Types of instructions

- **Data operations**
  - Arithmetic (add, subtract, …)
  - Logical (and, or, not, xor, …)
- **Data transfer**
  - Load (memory → register)
  - Store (register → memory)
- **Sequencing**
  - Branch (conditional, e.g., <, >, ==)
  - Jump (unconditional, e.g., goto)

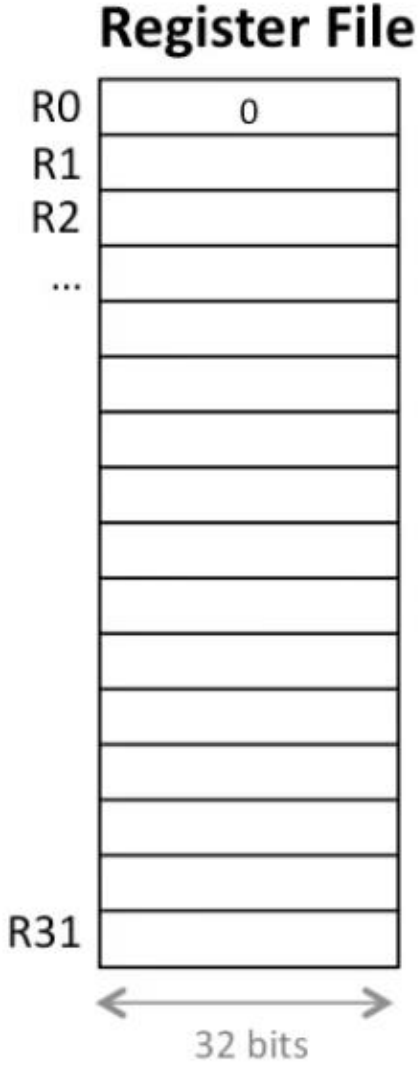
```
load r0 mem[7]
loop:
  r1 = r0 - 2
  j_zero r1 done
  r0 = r0 + 1
  jump loop
done:
```

Function	Instruction	Effect
add	add R1, R2, R3	R1 = R2 + R3
sub	sub R1, R2, R3	R1 = R2 – R3
add immediate	addi R1, R2, 145	R1 = R2 + 145
multiply	mult R2, R3	hi, lo = R1 * R2
divide	div R2, R3	low = R2/R3, hi = remainder
and	and R1, R2, R3	R1 = R2 & R3
or	or R1, R2, R3	R1 = R2 | R3
and immediate	andi R1, R2, 145	R1 = R2 & 143
or immediate	ori R1, R2, 145	R1 = R2 | 145
shift left logical	sll R1, R2, 7	R1 = R2 << 7
shift right logical	srl R1, R2, 7	R1 = R2 >> 7
load word	lw R1, 145(R2)	R1 = memory[R2 + 145]
store word	sw R1, 145(R2)	memory[R2 + 145] = R1
load upper immediate	lui R1, 145	R1 = 145 << 16
branch on equal	beq R1, R2, 145	if (R1 == R2) go to PC + 4 + 145*4
branch on not equal	bne R1, R2, 145	if (R1 != R2) go to PC + 4 + 145*4
set on less than	slt R1, R2, R3	if (R2 < R3) R1 = 1, else R1 = 0
set less than immediate	slti R1, R2, 145	if (R2 < 145) R1 = 1, else R1 = 0
jump	j 145	go to 145
jump register	jr R31	go to R31
jump and link	jal 145	R31 = PC + 4; go to 145

(Complete table is printed in the book for reference.)

Registers in MIPS

- **32 General Purpose Registers**
  - R0…R31 or $0…$31
  - Each is 32 bits wide
  - Values for instructions must come from registers
- **Some are special**
  - **R0** is always **zero**
  - **R29/R31** are used for function calls
- **A few special registers**
  - **PC (Program Counter)**: current instruction
  - Hi & Lo results of multiplication
  - Floating point registers
  - Control registers (for errors and status)



Q: What does add r3, r1, r0 do?

- Put r0+r3 → r1
- Put r1+r3 → r0
- Move r1 → r3

A: Move r1 → r3

add r3, r1, r0 has two sources (r1 and r0) and one destination (r3).
So r3 ← r1 + r0

But r0 is always 0.
This is the same as r3 ← r1+0 or Move r1 → r3.

Question: register order

Q: How do I store the results of a+b into c?
R1 = a; R2 = b; R3 = c

- add R1, R2, R0
- add R3, R2, R1
- add R2, R1, R3

op dest, src1, src2
dest ← src1 *op* src2

add R3, R1, R2

A: add R3, R2, R1
We want the destination to be c (R3)
We want the sources to be a and b (R2, R1)

(Note that the order of the sources does not matter for addition because it is commutative.)

The first register is the destination (R3) and the other two registers re the sources (R2, R1).

So add R3, R2, R1 gives us R3 ← R2+R1, which is c=b+a, which is the same as c=a+b.

Question: complex functions

Q: How many other registers are required to calculate the complex function
R5 = (R5+R6) + R7?

- 0
- 1
- 2
- 3

A: 0

We can first use R5 to hold the results of (R5+R6) then add R7.

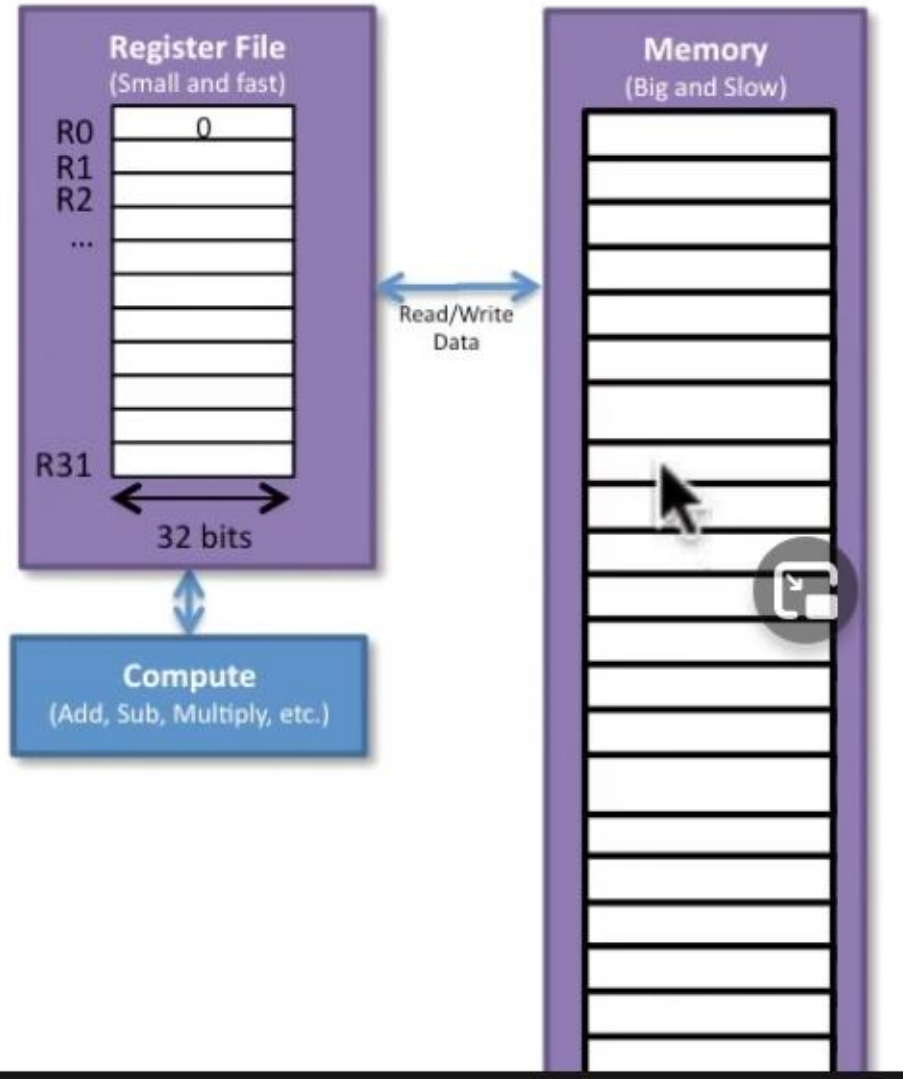
```
add R5, R5, R6   ; does R5 ← R5+R6
add R5, R5, R7   ; does R5 ← (R5+R6)+R7
```

Memory vs. registers

- MIPS is a **Load/Store Register File** machine
  - Instructions **compute only on data in the Register File**
  - Example:
    - **add R3, R2, R1**
    - all data needs to be in the Register File
  - But we only have 32 registers in the Register File
    - Clearly not enough for a big program

