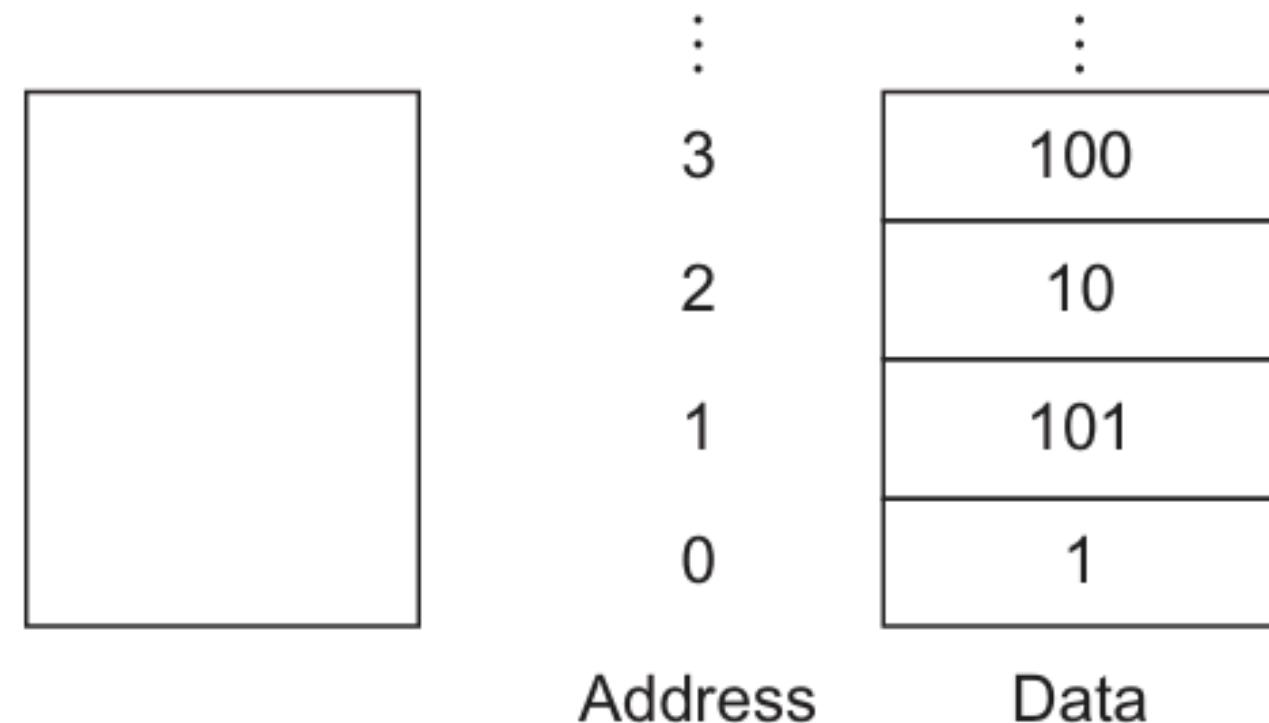


Memory Instructions: Word vs. Byte

lw \$t0, 1(\$s0)

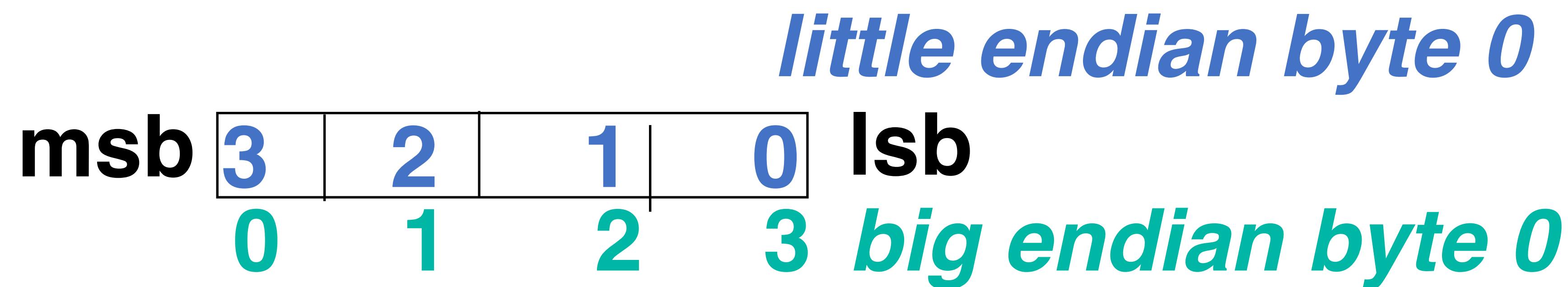
lb \$t0, 1(\$s0)

- `lw` is interpreted as “load word”
- `lb` is “load byte”
- For the `lw`, we need the base+offset ($\$s0 + 1$) to be **always divisible by 4 — word alignment**
- Why?
- Nothing such for `lb`
- **What Lies Beneath?**
 - `lb` just read the byte in the calculated address
 - `lw` reads four consecutive bytes starting from the calculated address.
- Why word alignment — again, it simplifies hardware OS, compiler....



Endianness (Byte ordering within a word)

- **Big Endian:** address of most significant byte = word address
(**xx00** = Big end of word), MIPS
- **Little Endian:** address of least significant byte = word address
(**xx00** = Little end of word), x86



Just for an example, do not take it for granted ...

```
unsigned int i = 1;  
char *c = (char*)&i; // reading the LSB  
Printf ("%d", *c);
```

```
unsigned int i = 12345678;  
char *c = (char*)&i;  
Printf ("%d", *c);
```

```
unsigned int i = 1;  
char *c = (char*)&i; // reading the LSB
```

```
Printf ("%d", *c);
```

Little endian: 1

Big endian: 0

```
unsigned int i = 12345678;
```

```
char *c = (char*)&i;
```

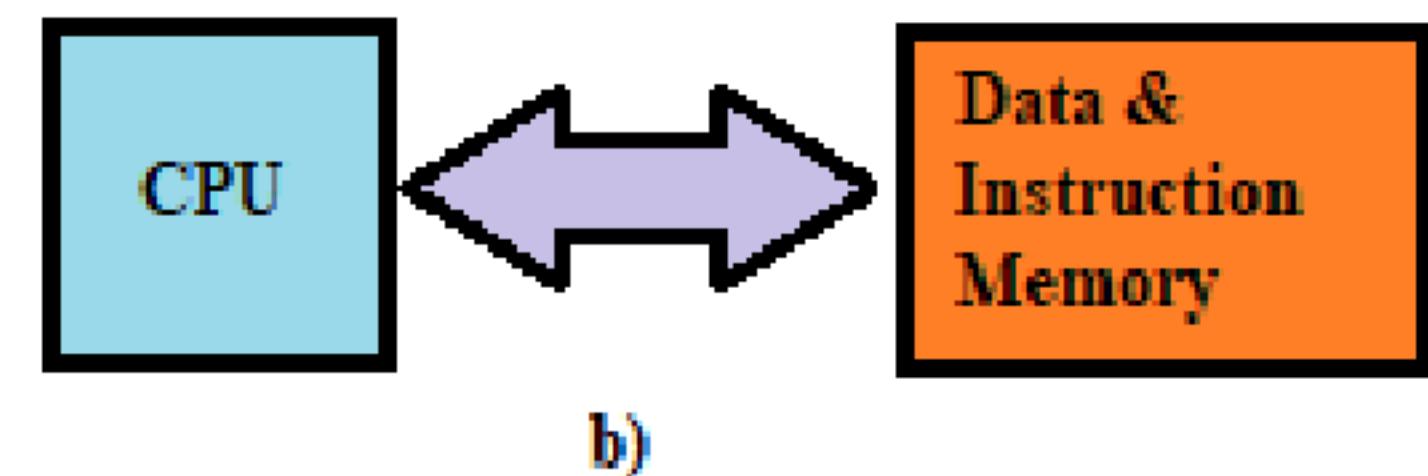
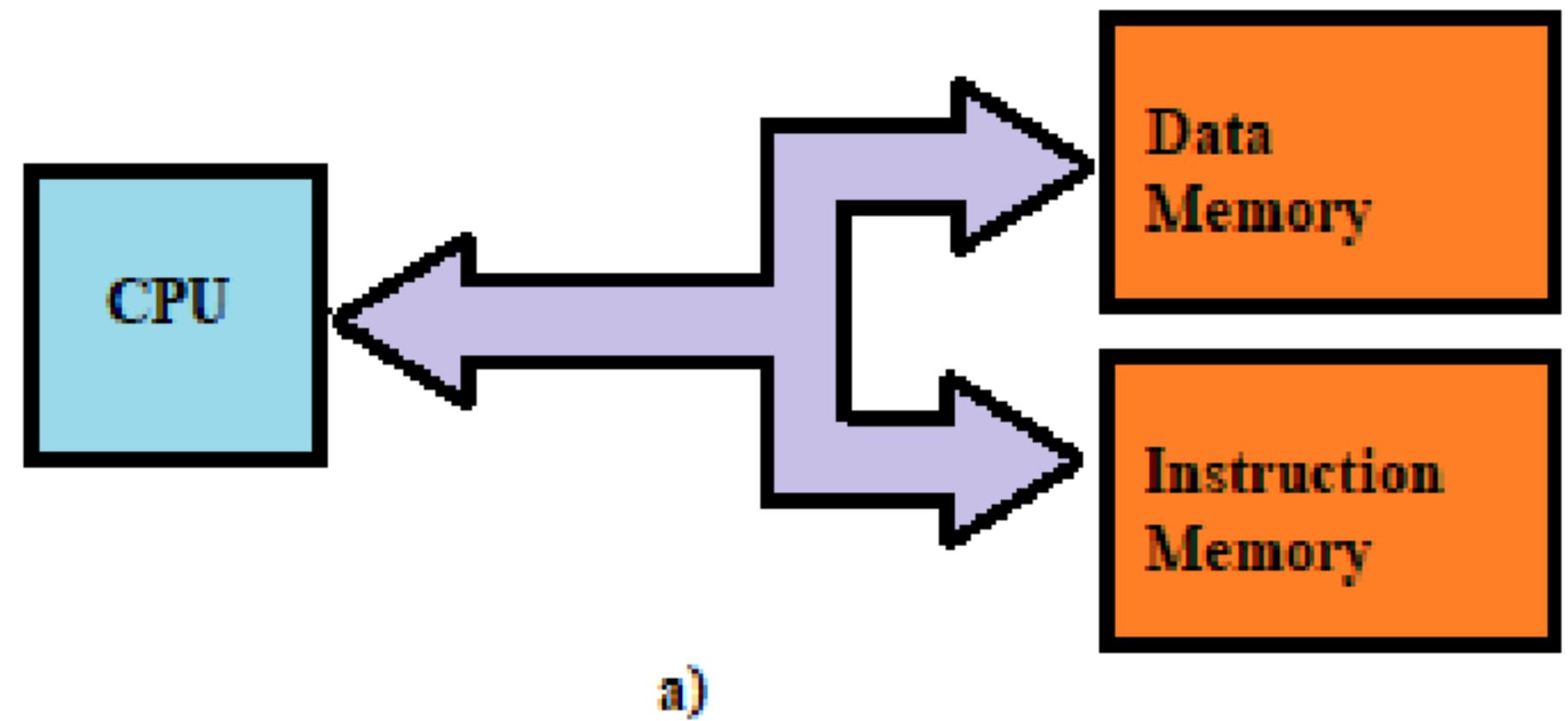
```
Printf ("%d", *c);
```

Little endian: 78

Big endian: 12

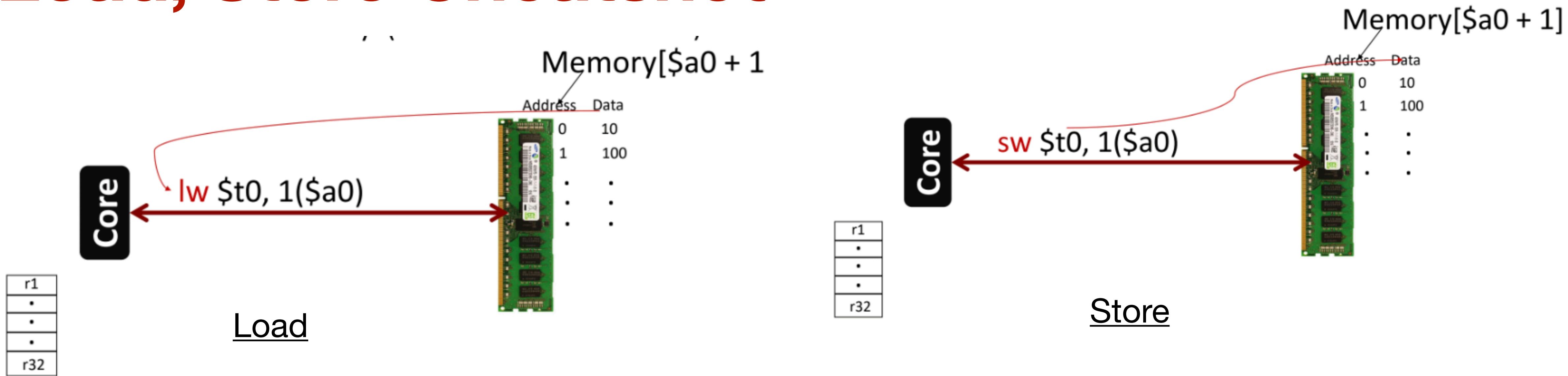
Another Important Point...

- Ok, **Von Neumann** said, data and code both are stored in the same memory.
 - In practice, this may lead to an issue — at a specific interval of time, **you can either fetch a data or an instruction.**
 - Affects parallelisation
 - **What if you separate the data and instruction memory and buses?**
 - That is called **Harvard Architecture.**
 - Modern commercial systems use a combination of both
 - RAM stores both instruction and data
 - But there are other intermediate memory (**caches**) which are separated for instruction and data



Source: Internet

Load, Store Cheatsheet



Program Counter

Points to the next instruction in the memory to be fetched

$g = h + A[8];$

PCX: `lw $t0, 8($3) # A[8]`
PCY: `add $s1, $s2, $t0 # g = h + t0`

$PCY = PCX + 4$



Load+Store+Instruction-fetch

Summary...

- Data and instructions at the same place
 - Registers are limited — 32 bit wide
 - Instructions are 32 bit wide
 - Registers are accessed by names
 - Memory is accessed by addresses



Decision Making...

- If, else statements in your program...
 - How they are interpreted as instructions??

beq (branch equals to) and
bne (branch not equals to)

beq \$t0, \$t1, L1
bne \$t0, \$t1, L1



Decision Making...

beq \$t0, \$t1, L1

goto L1 (statements labeled as L1) if \$t0 equals \$t1

bne \$t0, \$t1, L1

goto L1 (statements labeled as L1) if \$t0 does not equal to \$t1



Simple Example...

- Let's compile:

- `if (i == j) f = g + h; else f = g - h;`

Assume:

`$s0` has `i`, `$s1` has `j`, `$s2` has `g`, `$s3` has `h`, `$s4` has `f`

Unconditional Jump

jumps to a specific label

```
beq $s0, $s1, if_equal          # if i == j, jump to if_equal
sub $s4, $s2, $s3                  # else: f = g - h
j end_if                         # jump to end
if_equal:
    add $s4, $s2, $s3              # f = g + h
end_if:
```

Decision Making...

- So you can check conditions:
 - If ($x = 0$) ..
 - If ($x \neq 0$) ..
 - If ($x = y$) ..
 - If ($x \neq y$) ...
- But how about the following code??

```
if (a < b)  
    c=1  
else  
    c=0
```



Decision Making...

if (a < b)

c=1

else

c=0

- Set on less than (slt)
 - slt \$t0, \$s3, \$s4 # \$t0 = 1 if \$s3 < \$s4
 - slti \$t0,\$s2,10 # \$t0 = 1 if \$s2 < 10
- After using slt, we can use the beq or bne

Simple Example...