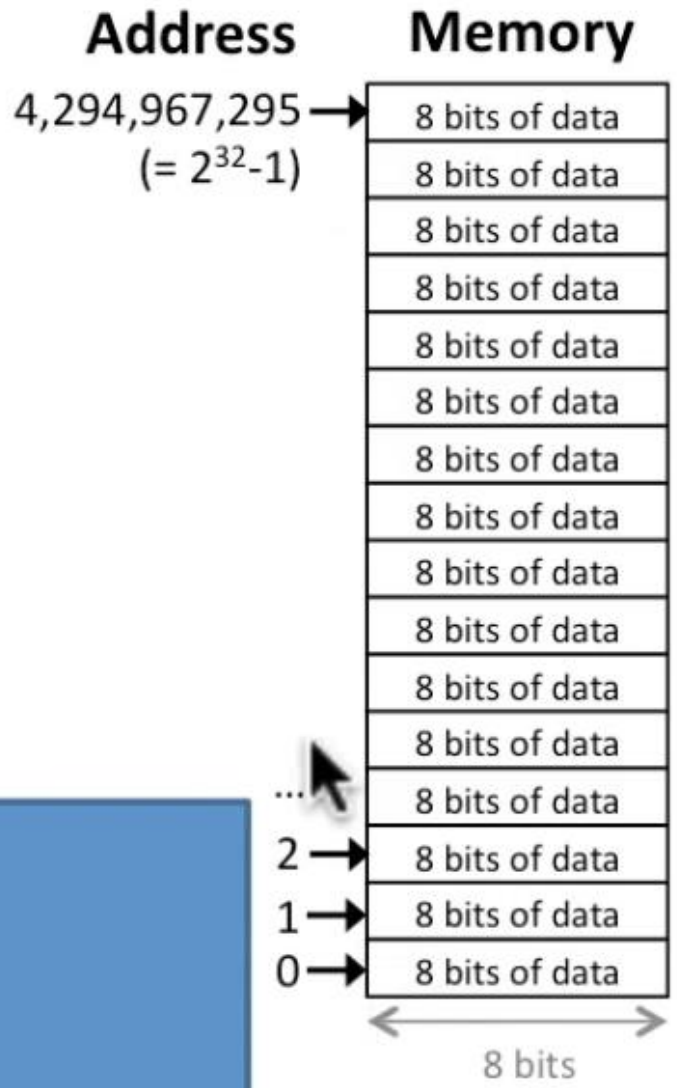
- Most data is stored in **memory** (large, but slow)
- Need to **transfer the data to the Register File** to use it
  - **Load**: load data from memory to the Register File (lw instruction)
  - **Store**: store data to the memory from the Register File (sw instruction)

- Hence, MIPS is a **Load/Store Register File** machine



Memory organization

- Memory is a large 1-dimension array
- Each location is **one byte** (8 bits)
- A **memory address** indexes into the array
- For a 32-bit computer, there are 2^{32} memory locations (4GB)
- For a 64-bit computer, there are 2^{64} memory locations (16EB)
  - 64-bit x86 machines tend to be limited to ~48-bits of address space, or 4PB.



Q: Why is the largest address for a 32-bit machine 2^{32}-1?

- We start counting at 0
- There's an error in the slide
- 2^{32} is reserved for errors

A: We start counting at 0

0000 0000 0000 0000 → 0

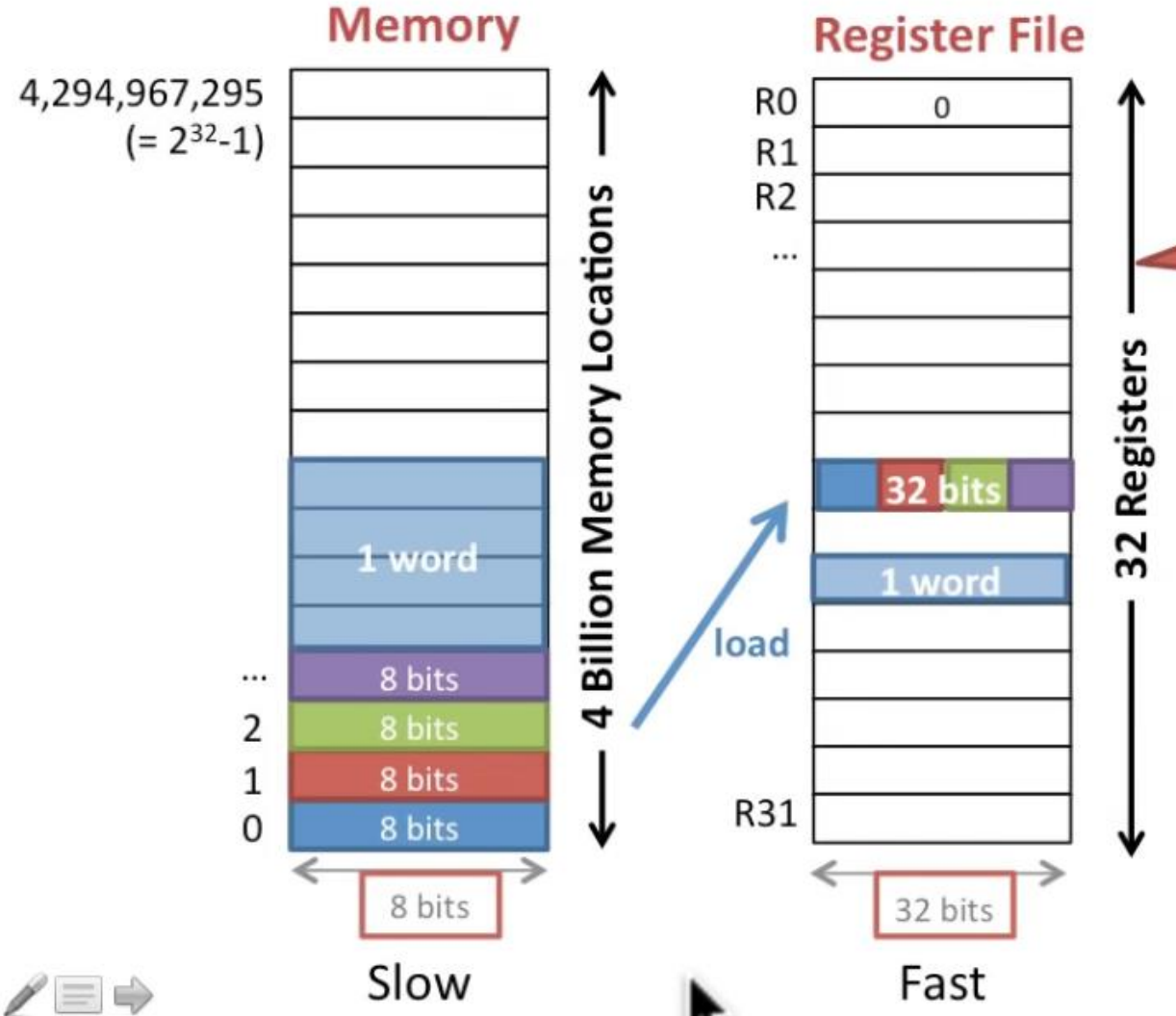
0000 0000 0000 0001 → 1

...

1111 1111 1111 1111 → 2^{32}-1

1 0000 0000 0000 0000 = 2^{32}, but needs 33 bits

Memory and register file



Q: How many memory locations do we need to fill a register in the Register File?

- 1
- 2
- 4

A: 4

Each entry in the register file is 32-bits. So we need 4 8-bit locations from the memory to make 32-bits for the register file.

This is inconvenient.
In MIPS we mostly deal with data in 4-byte chunks (32-bits) and we call this a **word**.

nd memory: MIPS Memory Organization

15

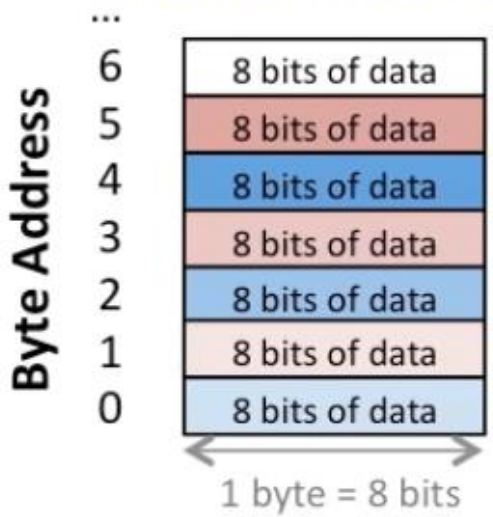
803 10 SHARE SAVE

MIPS Instruction Set Architecture (1 of 2)

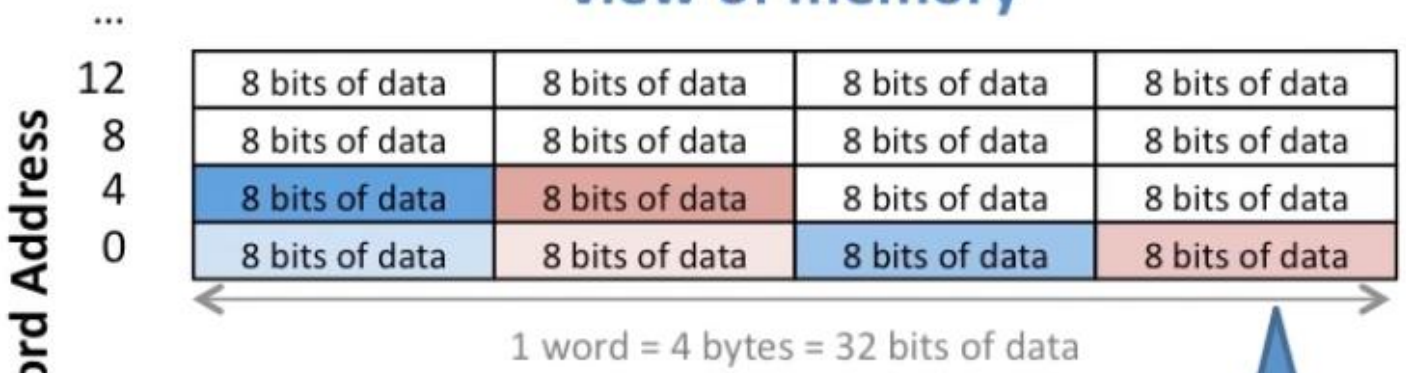
David B - 3 / 9

Viewing memory as bytes or words

Byte-addressable view of memory



Word-aligned view of memory



Registers hold 32 bits of data (1 word
Addresses are 32 bits of data (1 word

Loading 1 word now fills a whole register!

- Most data in MIPS is handled in **words** not bytes
  - A **word** is 32 bits or 4 bytes
  - A **word** is the size of a register

Q: What are the last 2 bits of a word address?

- Depends on the address
- Always 11
- Always 00

A: Always 00

Words are 4-bytes long, so every word address is an even multiple of 4 bytes. (E.g., 0, 4, 8, 12, …) In binary, this is the same as having the last two bits be zero. (0=00**00**, 4=01**00**, 8=10**00**, 12= 11**00**, etc.)

Access alignment

- **Aligned addresses** fall on 4 byte (word) boundaries (e.g., 0, 4, 8, 12…)
- **Unaligned addresses** do not (e.g., 1, 3, 7, 63)

- Some machines support unaligned accesses (not MIPS)
  - Hardware can convert to multiple aligned accesses (**complex**)
  - Hardware can detect it and have software fix it (**slow**)

- But there's typically a performance penalty
  - 2 memory accesses plus merging to get unaligned data
  - Intel provided high performance support in 2010 (Nehalem)

Example of aligned vs. unaligned access

Load Word at addr=9
Not Aligned

Load Word at addr=0
Aligned

…				
12	8 bits of data	8 bits of data	8 bits of data	8 bits of data
8	8 bits of data	8 bits of data	8 bits of data	8 bits of data
4	8 bits of data	8 bits of data	8 bits of data	8 bits of data
0	8 bits of data	8 bits of data	8 bits of data	8 bits of data

Remember, we load 32 bits at a time because the register file is 32-bits wide for each entry.

Question: memory and register files

Q: Is it reasonable that the Memory is much slower than the Register File?

- No, each memory location is only 8 bits wide and each register is 32 bits wide, so the registers should be slower since they are bigger.
- Yes, there are 4 billion memory locations and only 32 register locations so the memory should be slower because it is bigger.
- Unfair question: You haven't told me how the memory and register files are built, so how can I possibly know if this is reasonable?

A: Yes

The memory is 35 million times larger than the register file, so it will clearly be slower. (4 billion ÷ 32 ÷ 4 bytes per register = 35 million.)

While you don't know how these are built (yet) it is very reasonable to assume that we can build something that stores 35 million times less data much faster!

It turns out registers are about 200x faster.

Processor execution model

- Processor promises that the instruction execution will ***appear to be sequential and atomic***
  - **Sequential**: execute instructions in-order
  - **Atomic**: execute each instruction all at once

Why do we care?
If a program is not sequential and atomic we can't figure out what it should do. Impossible to debug and program.

- **Sequential**
  - Program says: R2 = R1 + R2 then R3 = R1 + R2
  - Processor may not do: R3 = R1 + R2 then R2 = R1 + R2 ← wrong result
- **Atomic**
  - Program says: R2 = R1 + R2 then R3 = R1 + R2
  - Processor has to **finish** R2= R1 + R2 **before starting** R3 = R1 + R2

- **Processors don't do either of these things** (too slow)
  - But it's important that they make it look like they do
  - We'll talk a lot about this when we get to pipelines

Question: non-sequential execution

Q: How many possible ways could the following program be executed by the processor if it didn't promise sequential execution?

```
add  r2, r3, r4
addi r3, r2, 12
sub  r4, r3, r2
```

- 1. You can't change the order without breaking the program.
- Unlimited. If it's not sequential it could be in any order.
- 6. There are only 6 ways to order them.

A: 6

If the processor does not promise sequential execution then it can execute any of the instructions first (3 choices) and any of the remaining two after (2 choices) before getting to the last one. So 3*2*1=6.

Note that it would be pretty much impossible to debug a program if you don't know which instructions will be executed first!

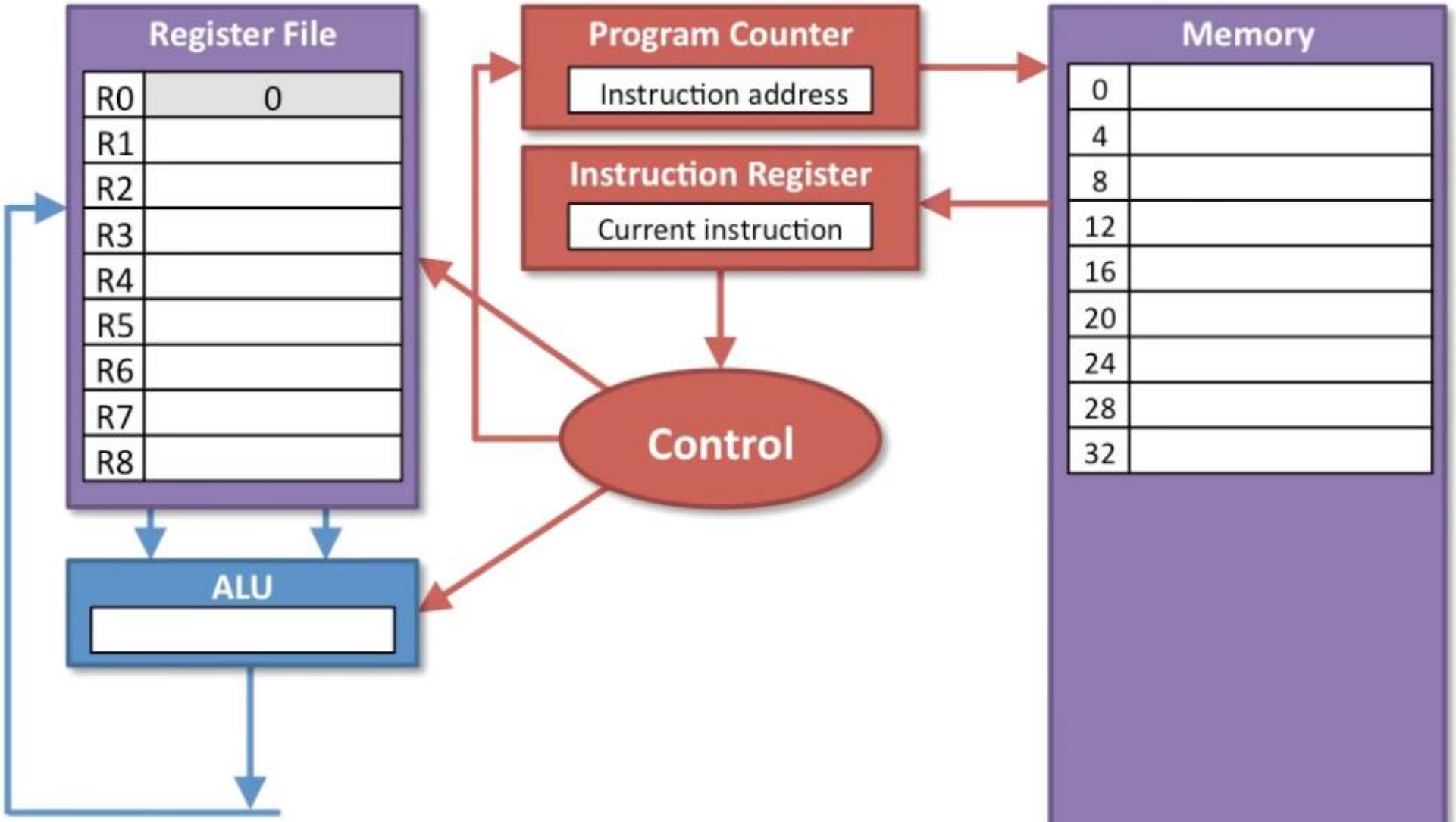
Stored program computers

- **Program and data are stored in memory**
  - Instructions have to be *fetched (loaded)* from memory for execution
  - Data has to be *fetched (loaded)* from memory for computation

- **No difference between data and instruction memory!**
  - This is where a lot of virus attacks come from: writing over data with instructions then executing them
  - Buffer overflows