- Let's compile:

- `if (i < j) f = g + h; else f = g - h;`

Assume:

`$s0` has `i`, `$s1` has `j`, `$s2` has `g`, `$s3` has `h`, `$s4` has `f`

```
    slt $t0, $s0, $s1          # $t0 = 1 if i < j
    beq $t0, $zero, ELSE      # if $t0 == 0, i >= j, jump to ELSE
    add $s4, $s2, $s3          # f = g + h
    j END_IF                  # jump to END_IF
```

ELSE:

```
    sub $s4, $s2, $s3          # f = g - h
```

END_IF:

Dealing With Loops

- Let's first see how we deal with **arrays**...

- $f = h + A[8]$

Assume:

- \$t0 has $A[8]$, \$s5 has base address of the array A, \$s4 has f, \$s3 has h
 - Also assume “A[8]” as `uint8_t` (a byte)

```
lbu $t0, 8($s5)          # Load word A[8] with byte offset  
add $s4, $s3, $t0        # f = h + A[8]
```

- But what is “A[8]” is int (4 bytes) ?????

```
lw $t0, 32($s5)          # Load A[8], 8 * 4 = 32 (word) offset  
add $s4, $s3, $t0        # f = h + A[8]
```

Dealing With Loops

- Let's consider:

- `while (A[i] > k) i = i+1;`

Assume:

\$s0 has i,

\$t1 has address of A[i]

\$t2 has A[i]

\$s6 has k

```
LOOP:  
    sll $t1, $s0, 2          # $s0 = i, i*4 for word offset  
    add $t1, $s5, $t1        # Compute address A[i]  
    lw $t2, 0($t1)           # Load A[i] (integer)  
    slt $t3, $t2, $s6        # $t3 = 1 if A[i] < k  
    bne $t3, $zero, END_LOOP # if A[i] < k, exit loop  
    addi $s0, $s0, 1          # i = i + 1  
    j LOOP  
END_LOOP:
```

Performs left logical shift by two bits..why??

Dealing With Loops

- What happens if:
 - `while (A[i] == k) i = i+1;`

Assume:

`$s0 has i,`

`$t1 has address of A[i]`

`$t2 has A[i]`

`$s6 has k`

Dealing With Loops

- What happens if:
 - `while (A[i] == k) i = i+1;`

Assume:

`$s0` has `i`,

`$t1` has address of `A[i]`

`$t2` has `A[i]`

`$s6` has `k`

LOOP:

```
sll $t1, $s0, 2  
add $t1, $s5, $t1  
lw $t2, 0($t1)  
bne $t2, $s6, END_LOOP  
addi $s0, $s0, 1  
j LOOP
```

END_LOOP:

```
# $s0 = i, i*4 for word offset  
# Compute address A[i]  
# Load A[i] (integer)  
# if A[i] != k, exit loop  
# i = i + 1
```

More on Jumping...

- What happens if:

- `while (A[i] == k) i = i+1;`

Assume:

`$s0` has `i`,

`$t1` has address of `A[i]`

`$t2` has `A[i]`

`$s6` has `k`

```
LOOP:  
    sll $t1, $s0, 2  
    add $t1, $s5, $t1  
    lw $t2, 0($t1)  
    bne $t2, $s6, END_LOOP  
    addi $s0, $s0, 1  
    j LOOP  
END_LOOP:
```

```
# $s0 = i, i*4 for word offset  
# Compute address A[i]  
# Load A[i] (integer)  
# if A[i] != k, exit loop  
# i = i + 1
```

- Normally:

- `PC, PC+4, PC+8,...`
- But jump instruction loads a new value to the `PC`
 - It's the offset in the program where the exception should divert (the label is basically that)

More on Jumping...

- What happens if:

- `while (A[i] == k) i = i+1;`

Assume:

`$s0` has `i`,

`$t1` has address of `A[i]`

`$t2` has `A[i]`

`$s6` has `k`

```
LOOP:  
    sll $t1, $s0, 2  
    add $t1, $s5, $t1  
    lw $t2, 0($t1)  
    bne $t2, $s6, END_LOOP  
    addi $s0, $s0, 1  
    j LOOP  
END_LOOP:
```

```
# $s0 = i, i*4 for word offset  
# Compute address A[i]  
# Load A[i] (integer)  
# if A[i] != k, exit loop  
# i = i + 1
```

- Normally:
 - PC, PC+4, PC+8,....
 - But jump instruction loads a new value to the PC
 - It's the offset in the program where the exception should divert (the label is basically that)
 - But jumping is even more exotic...Let's see why

More on Jumping...



shutterstock.com · 1981245995

More on Jumping...Working with Functions

```
int sum(int a, int b)
{
    int c=a+b;
    return c;
}
void main (void)
{
    int i=1;
    int j=2;
    int k = sum(i,j);
    // .....
}
```

Function call
jumps to a
location in
your code

- **Caller:** One who calls the function
- **Callee:** The function which is being called

- **Anatomy of a Function Call:**

- Put parameters in a place where the function can access them.
- Transfer control to the function.
- Acquire the storage resources needed for the function.
- Perform the desired task.
- Put the result value in a place where the caller program can access it.
- Return control to the point of origin, since a function can be called from several points in a program.

Working with Functions – The MIPS Case

- **MIPS Support for Function Call:**

- \$a0–\$a3: four argument registers in which to pass parameters
- \$v0–\$v1: two value registers in which to return values
- \$ra: one **return address** register to return to the point of origin

- **Ways of Jumping..:**

- jal Label: Jump and link
- jr \$ra: Jump back to the return address stored in \$ra



Working with Functions – The MIPS Case

- Ways of Jumping..:

- jal Label:
 - First, save PC+4 in \$ra
 - The instruction to be executed next is at Label
- jr \$ra: Jump back to the return address stored in \$ra
(PC + 4)



Working with Functions – The MIPS Case

```
int sum(int a)
{
    int c=a+4;
    return c;
}
void main (void)
{
    int i=2;
    int k = sum(i);
}
```

Complete Picture

sum:		
PC+100: addi \$v0, \$a0, 4	# c = a + 4, return in \$v0	
PC+104: jr \$ra	# return to PC+12	
main:		
PC+4: li \$a0, 2	# i = 2	
PC+8: jal sum	# call sum(i); \$ra = PC+12	
PC+12: addi \$s1, \$v0, 0	# k = return value (k = 6)	

Working with Functions – The MIPS Case

- **MIPS Support for Function Call:**

- \$a0–\$a3: four argument registers in which to pass parameters
- \$v0–\$v1: two value registers in which to return values
- \$ra: one **return address** register to return to the point of origin

- **Ways of Jumping..:**

- jal Label: Jump and link
- jr \$ra: Jump back to the return address stored in \$ra



Functions – More Parameters

- What if there are more than 4 parameters?
- What if the function needs more registers (beyond two return registers) to operate?
- **Remember:** caller must have its state restored after callee finishes.

```
int leaf_example (int g, int h, int i, int j, int k)
{
    int f;
    f = (g + h) - (i + j) + k;
    return f;
}

void main (void)
{
    int g=2;
    int h=3;
    int i=1;
    int j=1;
    int k=3;
    int l = leaf_example(g,h,i,j,k);
}
```

Functions – More Parameters

```
int leaf_example (int g, int h, int i, int j,  
int k)  
{  
    int f;  
    f = (g + h) - (i + j) + k;  
    return f;  
}  
  
void main (void)  
{  
    int g=2;  
    int h=3;  
    int i=1;  
    int j=1;  
    int k=3;  
    int l = leaf_example(g,h,i,j,k);  
}
```

leaf example:

```
addi $sp, $sp, -8          # make space for s0, t0  
sw $s0, 4($sp)             # save s0  
sw $t0, 0($sp)             # save t0
```

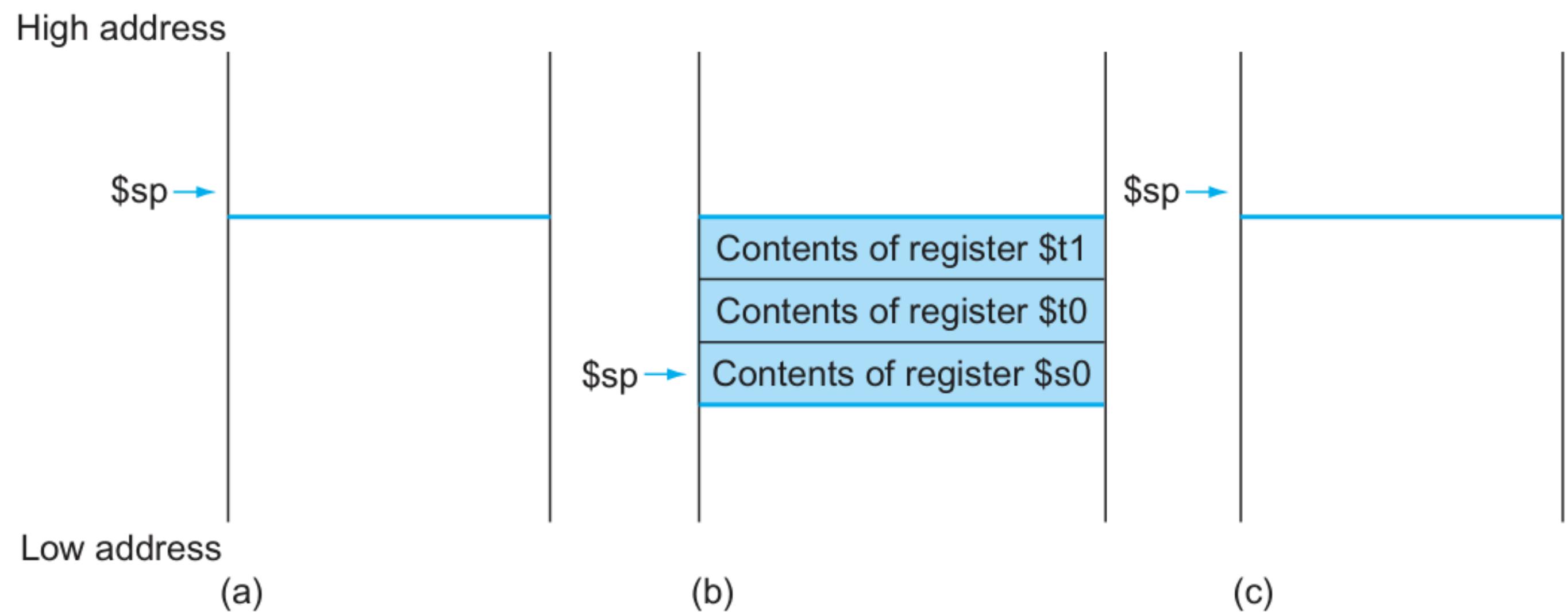
```
add $s0, $a0, $a1          # s0 = g + h  
add $t0, $a2, $a3          # t0 = i + j  
sub $v0, $s0, $t0           # v0 = (g + h) - (i + j)  
add $v0, $v0, $t2           # v0 += k (f = ... + k)
```

```
lw $t0, 0($sp)              # restore t0  
lw $s0, 4($sp)              # restore s0  
addi $sp, $sp, 8            # deallocate stack  
jr $ra                      # return
```

What is this
trick?

Stack – What is it?

- A region in DRAM
- Grows in the lower direction
- Push and Pop are the main operations
- **Stack pointer:** A special register
 - A value denoting the most recently allocated address in a stack that shows where registers should be spilled or where old register values can be found.



Functions – More Parameters

- **MIPS Support for Function Call:**

- We need registers beyond \$a3
 - Use other registers \$s0, \$t0
 - But the caller function might be already have some values in them.
- **Solution:** store them in memory (stack) before using in the function.
 - Restore back before exiting
- This idea of saving registers is called **register spilling**

leaf example:

A red arrow points from the question "What is this trick?" up towards the first three lines of assembly code. Another red arrow points from the question down towards the last four lines of assembly code.

```
addi $sp, $sp, -8      # make space for s0, t0
sw $s0, 4($sp)          # save s0
sw $t0, 0($sp)          # save t0
```



```
add $s0, $a0, $a1      # s0 = g + h
add $t0, $a2, $a3      # t0 = i + j
sub $v0, $s0, $t0      # v0 = (g + h) - (i +
add $v0, $v0, $t2      # v0 += k (f = ... + k)
```



```
lw $t0, 0($sp)          # restore t0
lw $s0, 4($sp)          # restore s0
addi $sp, $sp, 8        # deallocate stack
jr $ra                  # return
```

Nested Procedures

```
main() {  
    a = a + f1(a);  
}  
  
f1(a) {  
    a = a - f2(a); return a; }  
  
f2(a) {  
    a = a + f3(a); return a; }  
  
f3(a) {  
    a = a + 1;      return a; }
```

f1:
f2's argument in \$a0 to \$a3
jal f2

Nested Procedures

f1:

f2's argument in \$a0 to \$a3

jal f2

...

f2:

f3's argument in \$a0 to \$a3

jal f3

...

Nested Procedures

f1:

PC: f2's argument in \$a0 to \$a3

PC+4: jal f2 // \$ra = PC+8

...

f2:

PC+100: f3's argument in \$a0 to \$a3

PC+104: jal f3 // \$ra = PC+108

f3: ...

jr \$ra

Nested Procedures

f1:

PC: f2's argument in \$a0 to \$a3

PC+4: jal f2 // \$ra = PC+8

...

f2:

PC+100: f3's argument in \$a0 to \$a3

PC+104: jal f3 // \$ra = PC+108

jr \$ra ⊕ Oh no!!

f3: ...

jr \$ra

Nested Procedures

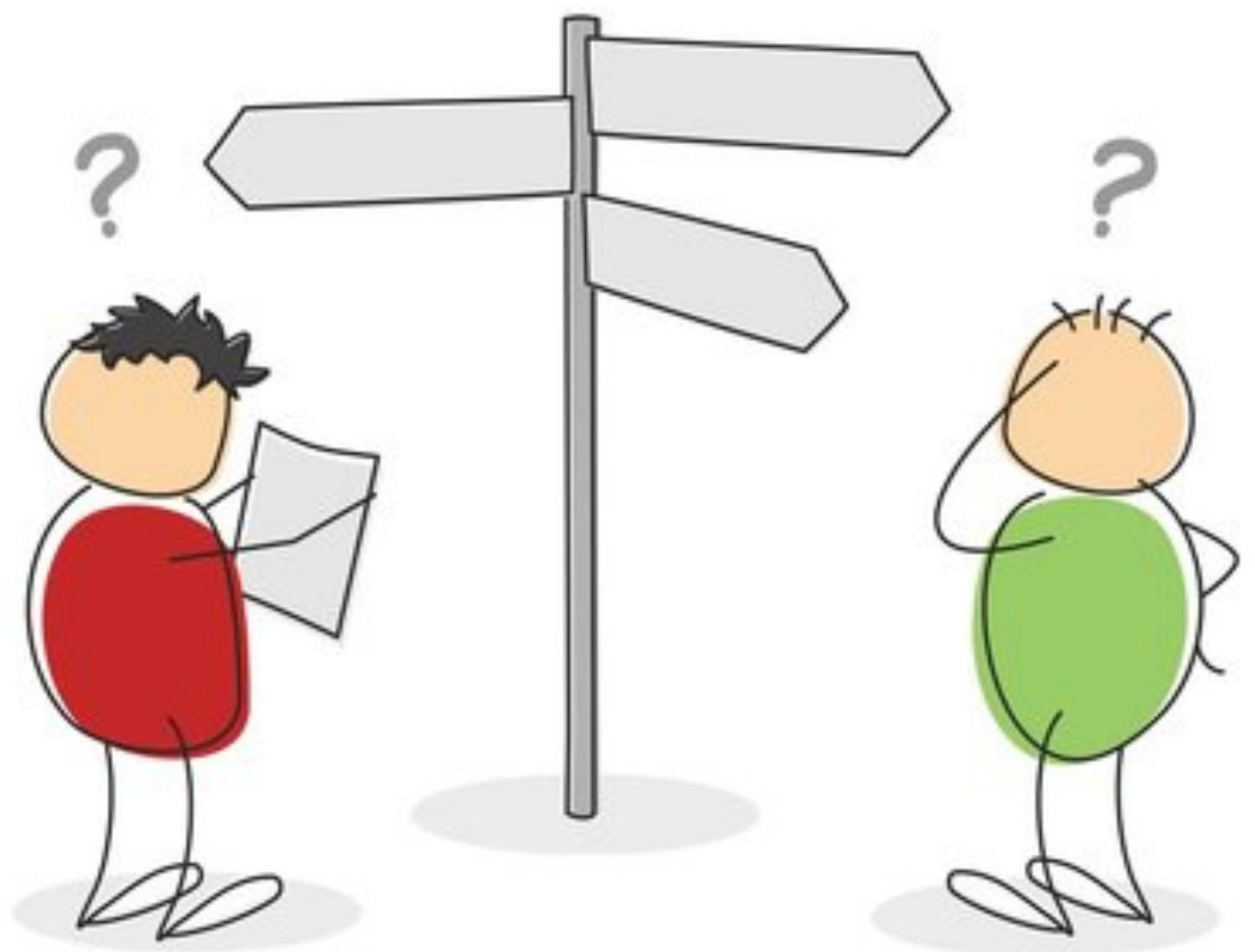
caller registers

callee registers

Why?

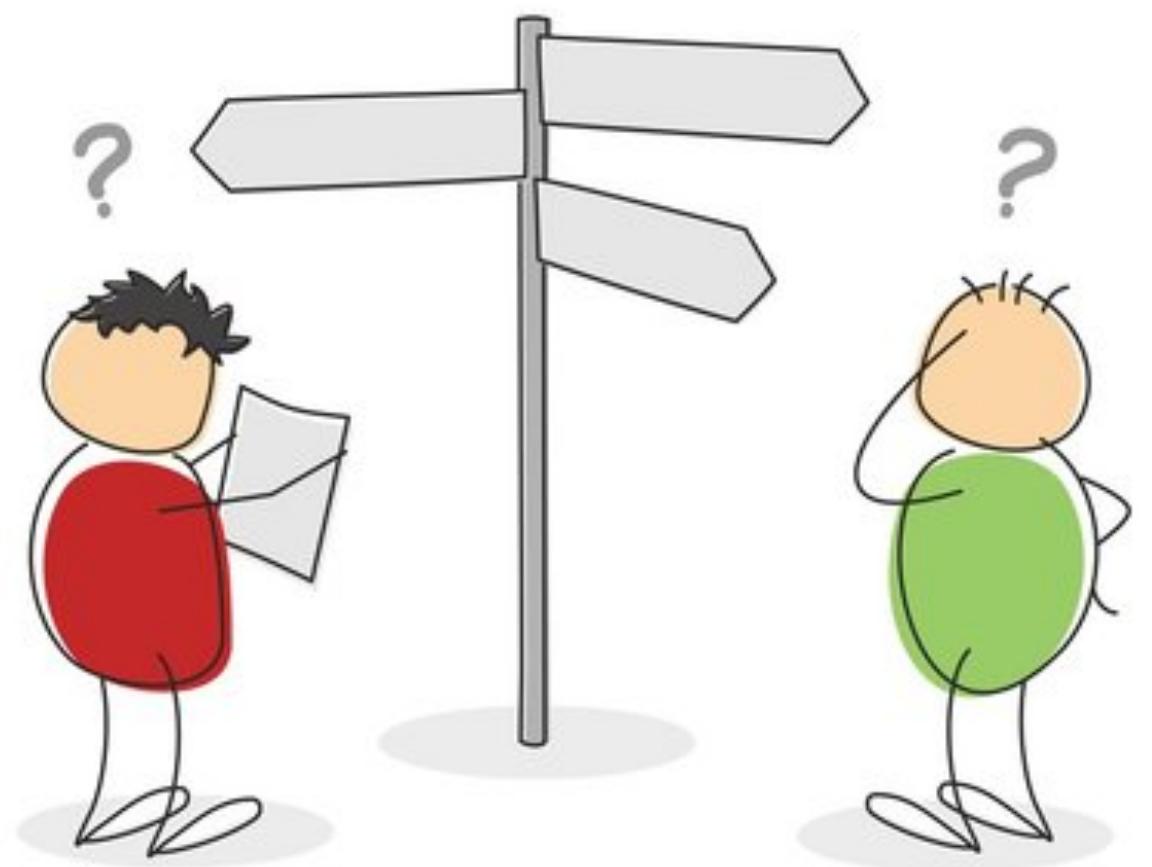
Callee does not know, registers used by callers, can be many callers too

Caller does not know the callee's plan ☺

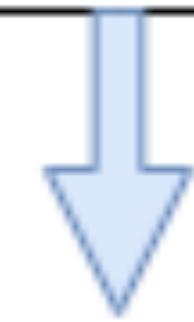


What to do?

- Push all the other registers that must be preserved onto the stack
 - Caller pushes any argument registers ($\$a0-\$a3$) or temporary registers ($\$t0-\$t9$) that are needed after the call.
 - Callee pushes the return address register $\$ra$ and any saved registers ($\$s0-\$s7$) used by the callee.
 - The $\$sp$ is adjusted to account for the number of registers put on the stack.
 - Everything is restored after the call



Stack



Heap

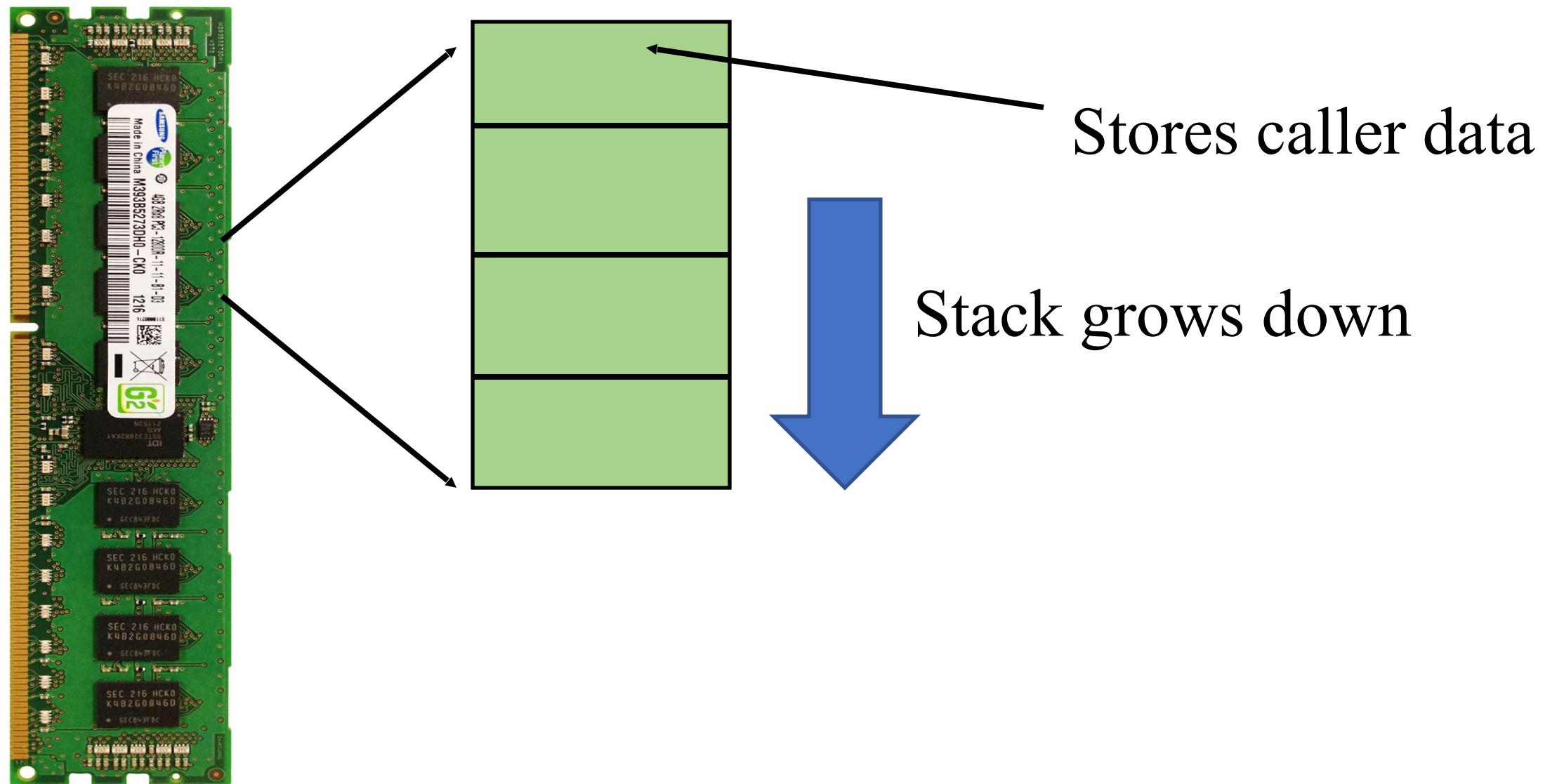
Data

Text

The loaded program

- System program that loads the executable into the memory.
- Every executable has a text, heap/stack data segments

MIPS way of handling it: The Stack (part of DRAM, for each function call)



`$sp` (stack pointer) points to the address where stack ends

One per function, private memory area, else the same problem ☹

Caller Save
If the caller uses these register, then the caller must stave them in case the callee overwrites them.

R0	\$0
R1	\$at
R2	\$v0
R3	\$v1
R4	\$a0
R5	\$a1
R6	\$a2
R7	\$a3
R8	\$t0
R9	\$t1
R10	\$t2
R11	\$t3
R12	\$t4
R13	\$t5
R14	\$t6
R15	\$t7

Constant 0

Reserved Temp.

Return Values

Procedure arguments

Caller Save

Temporaries:
May be
overwritten
by called
procedures

R16	\$s0
R17	\$s1
R18	\$s2
R19	\$s3
R20	\$s4
R21	\$s5
R22	\$s6
R23	\$s7
R24	\$t8
R25	\$t9
R26	\$k0
R27	\$k1
R28	\$gp
R29	\$sp
R30	\$fp
R31	\$ra

Callee Save

Temporaries:
May not be
overwritten by
called pro-
cedures

Caller Save

Temp

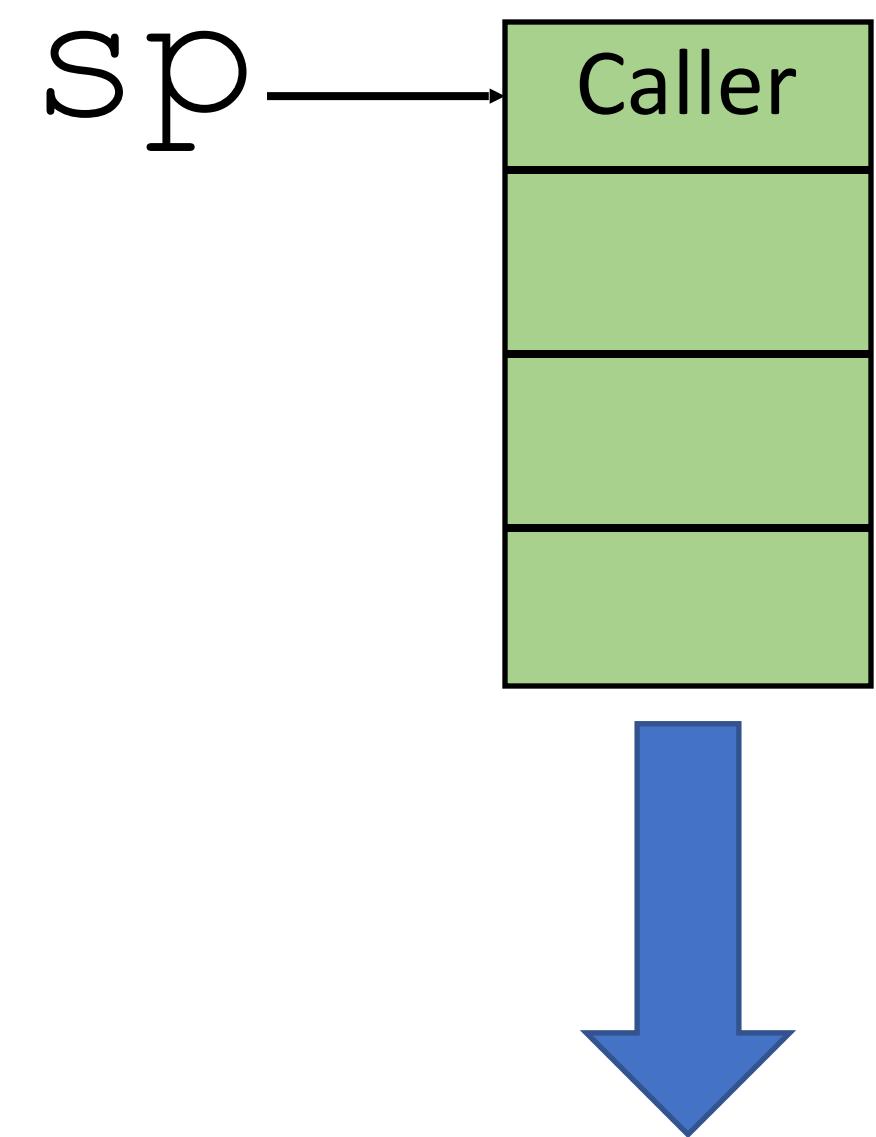
Reserved for
Operating Sys
Global Pointer

Callee Save
Stack Pointer
Frame Pointer
Return Address

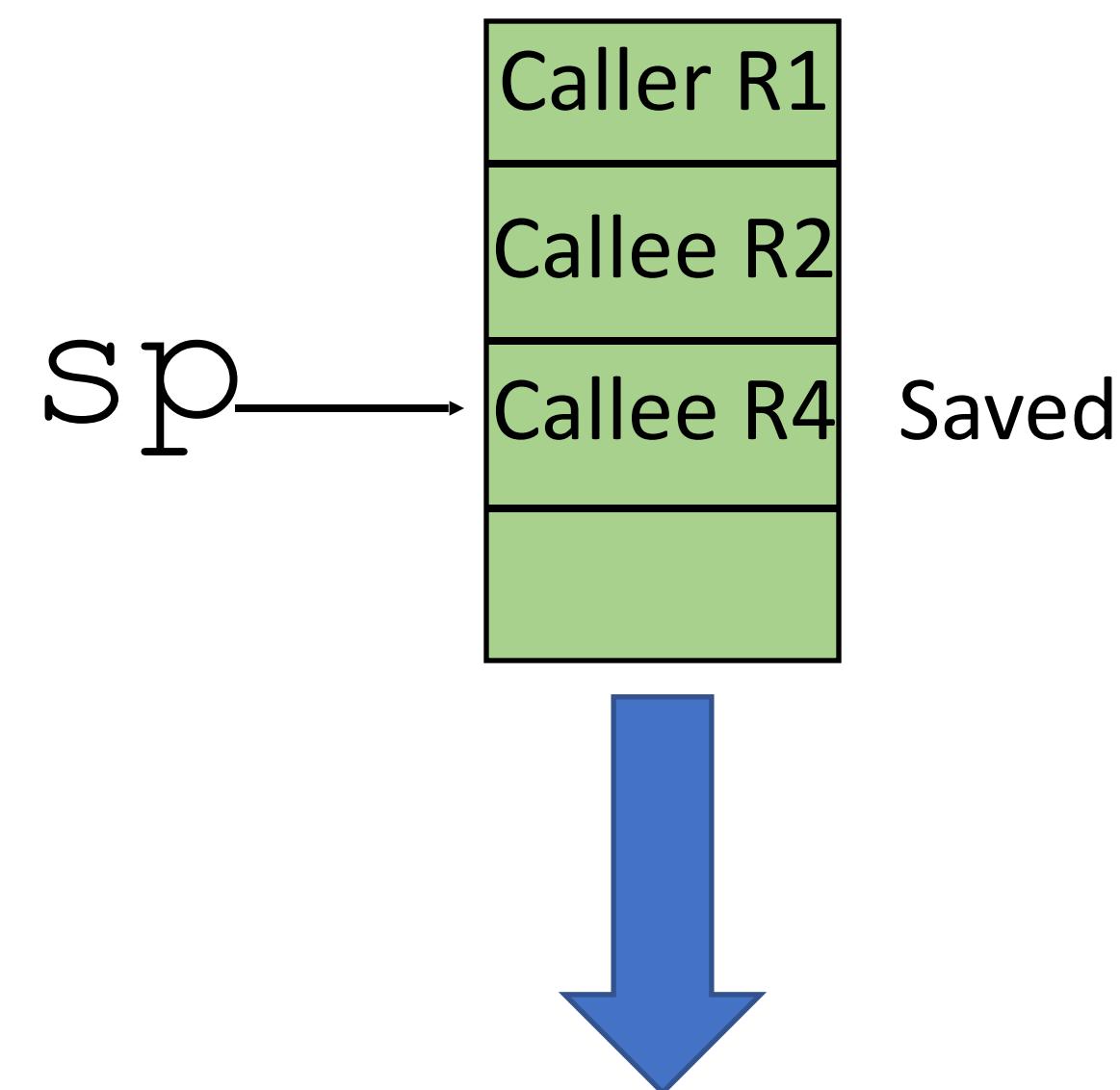
Callee Save

If the callee uses these register, then the callee must save and restore them in case the caller uses them.