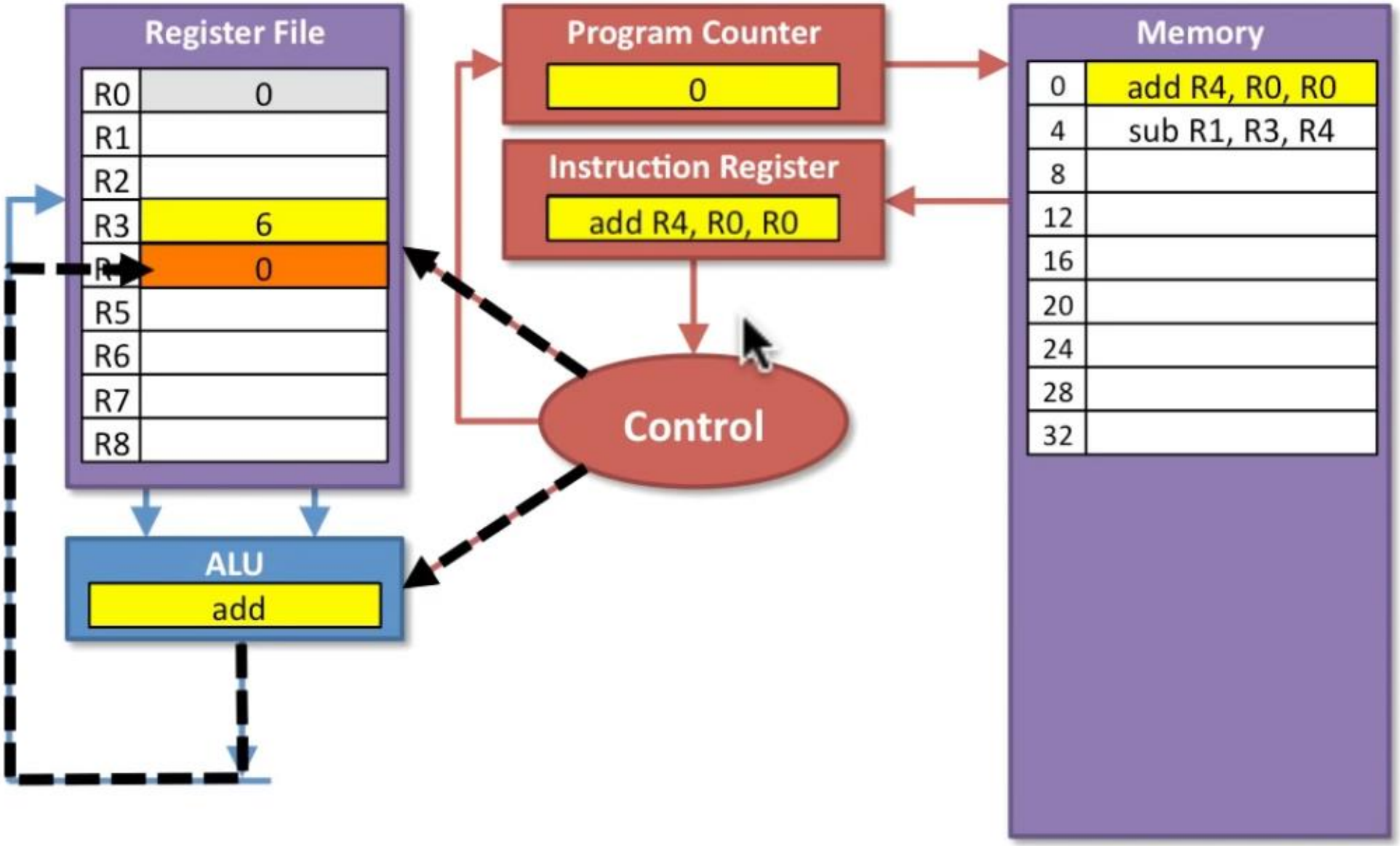
Data operations in detail

1. **Program Counter** holds the instruction address.
2. Instructions are ***fetched*** from memory into the **Instruction Register**.
3. **Control** logic ***decodes*** the instruction and tells the **ALU** and **Register File** what to do.
4. **ALU** ***executes*** the instruction and results flow back to the **Register File**.
5. The **Control** logic ***updates*** the **Program Counter** for the next instruction.

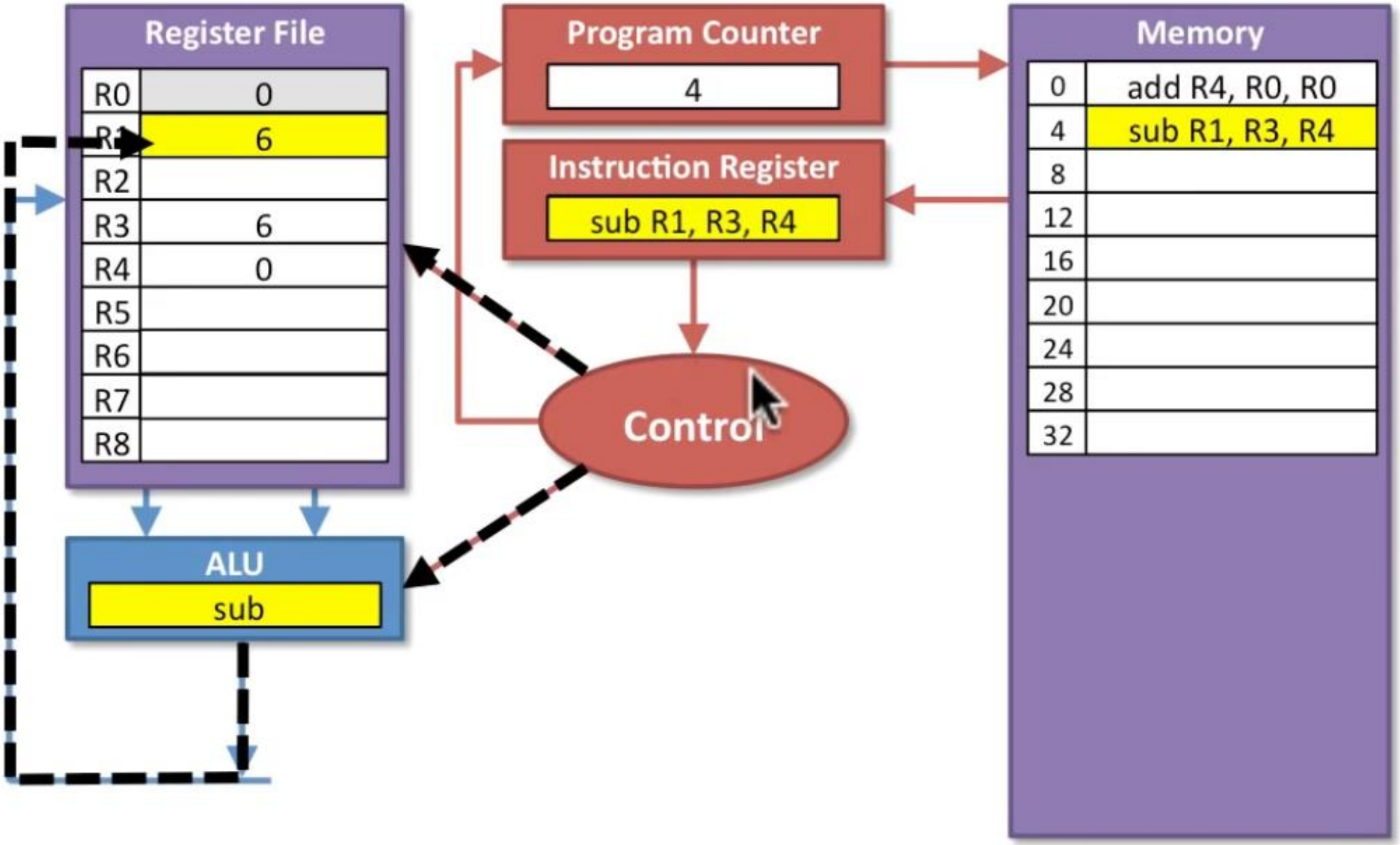


ALU = Arithmetic Logic Unit (Compute)

Add/sub example (1 of 2)



Add/sub example (2 of 2)



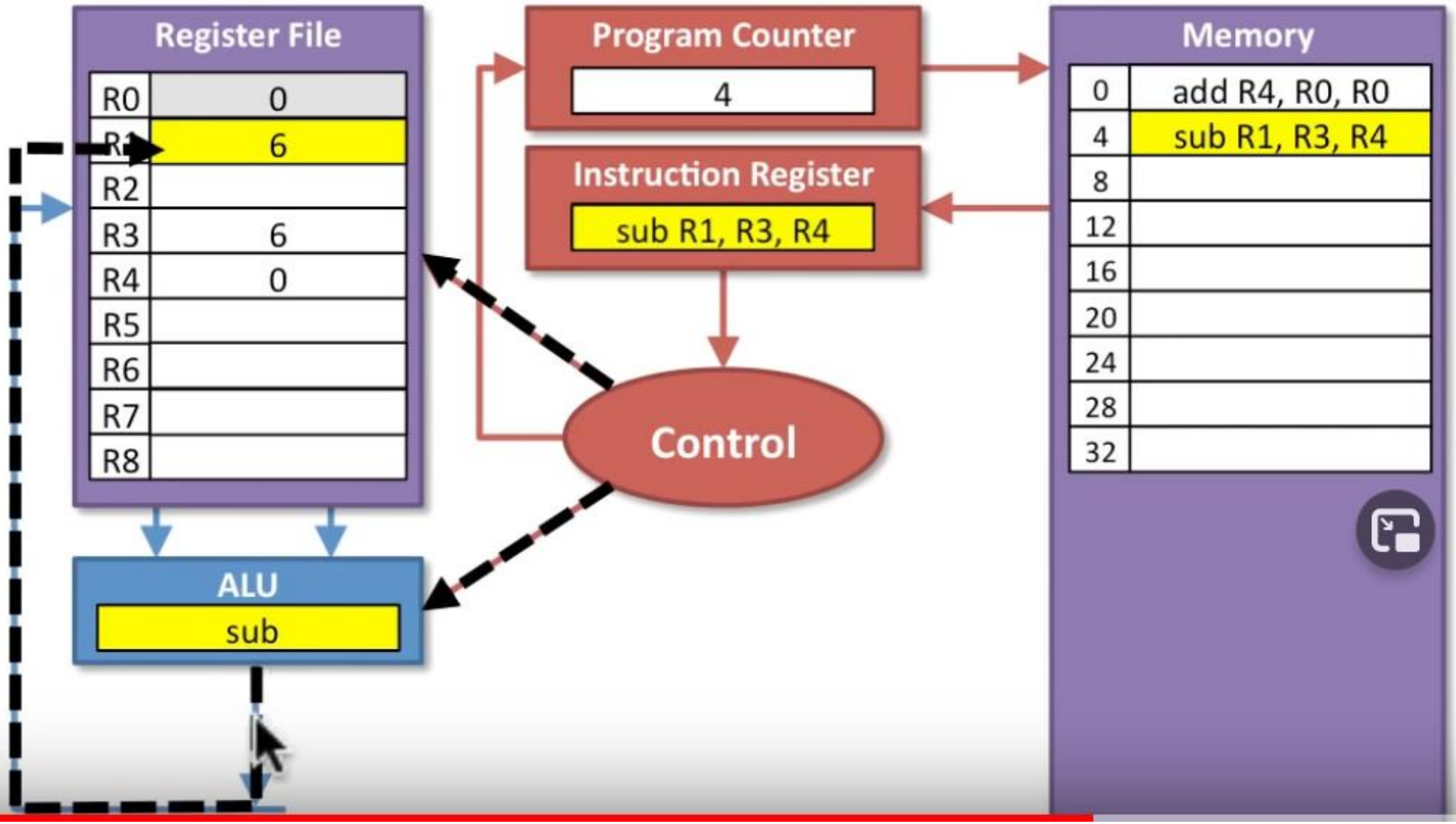
Add/sub example (2 of 2)

Program Counter (PC) specifies the address of the instruction to fetch.

The **Instruction Register** holds the current instruction.

Control tells the **ALU** and **Register File** what to do.

ALU computes the result and writes it back into the **Register File.**



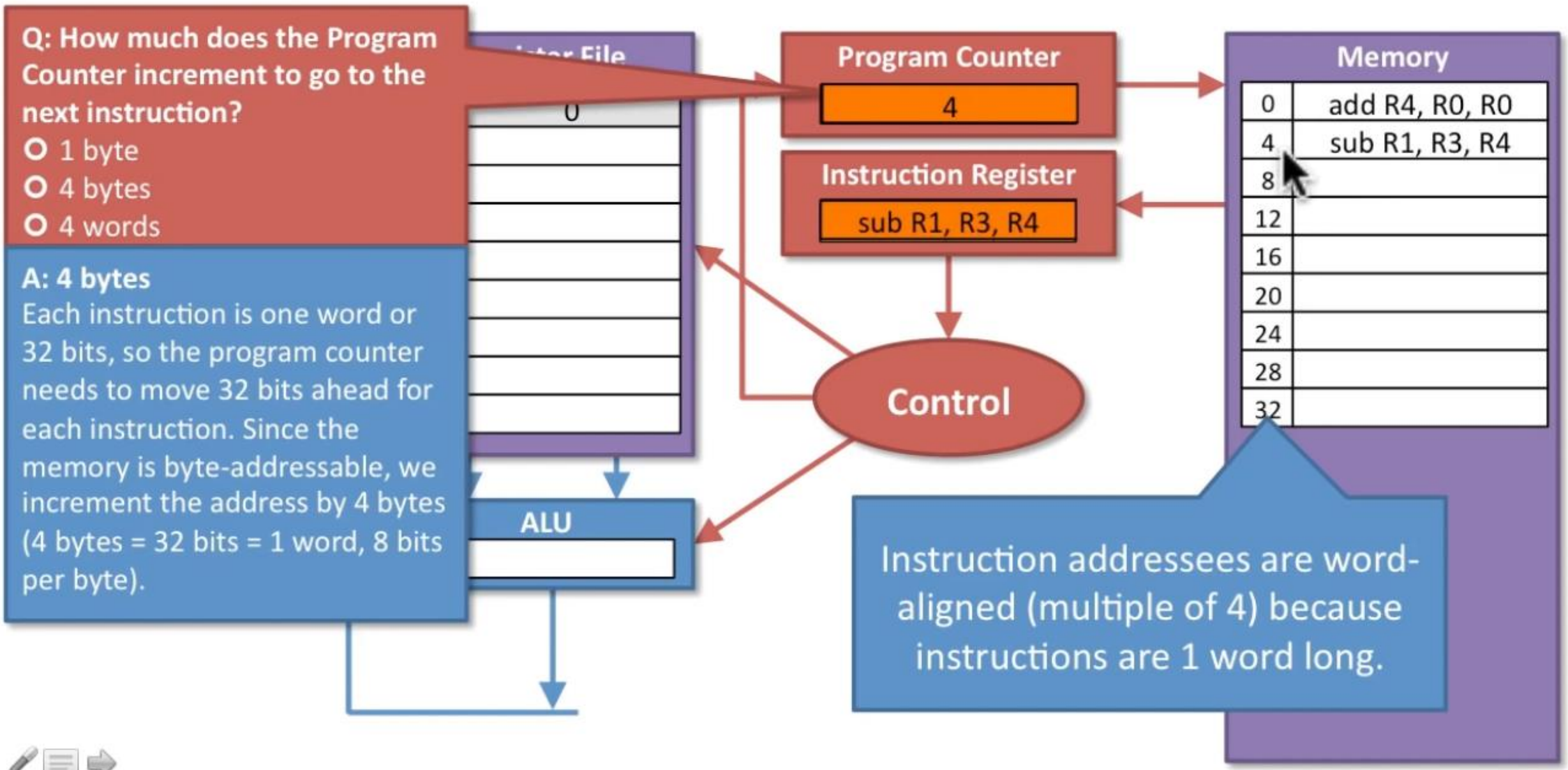
Question: incrementing the PC

Q: How much does the Program Counter increment to go to the next instruction?

- 1 byte
- 4 bytes
- 4 words

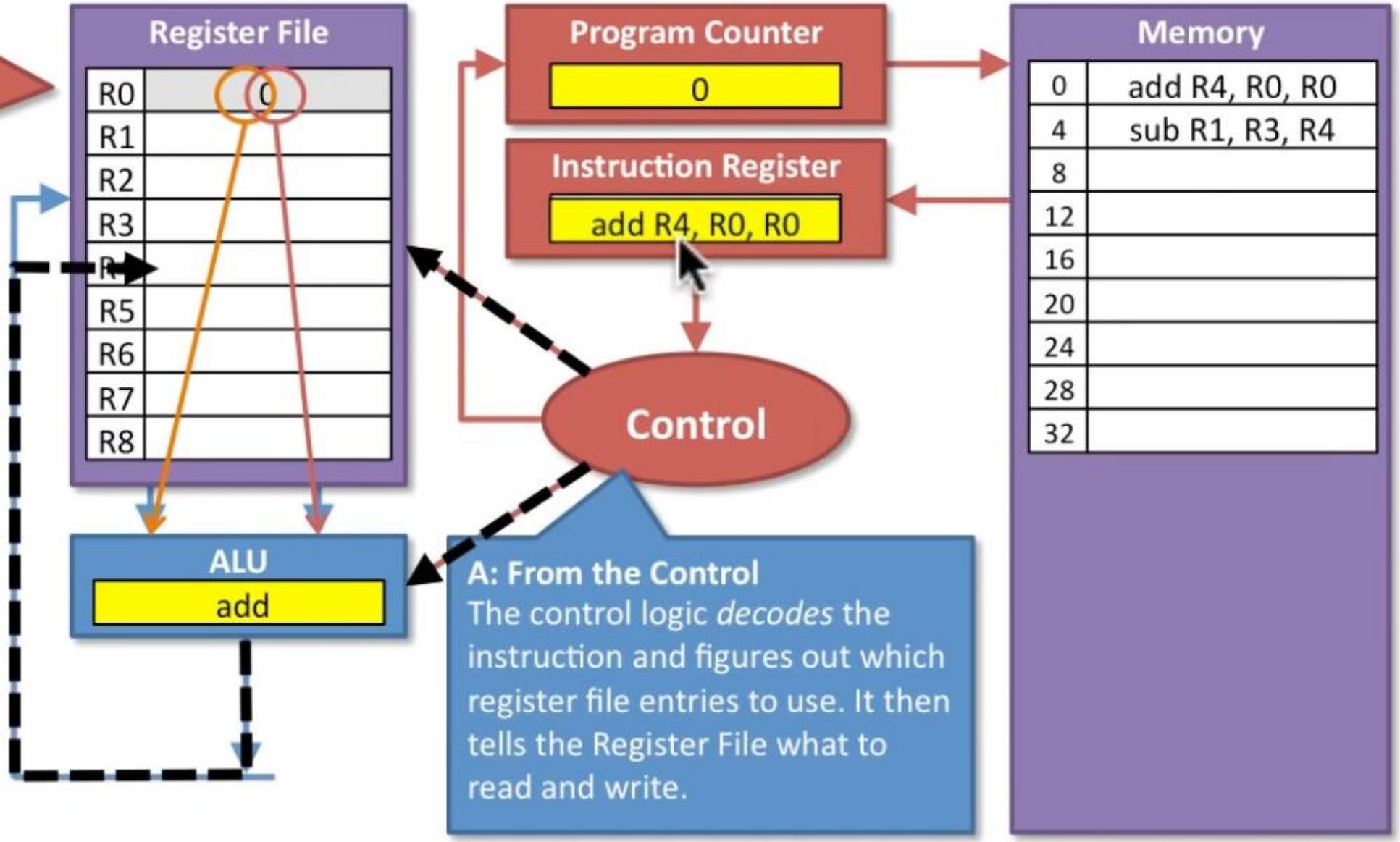
A: 4 bytes

Each instruction is one word or 32 bits, so the program counter needs to move 32 bits ahead for each instruction. Since the memory is byte-addressable, we increment the address by 4 bytes (4 bytes = 32 bits = 1 word, 8 bits per byte).