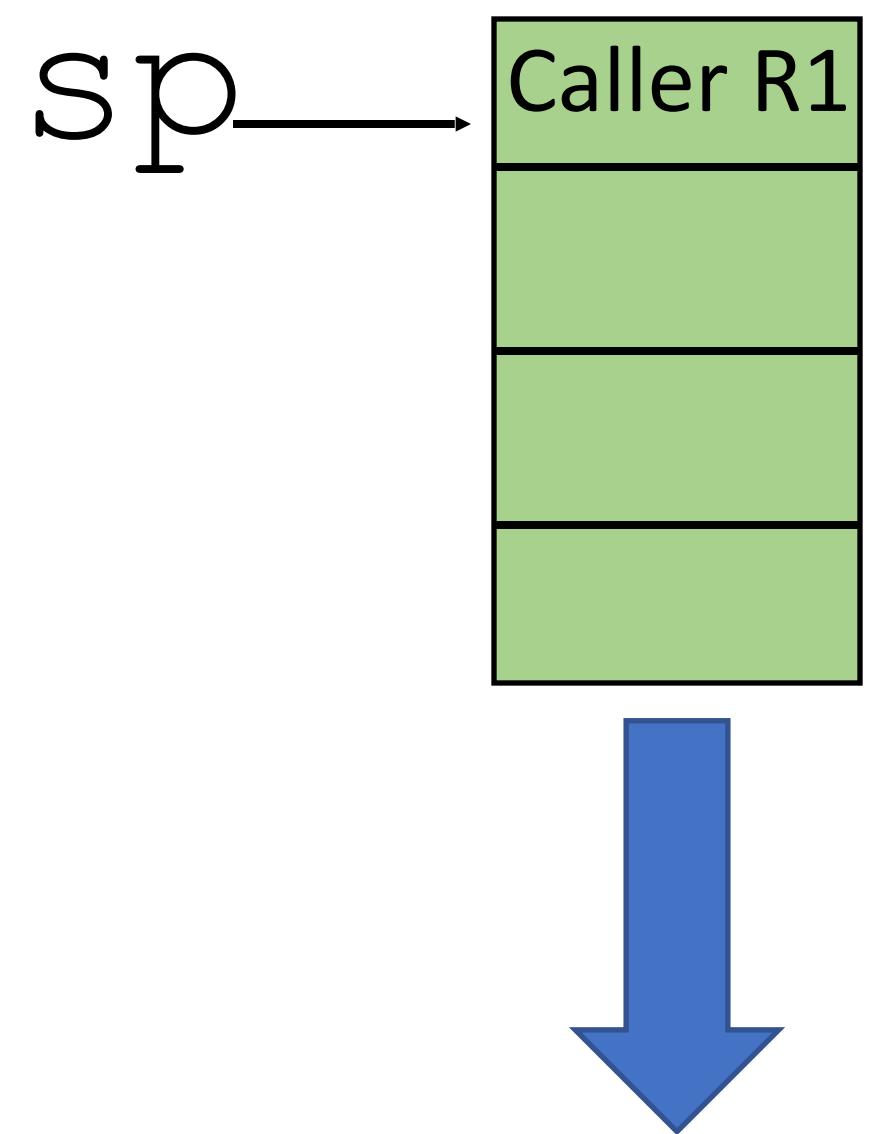
MIPS way of handling it: Before function call



MIPS way of handling it: Function call is ON

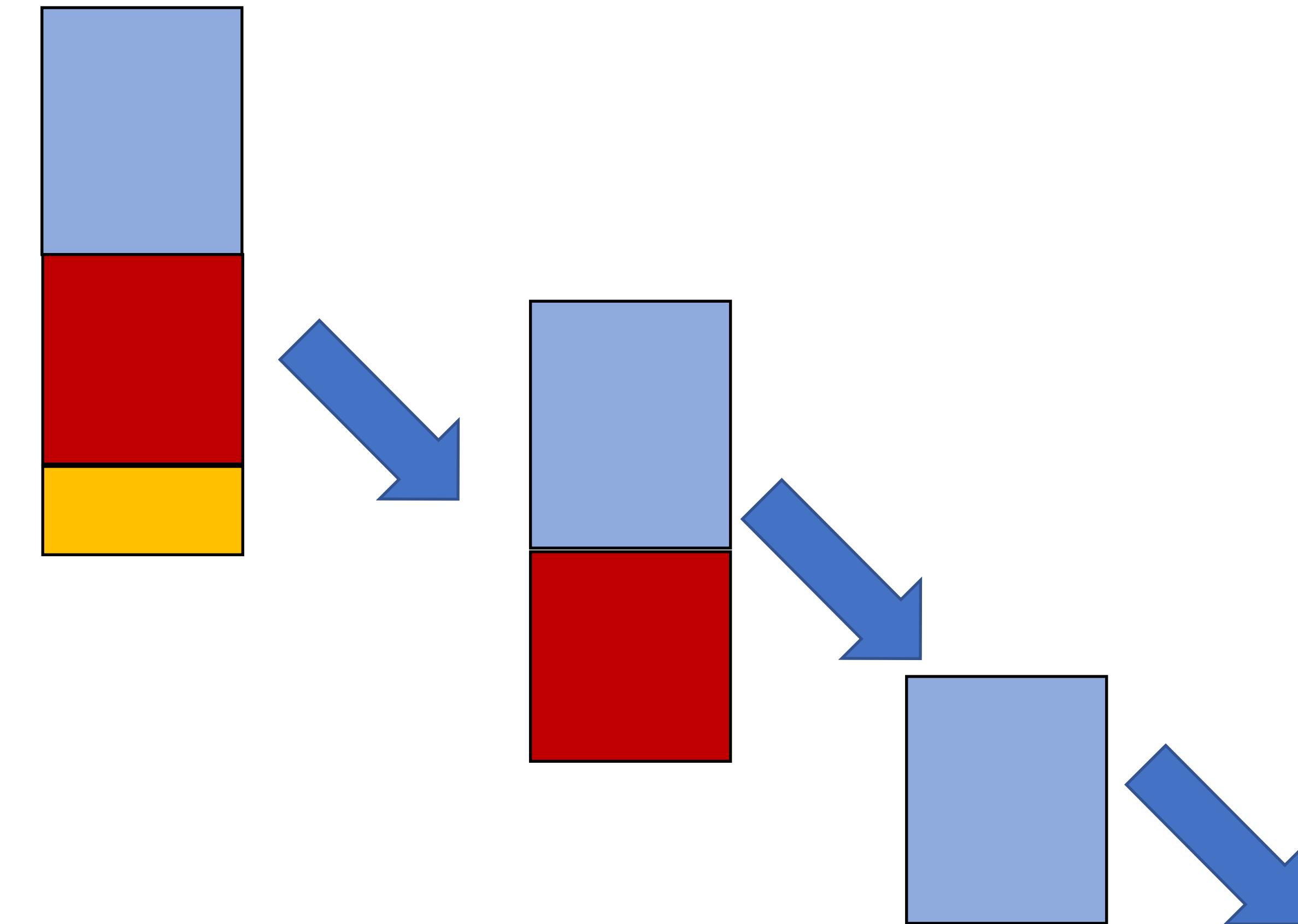


MIPS way of handling it: After the function call



Nested Functions (Remember main() is a function too ☺)

```
CS230 // jal cs230
{
    CS330 // jal cs330
    {
        CS430 // jal cs430
        {
            } //jr
        } //jr
    } // jr
```

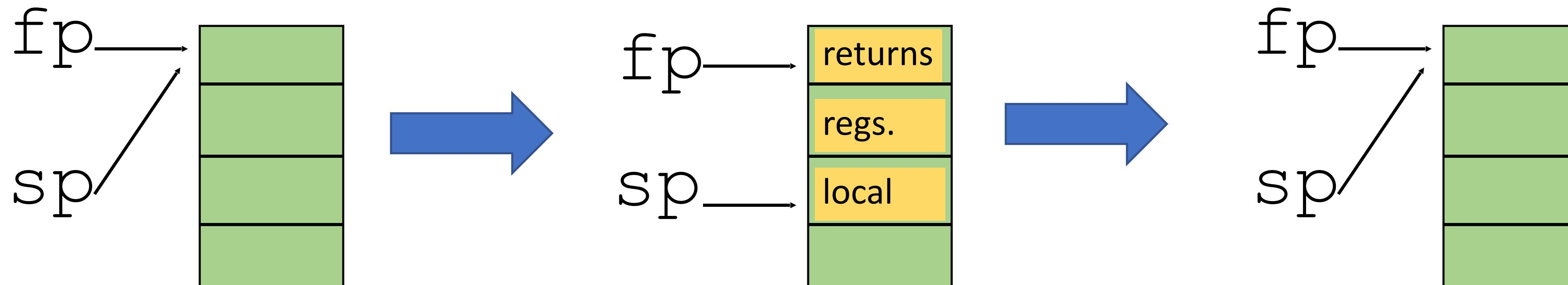


The final one: Frame pointer

- Stack also stores local variables and data structures (local arrays and structures) for a function along with the **return address(es)**.
- Frame pointer (\$fp) will get incremented and decremented based on the local arguments used.

The final one: Frame pointer

Frame pointer: Points to local variables and saved registers. Points to the **highest address** in the function frame. Stays there throughout the procedure. Stack pointer, **moves** around.



A Complete Example: Recursion

```
fact:  
1    addi $sp, $sp, -8          # make space for ra and n  
     sw $ra, 4($sp)           # save return address  
     sw $a0, 0($sp)           # save n  
  
int fact (int n) {  
    if (n < 1) return (1);  
    else return (n * fact(n - 1)); }  
  
RECURSE:  
        slti $t0, $a0, 1          # t0 = 1 if n < 1  
        beq $t0, $zero, RECURSE  # if n >= 1, do recursion  
        li $v0, 1                 # base case: return 1  
        addi $sp, $sp, 8           # deallocate stack  
        jr $ra                   # return  
  
        addi $a0, $a0, -1          # n - 1  
        jal fact                 # recursive call  
  
        lw $a0, 0($sp)            # restore n  
        mul $v0, $a0, $v0          # n * fact(n - 1)  
        lw $ra, 4($sp)            # restore return address  
        addi $sp, $sp, 8           # deallocate stack  
        jr $ra                   # return
```

A Complete Example: Recursion

```
fact:  
1    addi $sp, $sp, -8          # make space for ra and n  
     sw $ra, 4($sp)           # save return address  
     sw $a0, 0($sp)           # save n  
  
int fact (int n) {  
    if (n < 1) return (1);  
    else return (n * fact(n - 1)); }  
  
     slti $t0, $a0, 1          # t0 = 1 if n < 1  
     beq $t0, $zero, RECURSE  # if n >= 1, do recursion  
  
     li $v0, 1                 # base case: return 1  
     addi $sp, $sp, 8          # deallocate stack  
     jr $ra                   # return  
  
RECURSE:  
2    addi $a0, $a0, -1          # n - 1  
     jal fact                 # recursive call  
  
     lw $a0, 0($sp)           # restore n  
     mul $v0, $a0, $v0         # n * fact(n - 1)  
     lw $ra, 4($sp)           # restore return address  
     addi $sp, $sp, 8          # deallocate stack  
     jr $ra                   # return
```

A Complete Example: Recursion

```
int fact (int n) {
    if (n < 1) return (1);
    else return (n * fact(n - 1)); }
```

fact:

1 addi \$sp, \$sp, -8 # make space for ra and n
 sw \$ra, 4(\$sp) # save return address
 sw \$a0, 0(\$sp) # save n

slti \$t0, \$a0, 1 # t0 = 1 if n < 1
beq \$t0, \$zero, RECURSE # if n >= 1, do recursion

3 li \$v0, 1 # base case: return 1
 addi \$sp, \$sp, 8 # deallocate stack
 jr \$ra # return

RECURSE:

2 addi \$a0, \$a0, -1 # n - 1
 jal fact # recursive call

lw \$a0, 0(\$sp) # restore n
mul \$v0, \$a0, \$v0 # n * fact(n - 1)
lw \$ra, 4(\$sp) # restore return address
addi \$sp, \$sp, 8 # deallocate stack
jr \$ra # return

A Complete Example: Recursion

```
int fact (int n) {  
    if (n < 1) return (1);  
    else return (n * fact(n - 1)); }
```

fact:

1

```
addi $sp, $sp, -8          # make space for ra and n  
sw $ra, 4($sp)            # save return address  
sw $a0, 0($sp)            # save n
```

3

```
slti $t0, $a0, 1          # t0 = 1 if n < 1  
beq $t0, $zero, RECURSE  # if n >= 1, do recursion  
  
li $v0, 1                 # base case: return 1  
addi $sp, $sp, 8           # deallocate stack  
jr $ra                     # return
```

RECURSE:

2

```
addi $a0, $a0, -1          # n - 1  
jal fact                  # recursive call
```

4

```
lw $a0, 0($sp)             # restore n  
mul $v0, $a0, $v0           # n * fact(n - 1)  
lw $ra, 4($sp)              # restore return address  
addi $sp, $sp, 8             # deallocate stack  
jr $ra                     # return
```

- Do you notice something unusual??

A Complete Example: Recursion

```
int fact (int n) {
    if (n < 1) return (1);
    else return (n * fact(n - 1)); }
```

- Can you tell me why there I do not `lw $ra` in the base case???

- `$ra` does not change in the base case

fact:

1

```
addi $sp, $sp, -8          # make space for ra and n
sw $ra, 4($sp)             # save return address
sw $a0, 0($sp)             # save n
```

```
slti $t0, $a0, 1           # t0 = 1 if n < 1
beq $t0, $zero, RECURSE   # if n >= 1, do recursion
```

3

```
li $v0, 1                  # base case: return 1
addi $sp, $sp, 8            # deallocate stack
jr $ra                      # return
```

RECURSE:

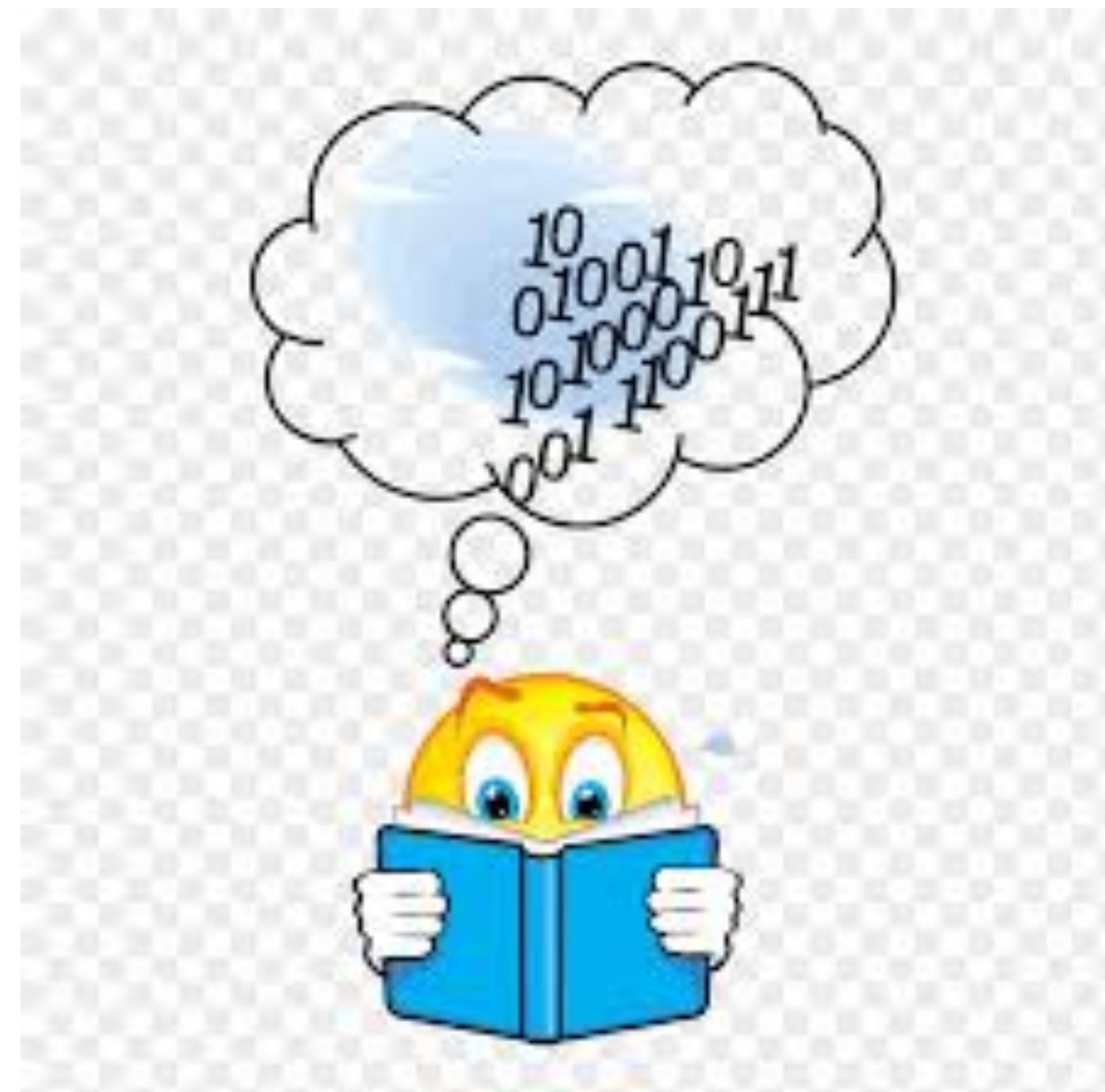
2

```
addi $a0, $a0, -1          # n - 1
jal fact                   # recursive call
```

4

```
lw $a0, 0($sp)             # restore n
mul $v0, $a0, $v0           # n * fact(n - 1)
lw $ra, 4($sp)              # restore return address
addi $sp, $sp, 8            # deallocate stack
jr $ra                      # return
```

Interpreting Instructions



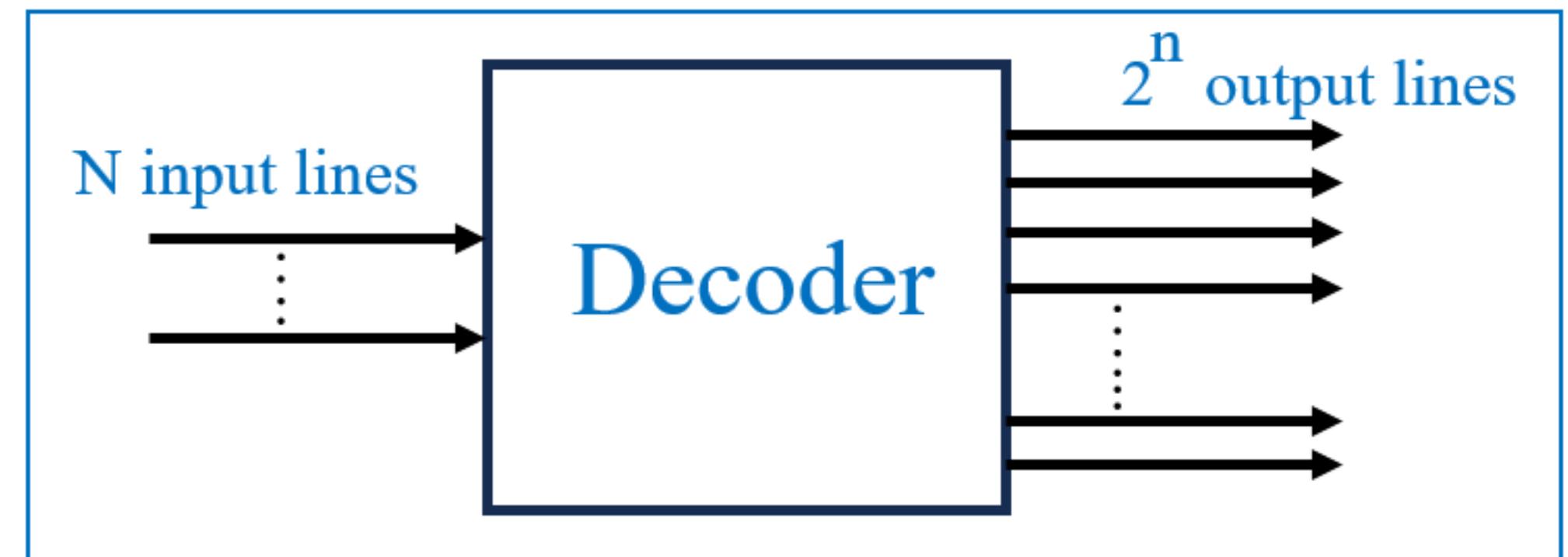
Why to Interpret?

- Everything is a string of bits...
- Instructions are 32 bit strings
- How do the hardware know what is `$lw` and what is `$addi`
 - Unique bit string for each of them
- But how does the machine interpret them?

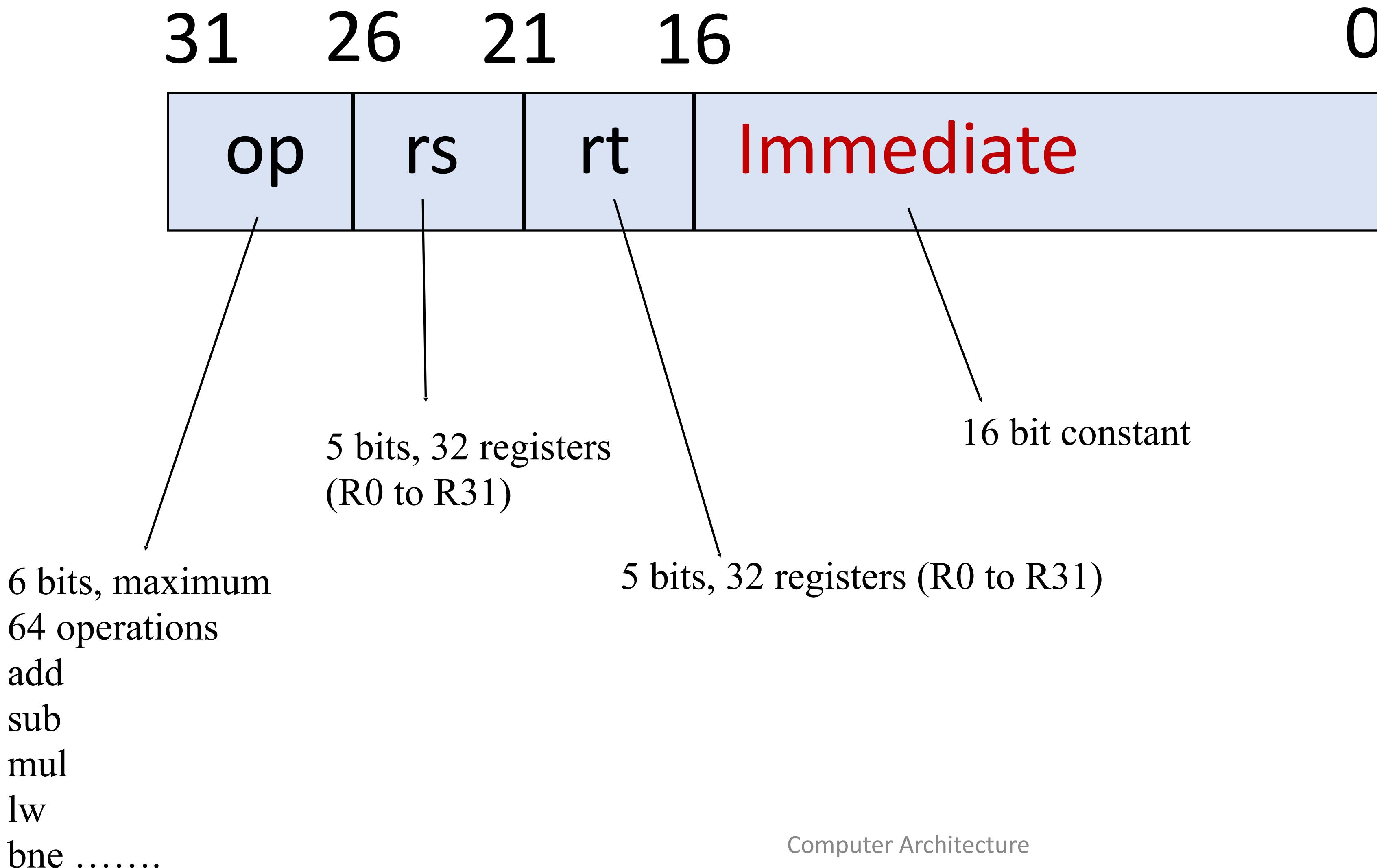


Decode it ...

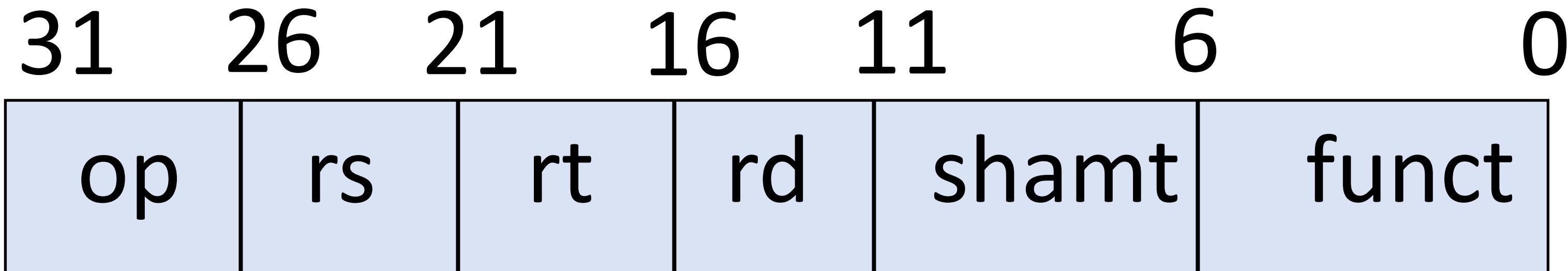
- Everything is a string of bits...
- Instructions are 32 bit strings
- How do the hardware know what is `$lw` and what is `$addi`
 - Unique bit string for each of them
- But how does the machine interpret them?



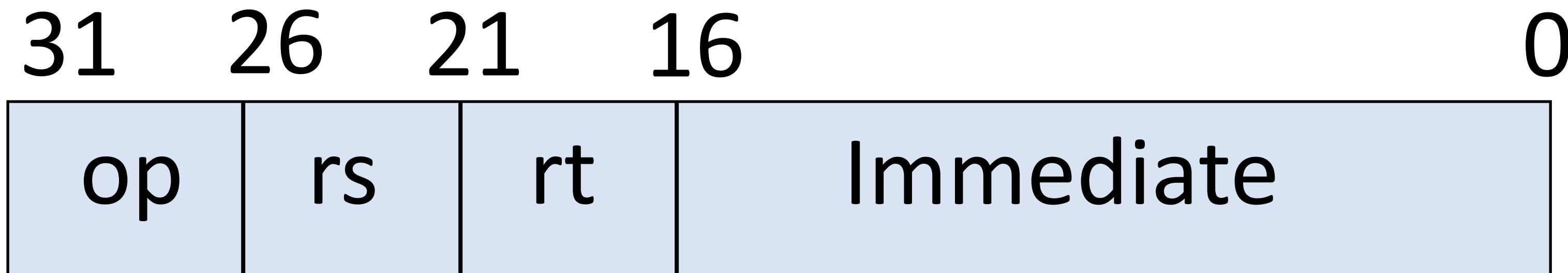
Instruction Decoding



Let's have a look



R-type



I-type



J-type

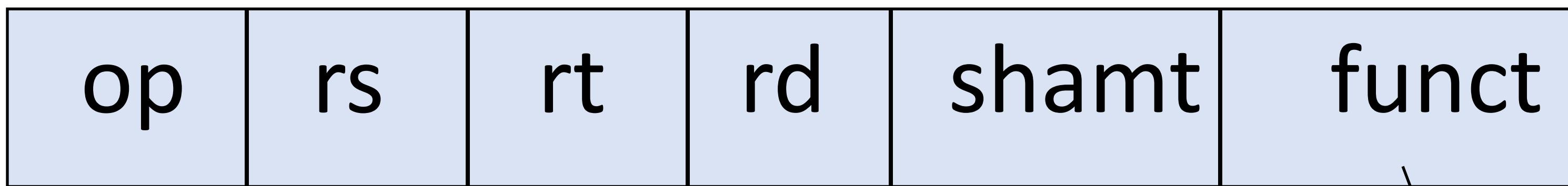
Good design demands good compromises

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32	n.a.
sub	R	0	reg	reg	reg	0	34	n.a.
addi	I	8	reg	reg	n.a.	n.a.	n.a.	constant
lw	I	35	reg	reg	n.a.	n.a.	n.a.	address
sw	I	43	reg	reg	n.a.	n.a.	n.a.	address



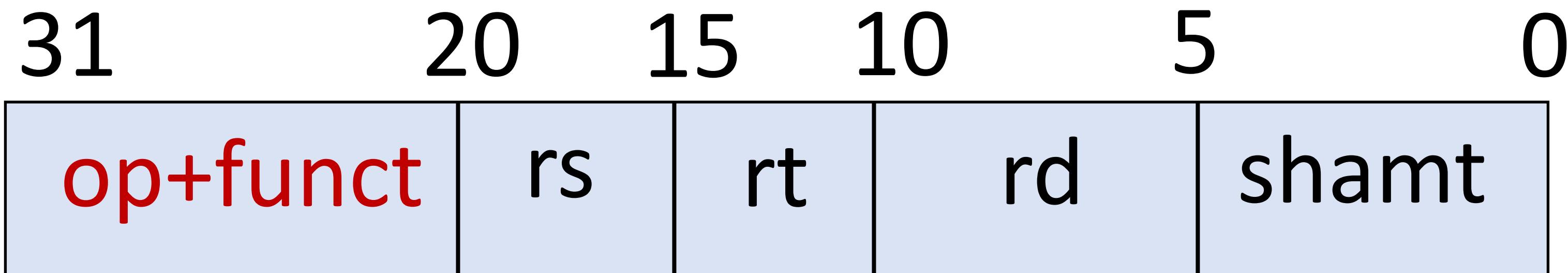
tells how to treat the last set of fields:
three fields or one field, still why funct ☹

MIPS encoding Anatomy

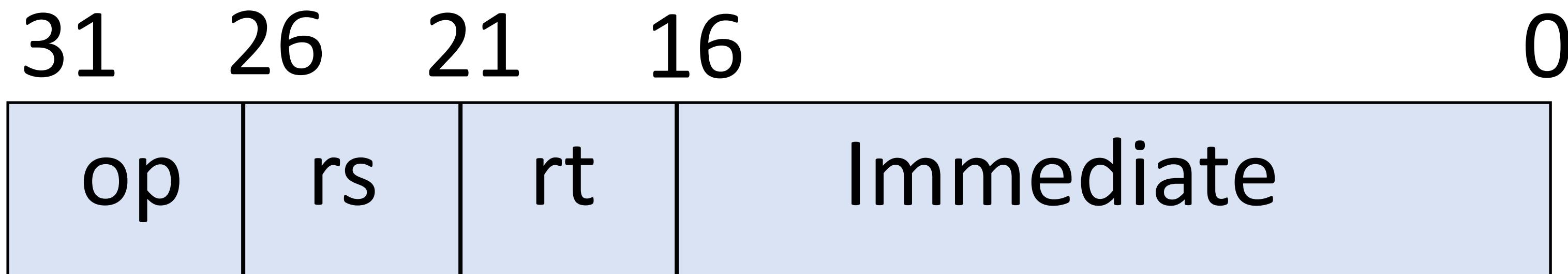


Why this field?
Wastage of space ☹

Why not?