Question: controlling the register file

Q: Where does the the Register File get the information on which registers to read and write from? (click on the right part in the diagram)

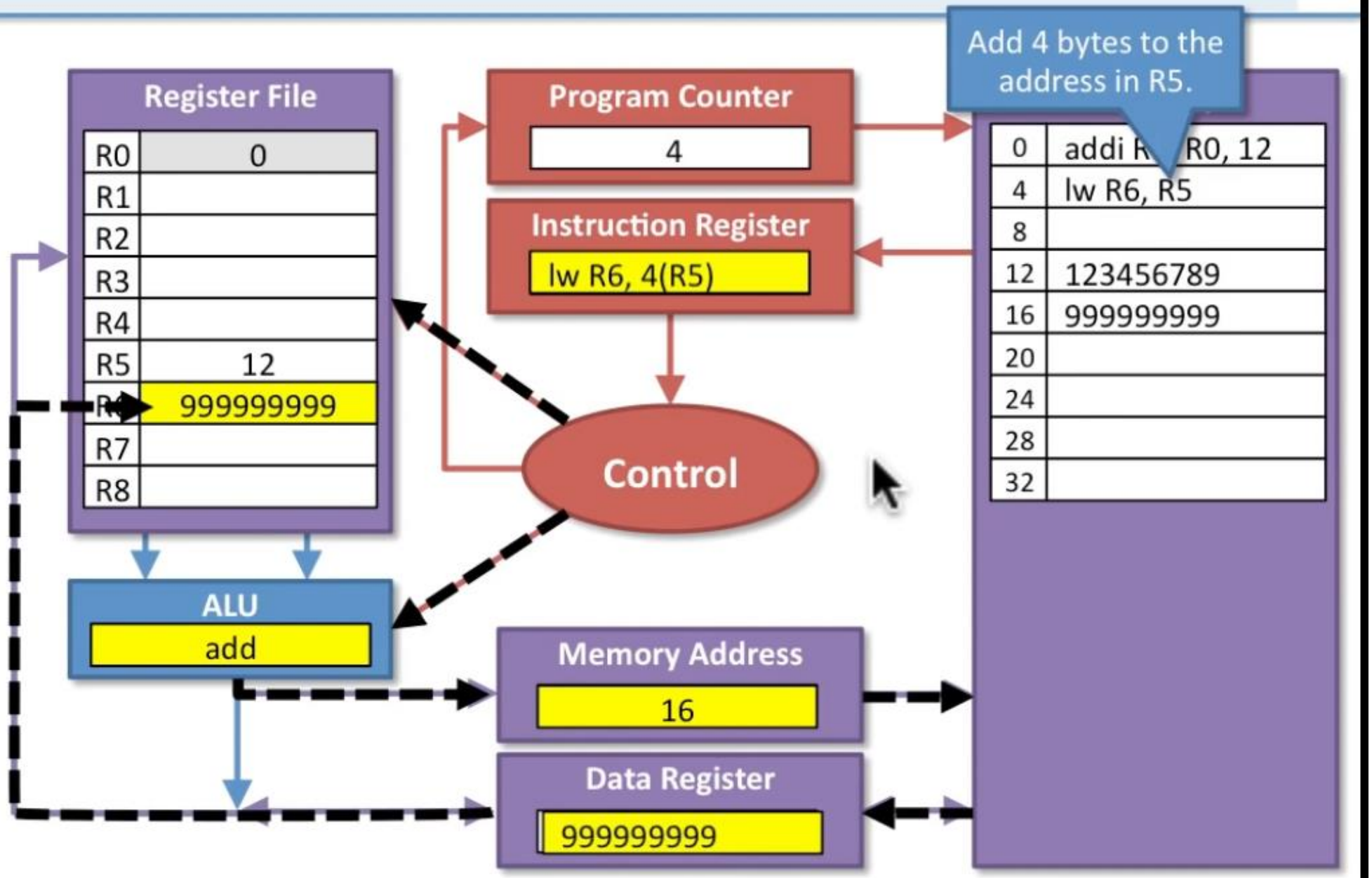


A: From the Control
The control logic *decodes* the instruction and figures out which register file entries to use. It then tells the Register File what to read and write.

Load word with offset

An offset can be added to addresses as part of the lw/sw instructions.

lw R6, 4(R5)

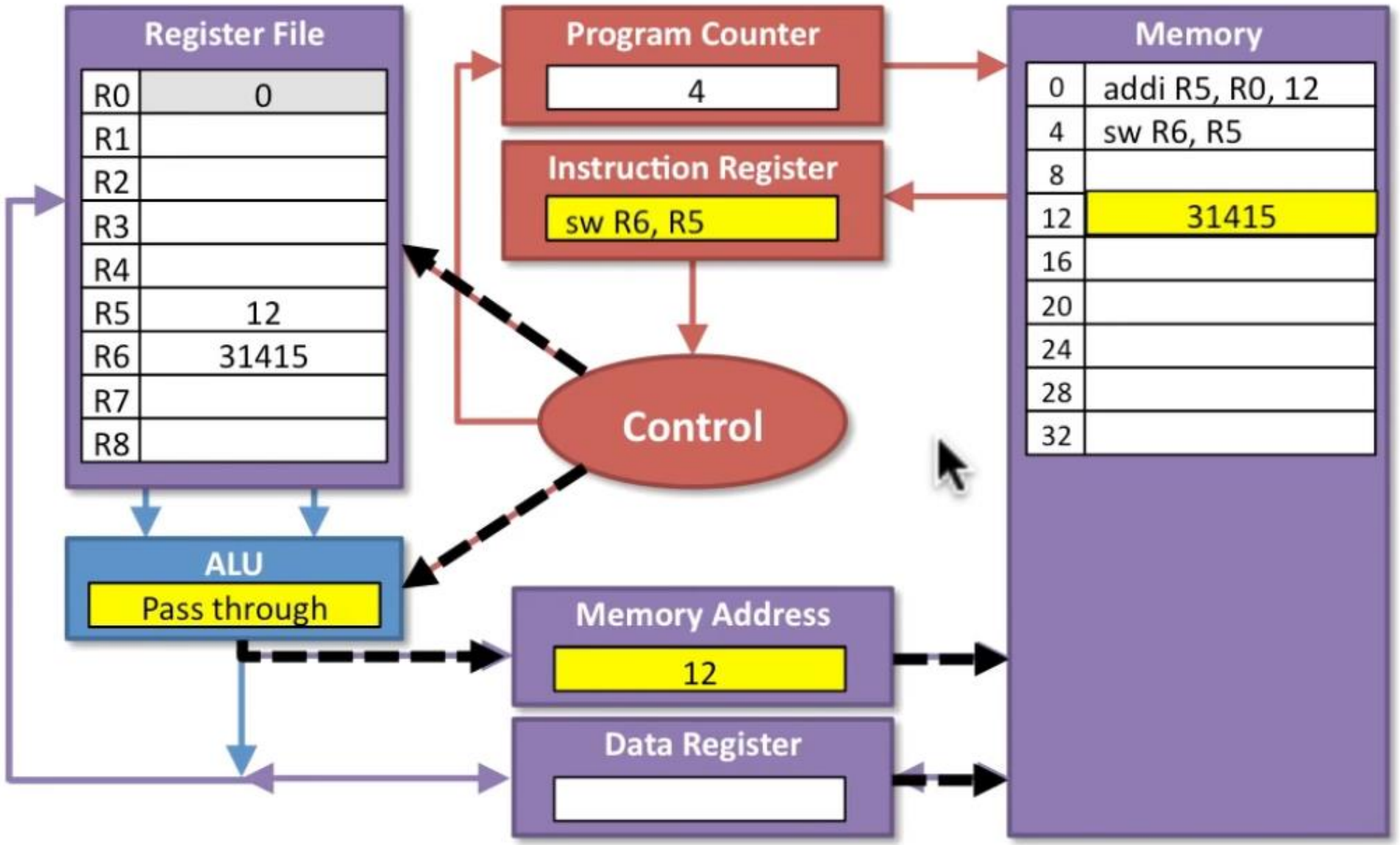


Store word example

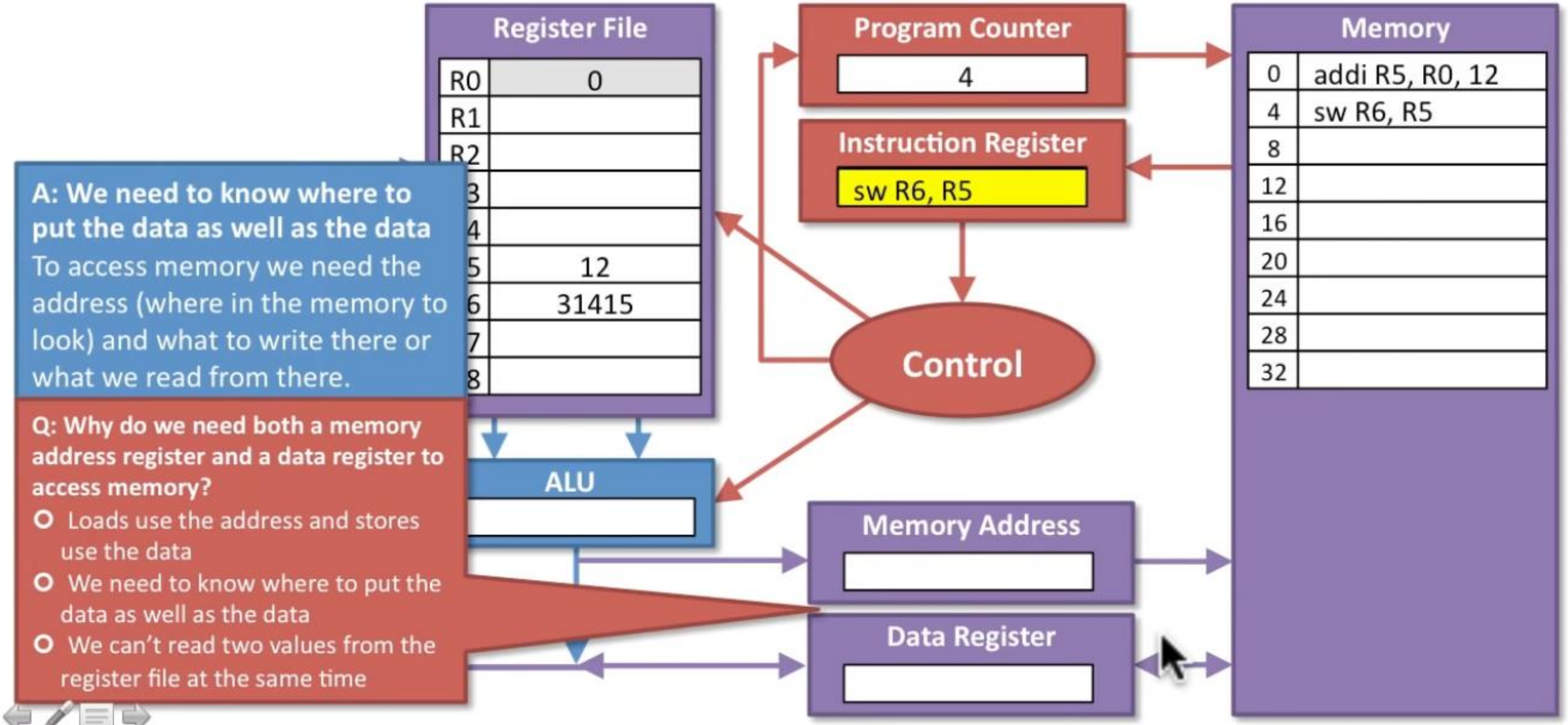
For stores we need:

the address (from ALU)

the data (from register)



Question: accessing memory



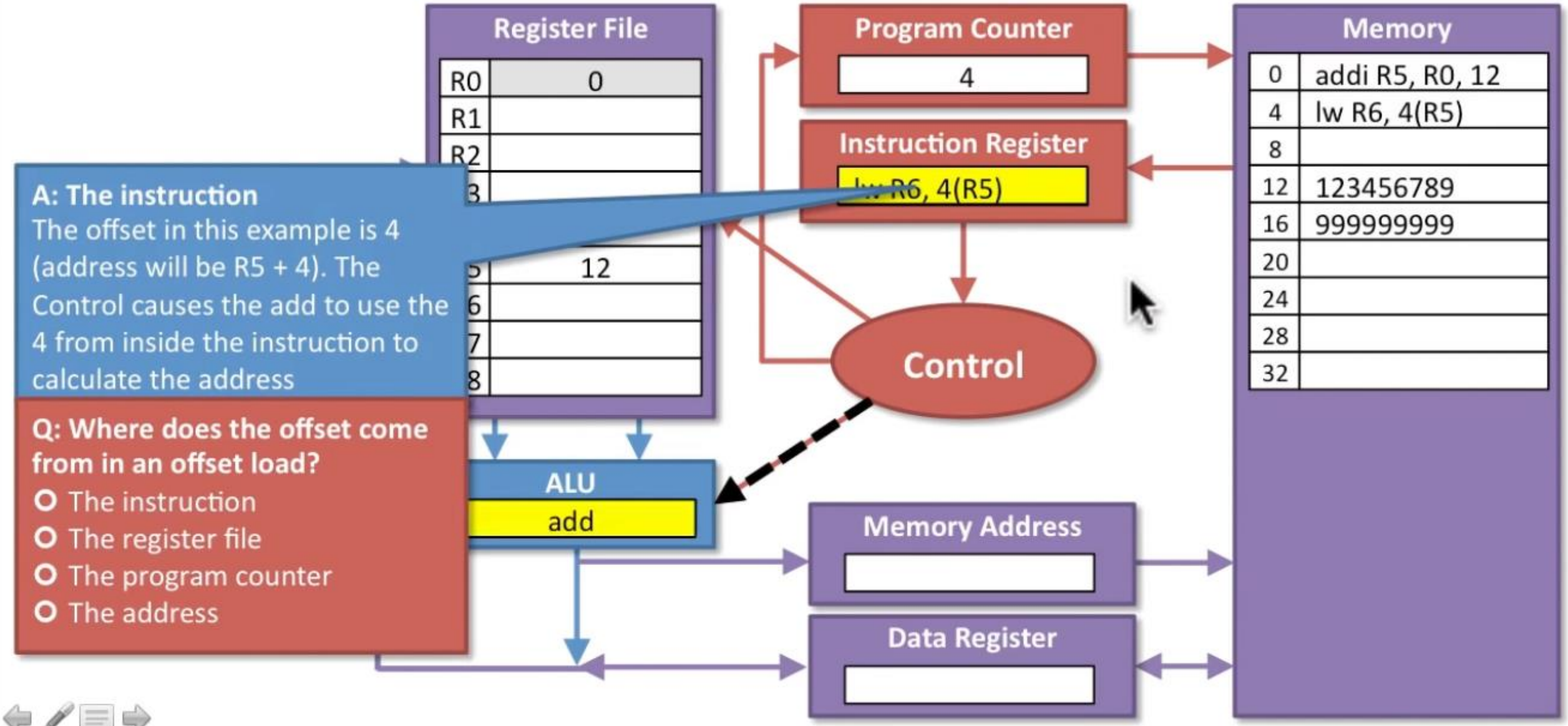
A: We need to know where to put the data as well as the data

To access memory we need the address (where in the memory to look) and what to write there or what we read from there.

Q: Why do we need both a memory address register and a data register to access memory?

- Loads use the address and stores use the data
- We need to know where to put the data as well as the data
- We can't read two values from the register file at the same time

Question: offset calculation



A: The instruction

The offset in this example is 4 (address will be R5 + 4). The Control causes the add to use the 4 from inside the instruction to calculate the address

Q: Where does the offset come from in an offset load?

- The instruction
- The register file
- The program counter
- The address

Sequencing (control instructions)

- **Sequencing instructions make decisions**
  - What instruction to execute **next**?
  - They change the "**control flow**" of the program

- **MIPS conditional branch instructions**
  - `bne R0, R1, Label` branch if *not equal* to label
  - `beq R3, R4, Label` branch if *equal* to label

- **Example:**

```
                    R1 = i; R2 = j; R3 = h

if (i==j)           bne R1, R2, Skip
  h = i+j;            add R3, R1, R2
...                 Skip:
                      ...
```

Branch to here if R1!=R2

Sequencing: unconditional jump

- MIPS unconditional branch instruction: jump
  - `j Label` jump to label
- Example:

```
                        R1 = i; R2 = j; R3 = h

if (i==j)               bne R1, R2, DoElse
  h = i+j;               add R3, R1, R2
                         j SkipElse
else                    DoElse:
  h = i-j;               sub R3, R1, R2
                        SkipElse:
...                     ...
```

Branch to here if R1!=R2

Always skip the subtraction if we did the addition.

Branch Instructions

- **Change the flow** of the program → **change the Program Counter**
  - `j` jump — goto label no matter what
  - `bne` branch not equal — goto label only if registers are not equal
- Example: `if (a==b) c=1; else c=2;`

```
R5 = a; R6 = b; R7 = c
```

	Instruction	Comment
`if (a==b)`	`bne R5, R6, Else`	`; if (a!=b) goto Else`
`c=1;`	`addi R7, R0, 1`	`; c <-- 1+0`
	`j SkipElse`	`; goto SkipElse`
`else c=2;`	`Else: addi R7, R0, 2`	`; c <-- 2+0`
`...`	`SkipElse: ...`	

Always skip setting to 2 if we set it to 1.

Sequencing: Loops


```
for (j=0; j<10; j++) {
  b = b + j;
}
...
```

R5 = j; R6 = b;

Remember that bne/beq only compare **registers** to **registers**. We need to put 10 in a **register** first!