R-type



I-type



J-type

Potential Causes: Simple is The Best

- Every instruction looks the same
- Same number of bits for the **op** for all the instruction
- All the fields starts from the same location
- Uniform treatment for all instructions make the decoder hardware simple:
 - Believe me, you can save some MUXs



Tips to Become a Good Engineering Student

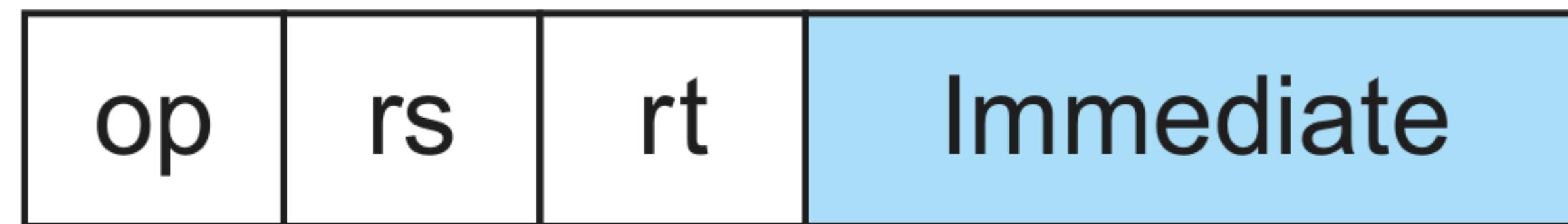
Addressing Modes

- Where is your data??



Addressing Modes

Immediate addressing



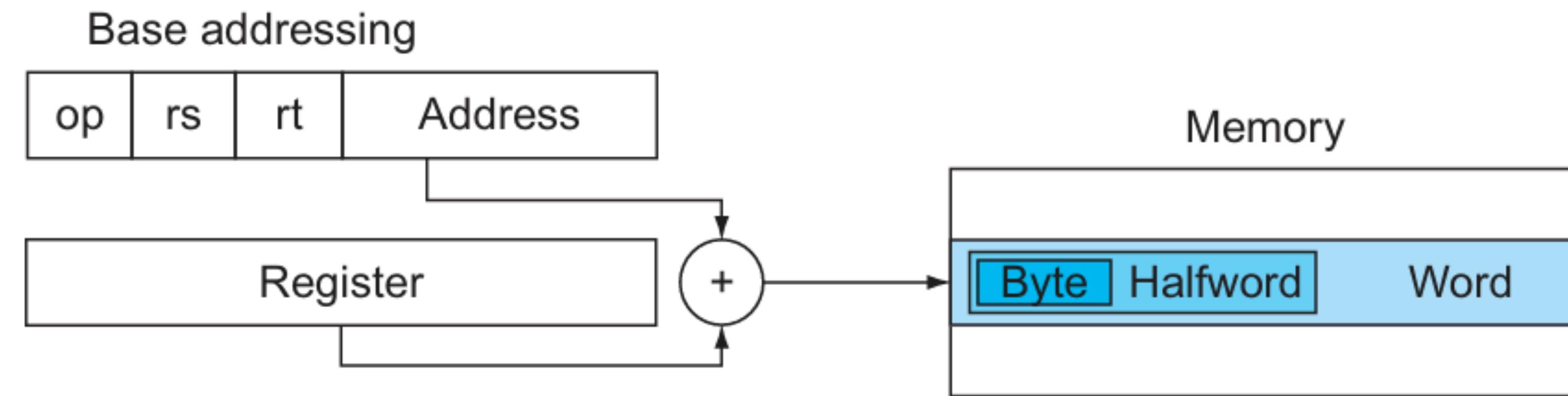
- Data is packed within instruction
- addi \$t0, \$t0, 3

Addressing Modes



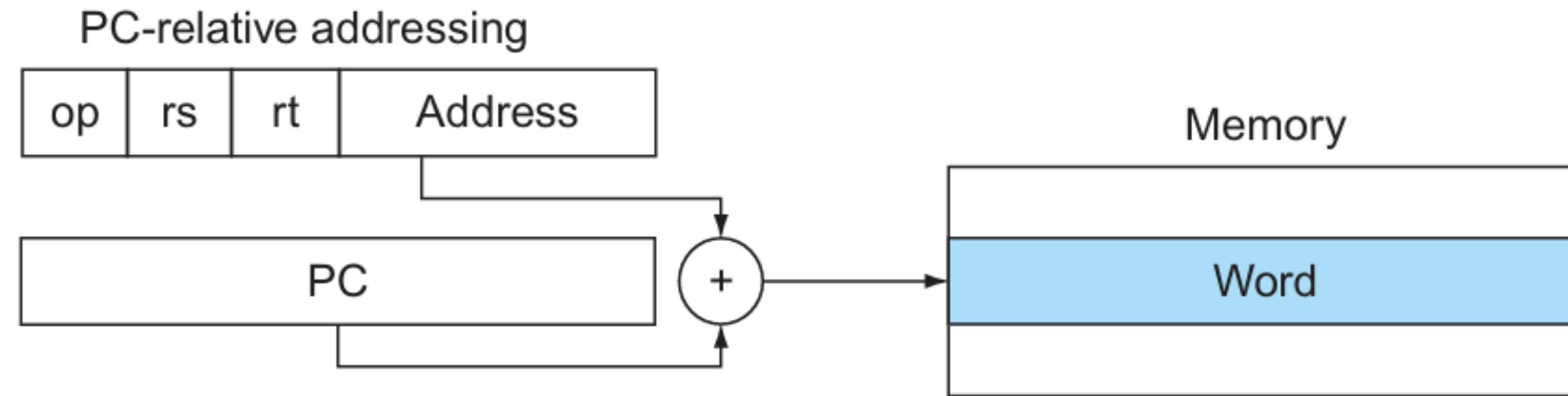
- Data is inside a register
- add \$t0, \$t1, \$s0

Addressing Modes



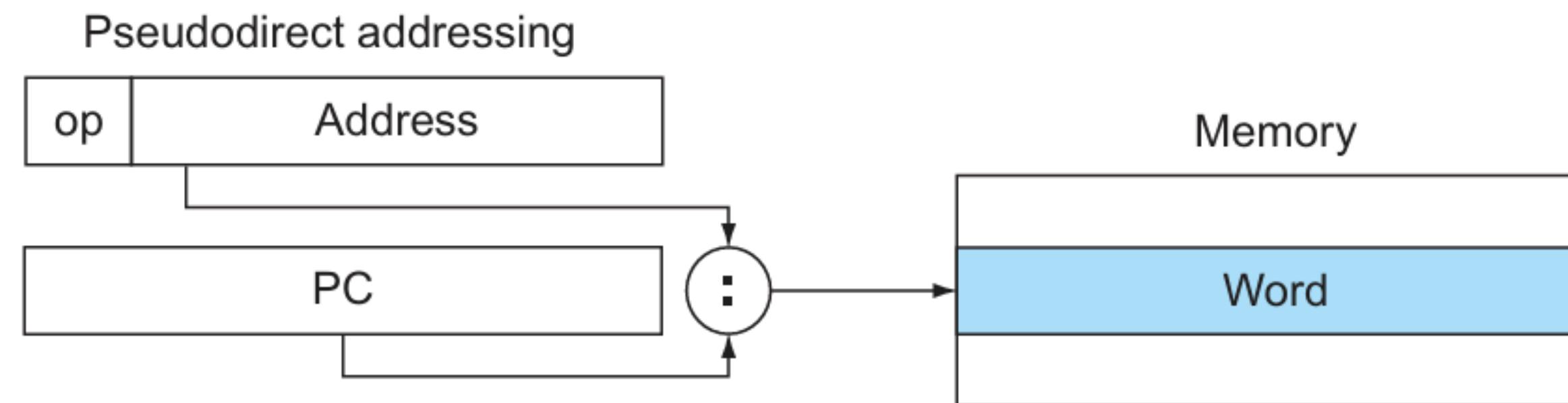
- Data is in memory
- Address is the sum of a register value + offset specified in the instruction.
- `lw $t0, 4($s0)`, `lw $t0, ($s0)`
- `lw/sw` have many versions for accessing byte halfword and word.
- Great for accessing arrays, structures, and with pointers

Addressing Modes



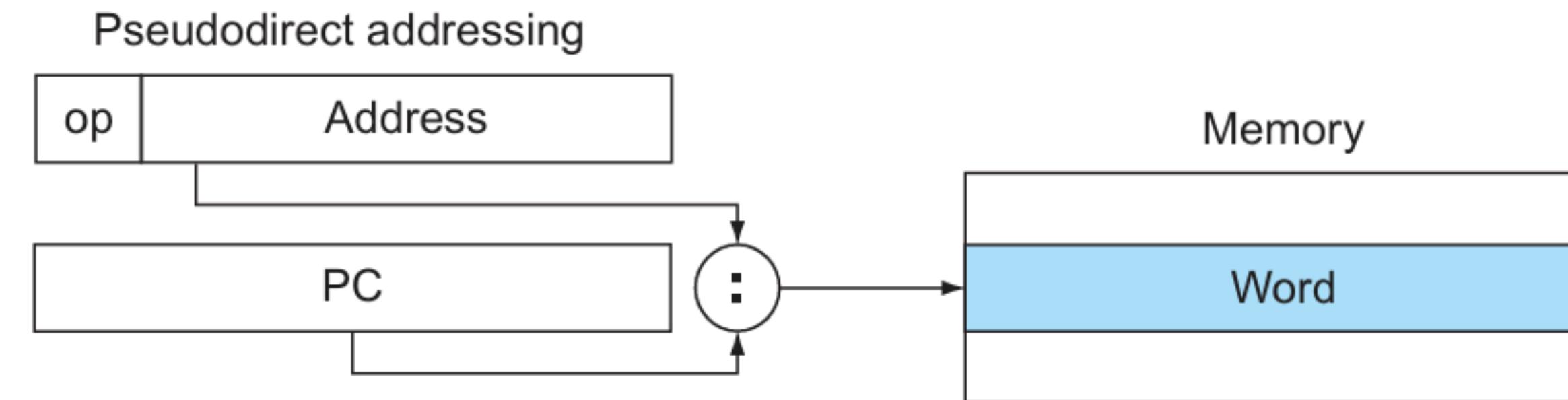
- Conditional branches
- The address field basically specifies an offset
- Adds a 16-bit address (sign extended to 26 bit and shifted left by 2 bits) to the PC
- `beq $t0, $zero, RECURSE`

Addressing Modes



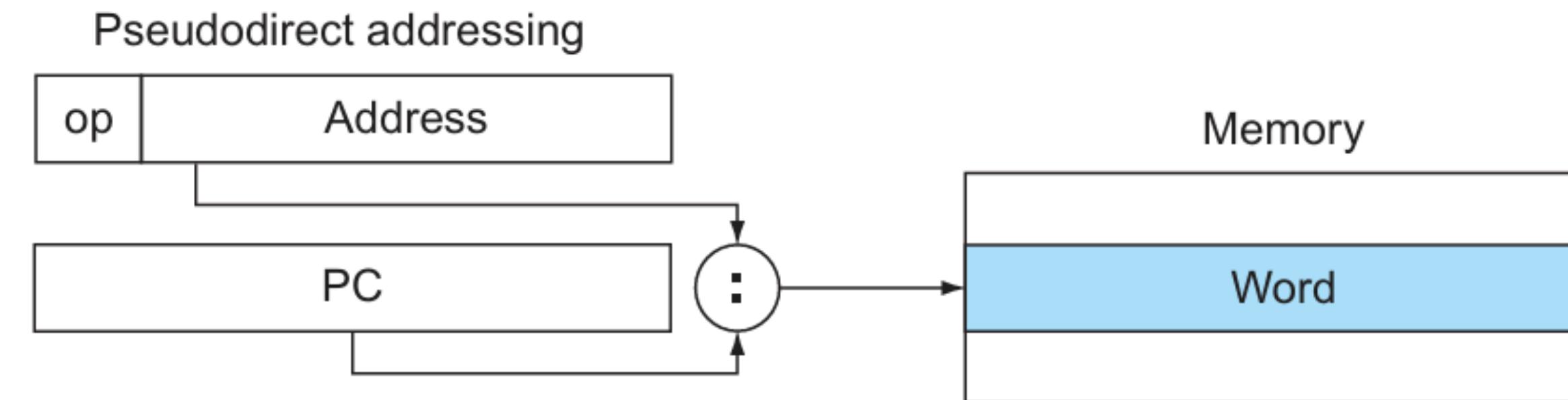
- Again for branching
- `jal` label
- Concatenates a 26-bit address shifted left 2 bits with the 4 upper bits of the PC
 - Why?

Addressing Modes



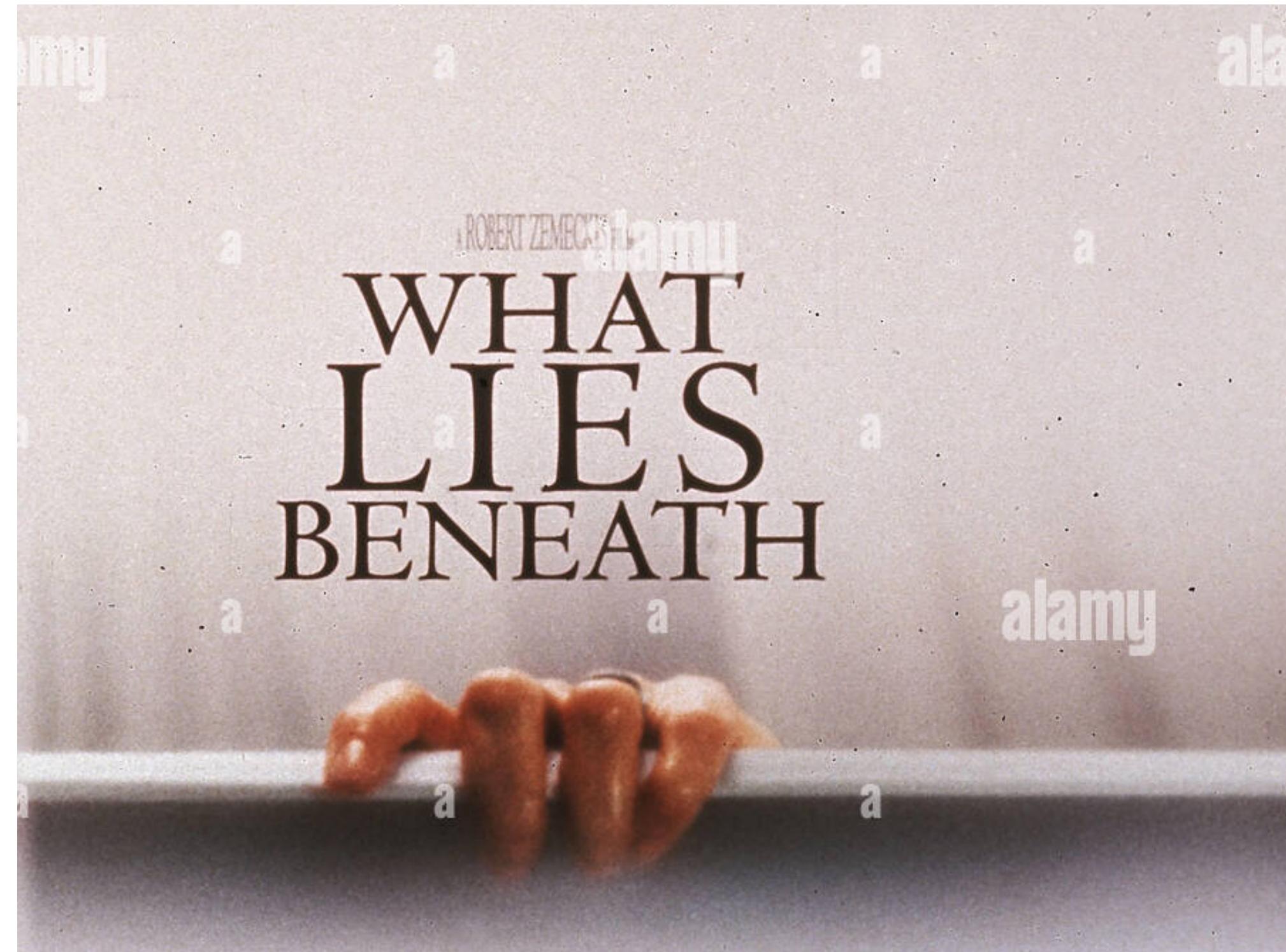
- Again for branching
- `jal label`
- Concatenates a 26-bit address shifted left 2 bits with the 4 upper bits of the PC
 - Why?
 - Remember word alignment of memory?????
 - 26 shifted by 2 bit is 28 bits with two zeros at the end — always divisible by 4
 - You can address 2^{28} bits = 256 MB
 - Add the upper 4 bits of PC — can access several 256 MB chunks..

Addressing Modes



- Again for branching
- `jal label`
- Concatenates a 26-bit address shifted left 2 bits with the 4 upper bits of the PC
 - Why?
 - Remember word alignment of memory?????
 - 26 shifted by 2 bit is 28 bits with two zeros at the end — always divisible by 4
 - You can address 2^{28} bits = 256 MB
 - Add the upper 4 bits of PC — can access several 256 MB chunks..

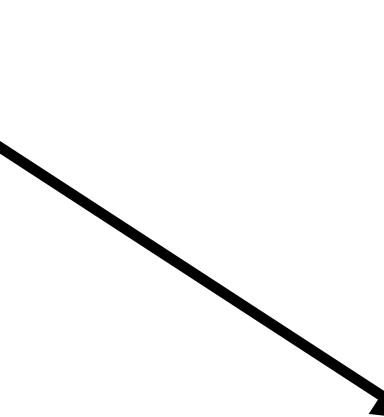
ISA Vs. Microarchitecture



ISA Vs. Microarchitecture

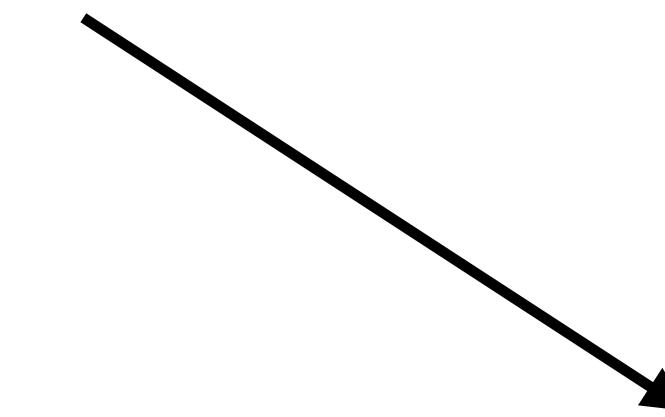
ISA

- What a programmer gets to see
- add, lw



Microarchitecture

- How the instructions are implemented
- How the add is implemented — ripple carry/carry-save



Hardware Design

- How to implement the ripple carry with gates
- Deals with gates, flip-flops

ISA Vs. Microarchitecture

ISA: Programmer's View

- Opcodes
- Registers
- Addressing modes
- Alignment
- Instruction types/formats
- Access control user/OS
- Address space

ISA satisfies the need of assembly level
programmers, compilers, assemblers

Microarchitecture

- Pipelining
- Caches
- Branch Predictors
- Prefetches
- Memory controllers

Microarchitecture is: how to bring speed..

Microarchitecture State

The information held in the processor at the end of an instruction to provide the processing context for the next instruction.

Processor is in state S

Instruction

Processor moves to state S'

ISA Vs. Microarchitecture

ISA: Programmer's View

- Opcodes
- Registers
- Addressing modes
- Alignment
- Instruction types/formats
- Access control user/OS
- Address space

ISA satisfies the need of assembly level
programmers, compilers, assemblers

Microarchitecture

- Pipelining
- Caches
- Branch Predictors
- Prefetches
- Memory controllers

Microarchitecture is: how to bring speed..

ISA Vs. Microarchitecture

ISA: Responsibility of the Designer

- Register — ISA
- Memory access instruction — ISA

Microarchitecture: Responsibility of the Designer

- Cycles to access the register — Microarch
- Cycles to access memory — Microarch

Computer Architecture

ISA+ Microarchitecture

Design Challenges

- Closer to software
 - complex instructions — e.g., instruction for GCD
 - Easy for programmers (perhaps..)

Design Challenges

- Closer to software
 - complex instructions — e.g., instruction for GCD
 - Easy for programmers (perhaps..)
 - Complex microarchitecture
 - Complex hardware design
- Closer to hardware
 - Simple instructions
 - More instructions in a program
 - Simpler and clean hardware.

Design Challenges

- Closer to software
 - complex instructions — e.g., instruction for GCD
 - Easy for programmers (perhaps..)
 - Complex microarchitecture
 - Complex hardware design



- Closer to hardware
 - Simple instructions
 - More instructions in a program
 - Simpler and clean microarchitecture and hardware.

The RISC CISC Debate

The RISC CISC Debate

- **RISC:** Reduced Instruction Set Computer — MIPS, RISC-V
 - **CISC:** Complex Instruction Set Computer — X86 [**actually not fully**]
-
- **Why CISC:**
 - Implementing commonly used instructions in hardware can lead to significant performance benefits.
 - For example, use of string instructions had significant performance benefits.
 - **Why RISC:**
 - The rarely used instructions can be eliminated to save chip space --- on chip cache and large number of registers can be provided.

The RISC CISC Debate

- Rich instruction set:
 - Some simple, some very complex
- Complex addressing modes:
 - **Orthogonal addressing** (Every possible addressing mode for every instruction).
- Many instructions take multiple cycles:
- Instructions are of variable sizes
- Microcode control

The RISC CISC Debate

• Direct (absolute)	ADD R1, (1001)
• Register indirect	SUB R2, (R1)
• Indexed	ADD R1, (R2 + R3)
• Scaled	SUB R2, 100(R2) [R3]
• Autoincrement	ADD R1, (R2) +
• Autodecrement	SUB R2, - (R1)
• Memory indirect	ADD R1, @ (R3)

And more ...

The RISC CISC Debate – X86

12 addressing modes:

- Register.
- Immediate.
- Direct.
- Base.
- Base + Displacement.
- Index + Displacement.
- Scaled Index + Displacement.
- Based Index.
- Based Scaled Index.
- Based Index + Displacement.
- Based Scaled Index + Displacement.
- Relative.

Operand sizes:

- Can be 8, 16, 32, 48, 64, or 80 bits long.
- Also supports string operations.

Instruction Encoding:

- The smallest instruction is one byte.
- The longest instruction is 12 bytes long.
- The first bytes generally contain the opcode, mode specifiers, and register fields.
- The remainder bytes are for address displacement and immediate data.

The RISC CISC Debate – X86

- One instruction could do the work of several instructions.
 - For example, a single instruction could load two numbers to be added, add them, and then store the result back to memory directly.
- Many versions of the same instructions were supported:
 - For example, one version would read two numbers from memory, and store the result in a register. Another version would read one number from memory and the other from a register and store the result to memory.

The RISC CISC Debate

- Compilers were hard to build especially for machines with registers
 - Make machine do more work than software
 - Have instructions load and store directly to memory (memory-to-memory operations)
- Magnetic core memory was used as main memory which was slow and expensive
 - Minimize assembly code size
 - **Complex Instruction Set Computers (CISC)**

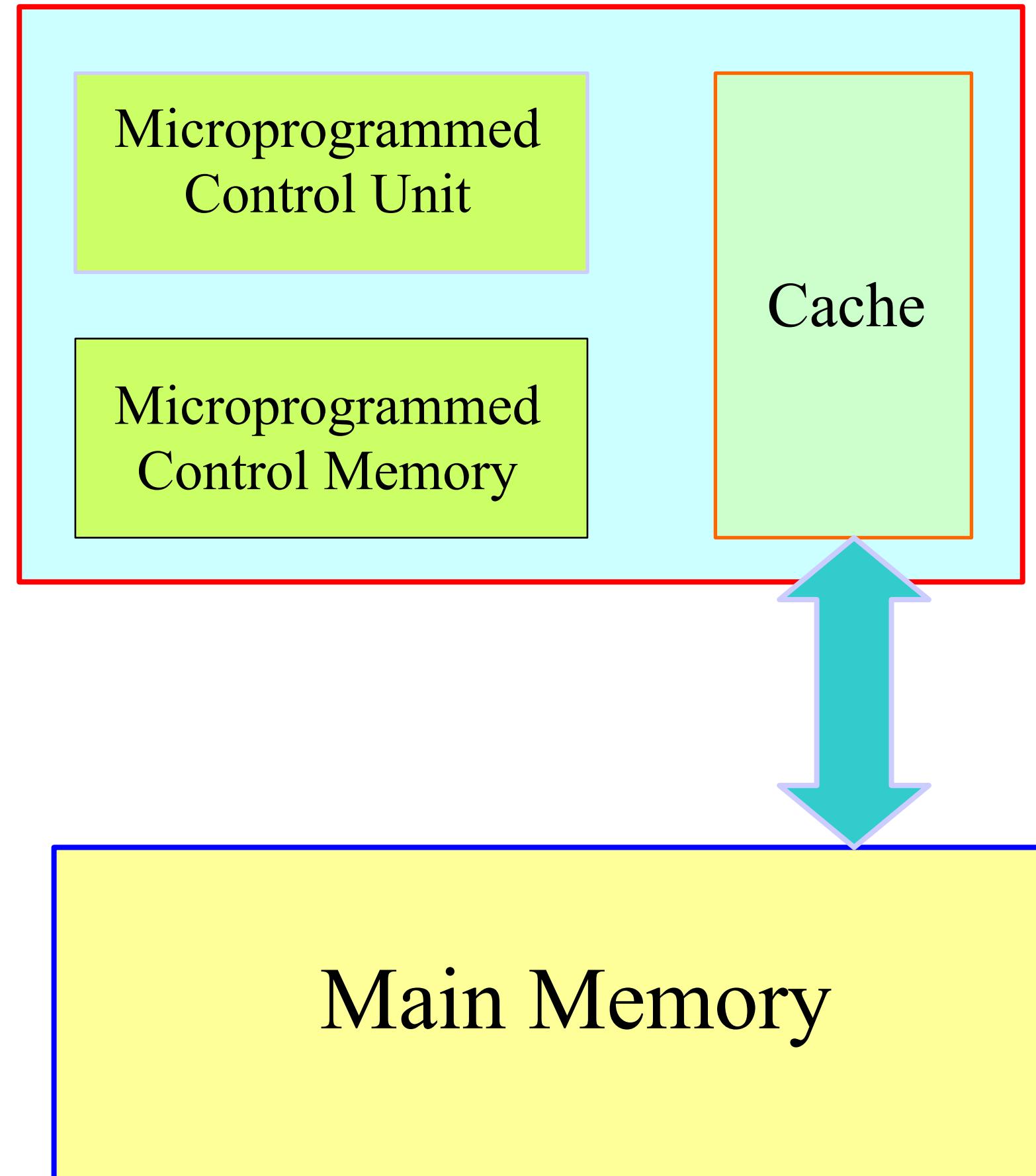
The RISC CISC Debate

- Compilers were improving:
 - Optimizing compilers could effectively use registers
- Caches:
 - Allowed main memory to be accessed at similar speeds to control memory
- Semiconductor memory was replacing magnetic core memory:
 - Reduced performance gap between control and main memory

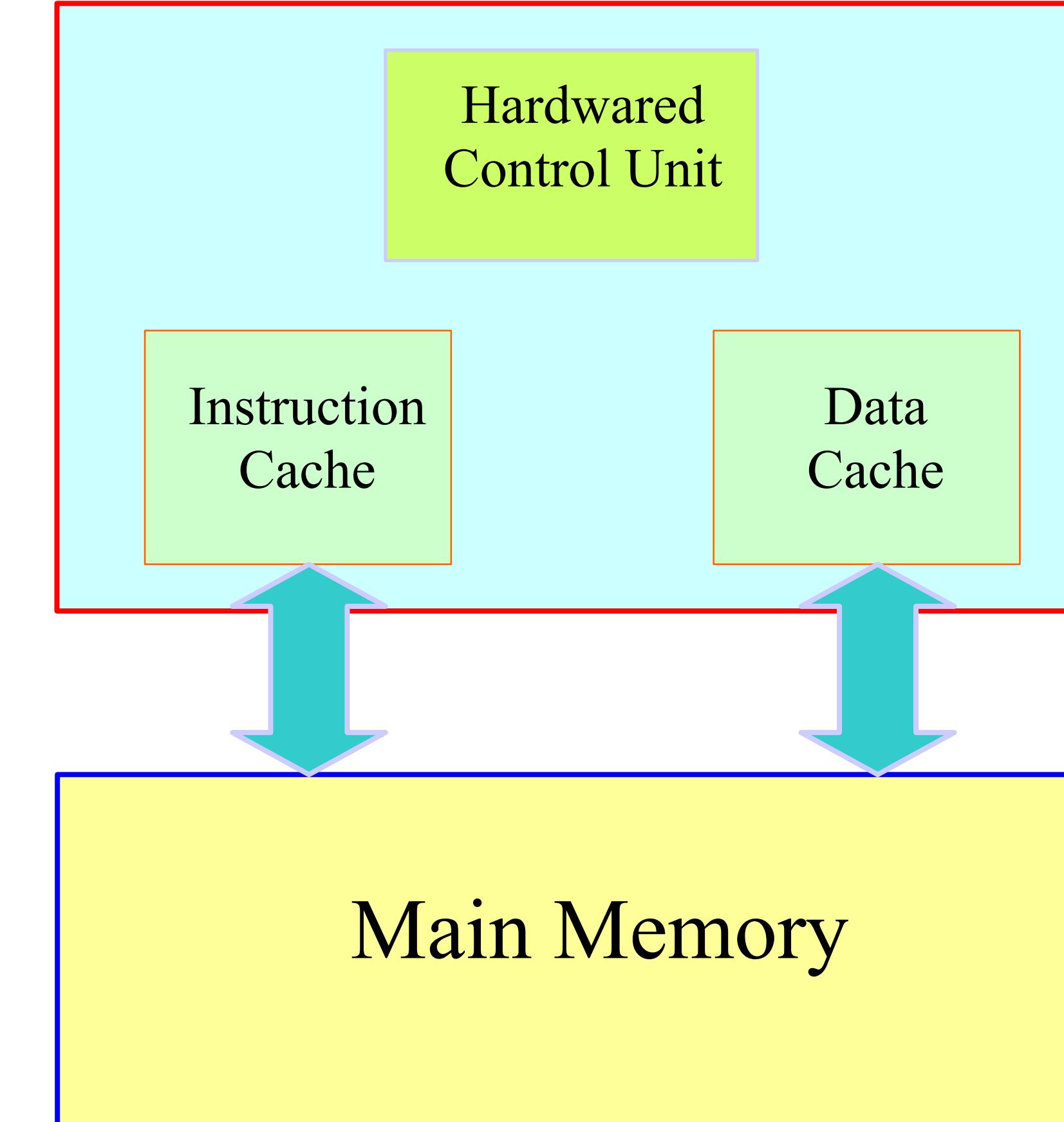
The RISC CISC Debate – RISC

- Small number of instructions
- Small number of addressing modes
- **Large number of registers (>32)**
- Instructions execute in one or two clock cycles
- Uniformed length instructions and fixed instruction format.
- **Register-Register Architecture:**
 - Separate memory instructions (**load/store**)
 - Separate instruction/data cache
- **Hardwired control**
- **Pipelining**

The RISC CISC Debate



(a) CISC Organization



(b) RISC Organization

Implementation Issues

- A CISC processor typically devotes about half of its area to the control unit.
 - A RISC processor typically uses only about 10% of the area for the control unit, devoting to registers instead.
- Design and implementation time.
 - The simple control unit and circuitry of RISC result in faster design cycles.

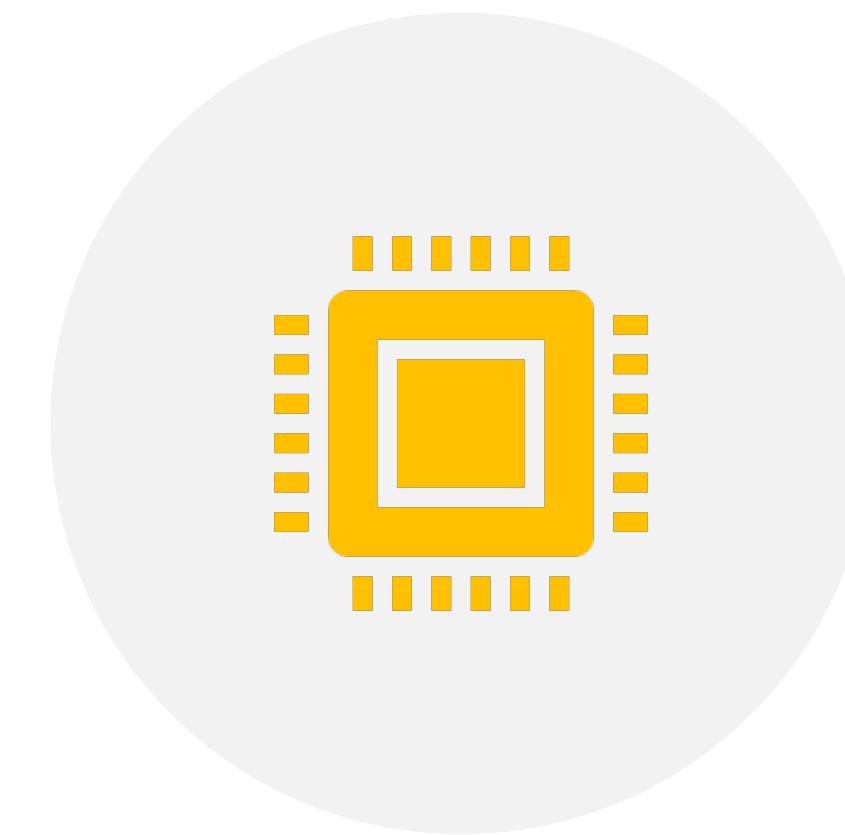
Mystery



INTEL CONVERTS CISC ONES INTO RISC
ONES, AND GENERATE
MICROOPERATIONS.



INTELLIGENT CISC-RISC DECODER



CONSUMES AROUND 2% OF THE CHIP
AREA. GOOD OR BAD?

How Easy?

A billion-dollar idea ☺

- i) requires changes to microarchitecture.
- ii) requires changes to ISA.
- iii) both (i) and (ii)

Think about the trade-offs ☺

Does it affect the system stack?

The Other ISAs

World of ISAs



x86: Intel, AMD: Laptops, Desktops,
Servers



ARM: Arm, Qualcomm, Apple,
Samsung: Mobiles



btw ARM: Advanced RISC Machines ☺

RISC-V: Open-source ISA, what does it mean?

RISC-V, opensource

– What is the license model?

The RISC-V ISA is free and open with a permissive license for use by anyone in all types of implementations. Designers are free to develop proprietary or open source implementations for commercial or other exploitations as they see fit. RISC-V International encourages all implementations that are compliant to the specifications.

Note that the use of the RISC-V trademark requires a license which is granted to members of RISC-V International for use with compliant implementations. The RISC-V specification is based around a structure which allows flexibility with modular extensions and additional custom instructions/extensions. If an implementation was based on the RISC-V specification but includes modifications beyond this framework, then it cannot be referenced as RISC-V.

– Does that mean free for industry to use and play with, but then we pay if we produce a product using this ISA?

The RISC-V ISA is free for product use too. Those who want to use the RISC-V logo should join RISC-V International (see question No. 1).

– If our company builds a RISC-V implementation, is it required to release its source code for the RISC-V core?

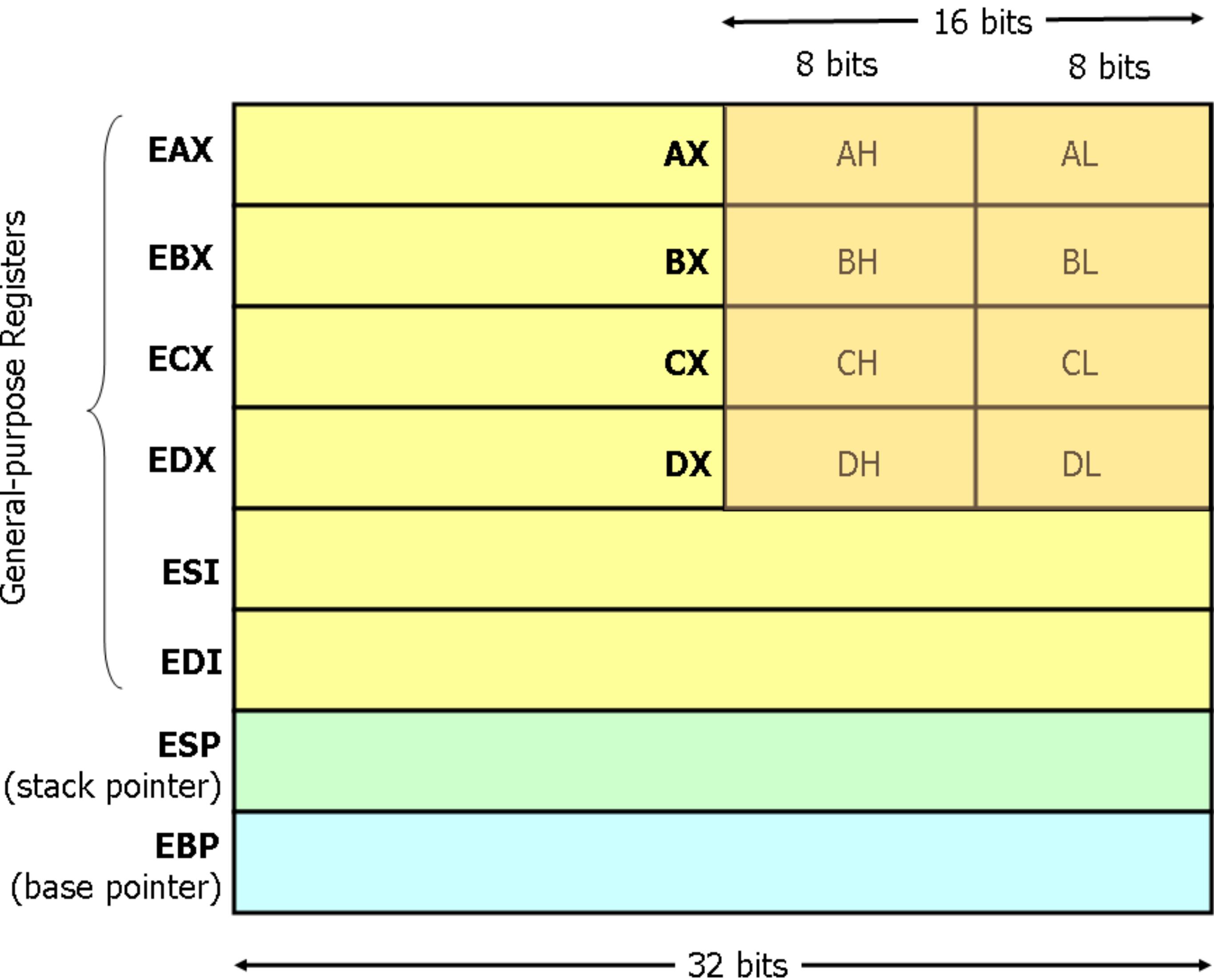
No, the source code can be completely closed.

x86

Registers:

80386

For 64-bits, `rax`, `rbx`



Subtle Differences

- x86 arithmetic/logic instructions: one operand should act as both source and destination

add \$s0, \$s1 // Add \$s0 and \$s1 and put it in \$s0 ☺

- One of the operands can be in memory:

wow (programmer) or oh no (instruction size ☹) !!

add \$s0, Mem[\$s1] ☺

- No more fixed-length instructions ☹ can be 4/8/X bytes

Why?

Fixed Width or Variable Width

Variable: No fixed size

6 bytes for add, 2 bytes for load

Smaller code footprint (compact)

Fixed:

4 bytes for all, Larger code footprint, **simple decoding**

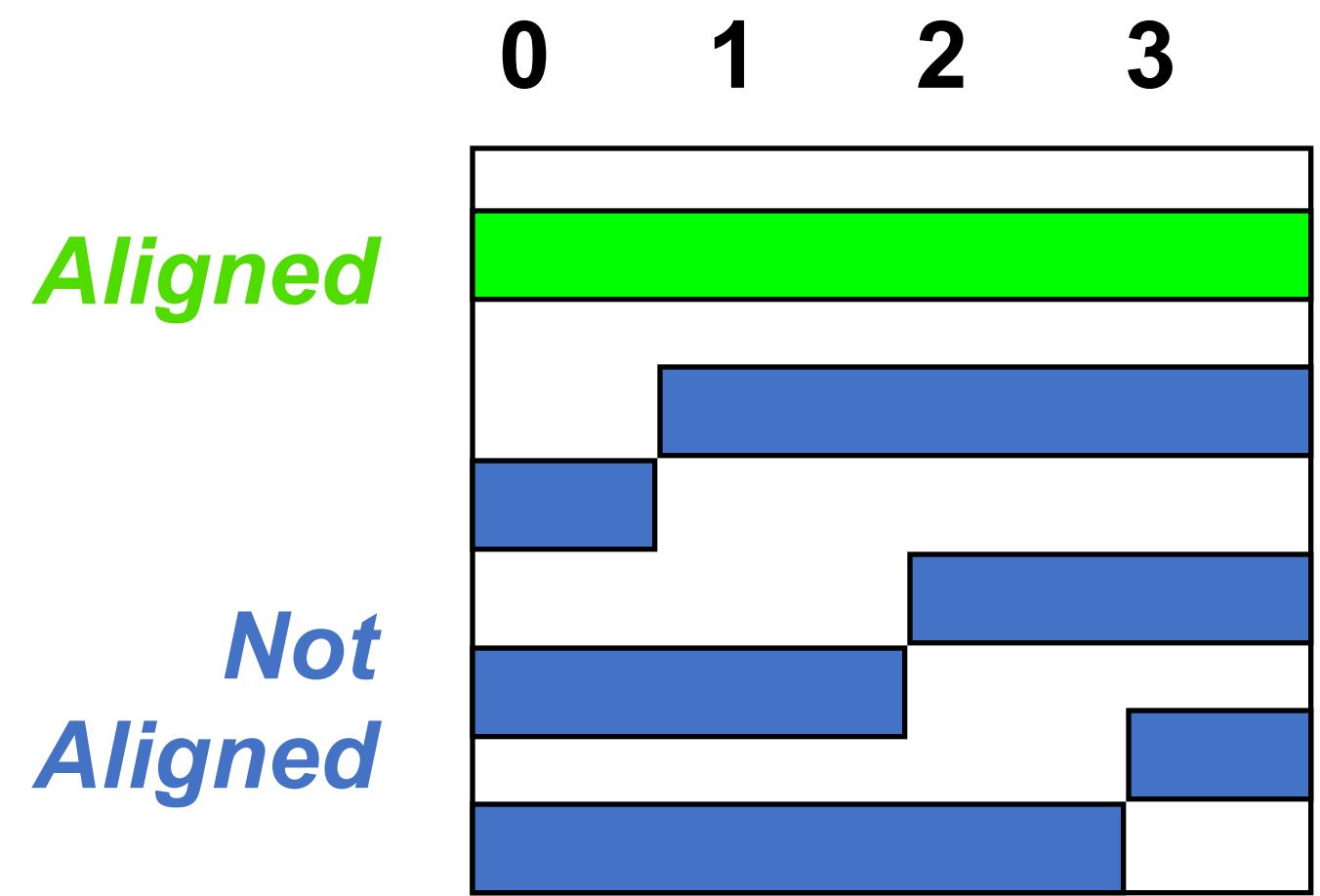
Fallacies

CISC instructions
provide higher
performance

Assembly language
codes provide higher
performance



Instruction Alignment: Why we need it?



Aligned:

x-byte access starting from an address y: $y \% x$ must be zero.

MIPS vs X86

MIPS does not allow **unaligned** accesses

x86 **does not enforce** alignment ☺

Whose job is to generate aligned/unaligned accesses?

MIPS vs X86

MIPS does not allow **unaligned** accesses

x86 **does not enforce** alignment ☺

Whose job is to generate aligned/unaligned accesses?

Compiler

Let's go a bit deeper

Object of size s bytes at byte add. A is aligned if $A \bmod s = 0$

Alignment for faster transfer of data ?

Why fast ??

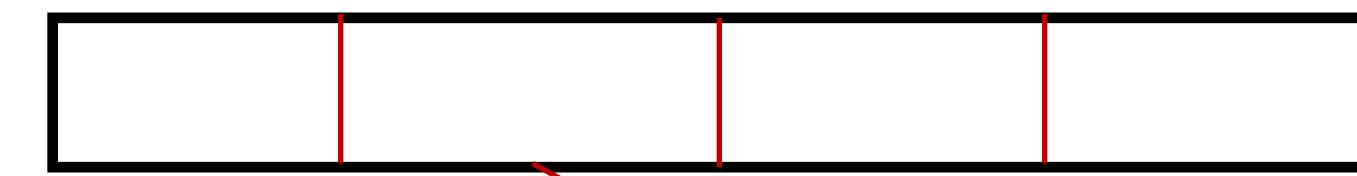
Think about memory (caches if you know).

Memory operations and alignment network

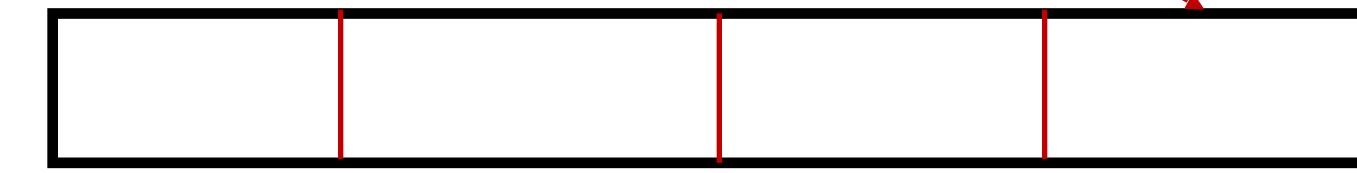
LOADs and STOREs need an alignment network that makes sure data loaded/written are aligned.

lb R1, 1(\$s3)

4-byte chunk



Register R1



For the Curious ones

<https://lemire.me/blog/2012/05/31/data-alignment-for-speed-myth-or-reality/>