We need the **constant** 10 for the loop comparison, so put it in **register** R1.

```
          Instruction              Comment
          addi R5, R0, 0           ; j ← 0 + 0
          addi R1, R0, 10          ; R1 ← 0 + 10
Loop:     beq  R5, R1, Exit        ; if ( j == 10) goto Exit
          add  R6, R6, R5          ; b ← b + j
          addi R5, R5, 1           ; j ← j + 1
          j    Loop                ; goto Loop
Exit:     ...                      ; pop out of loop, continue
```

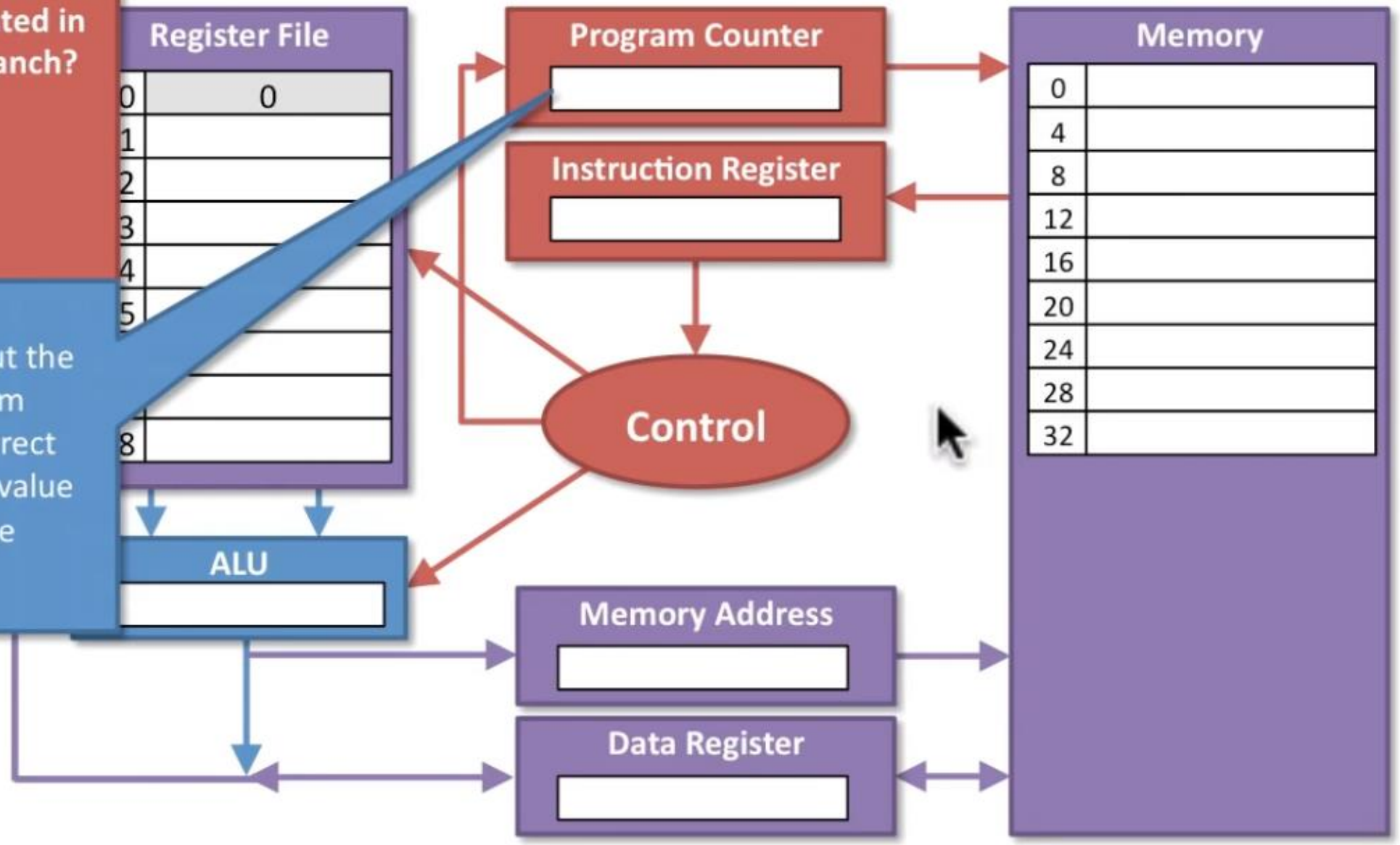
Question: handling branches

Q: What needs to be updated in the processor to take a branch?

- The instruction
- The register file
- The program counter
- The address

A: The program counter

On a branch we need to put the correct value in the program counter so we load the correct next instruction. The right value will depend on whether the branch is taken.

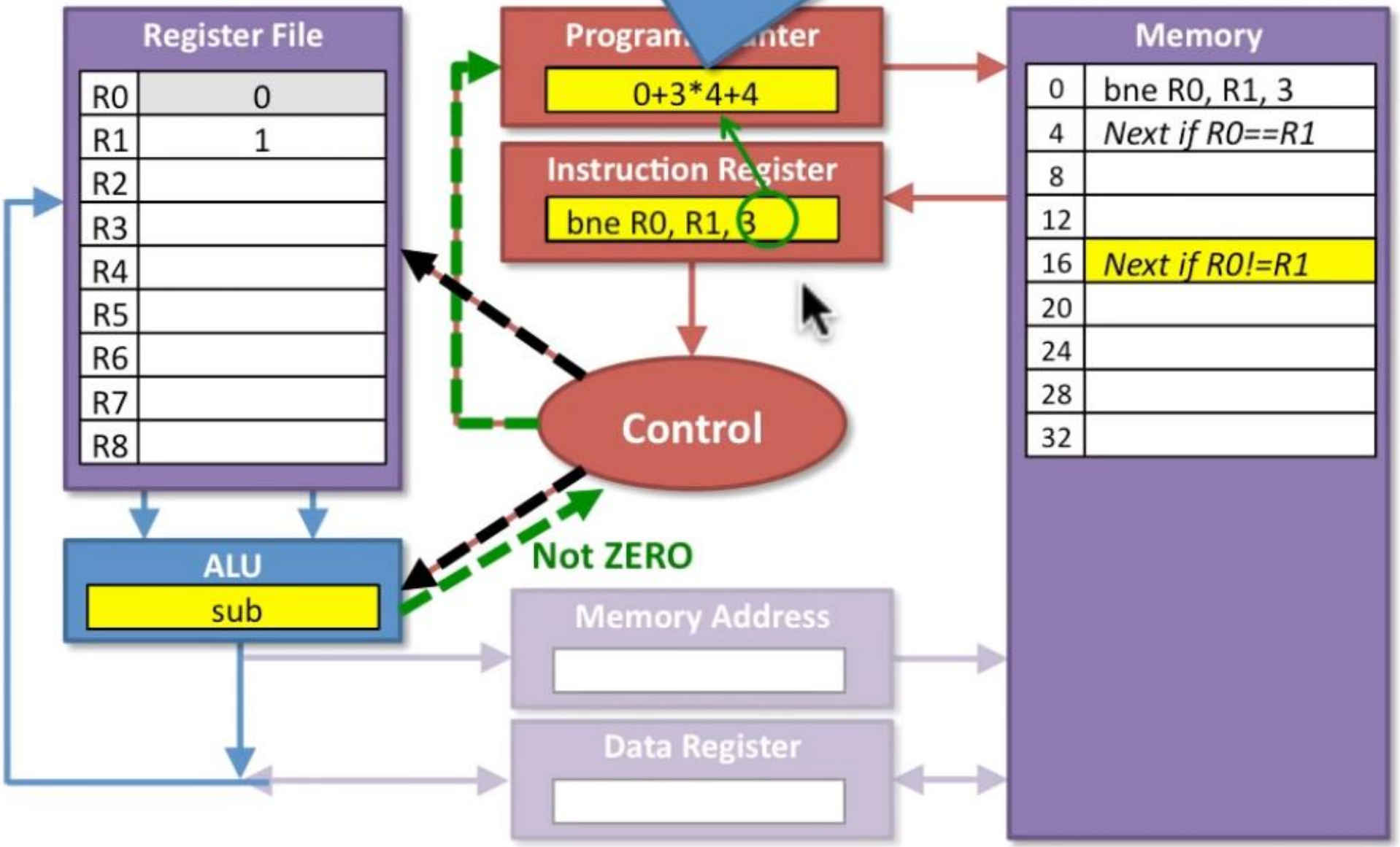


Sequencing in detail

1. ALU *compares* registers
2. Result tells the Control whether to branch
3. If the branch is *taken*, then the Control **adds** a constant from the instruction to the Program Counter
4. The Control always adds 4 to the Program Counter

For unconditional *jumps* the Control **replaces** the Program Counter with the constant from the instruction.

The label constant is in instruction **words**, so it needs to be multiplied by 4 to convert to **byte address**.



Question: if-then-else

Q: Why did we need both an unconditional jump and a conditional branch to do if-then-else?

- The conditional branch is for the if part and the jump for the else part
- You don't. You only need a conditional branch for the if part.
- Because you always need to jump over the else part if you do the if part.

A: Because you always need to jump over the else part if you do the if part.

If we didn't have the unconditional jump, then we would do both the if part (add) and then go right through to the else part (sub). The unconditional jump forces us to skip the else part if we do the if part.

R1 = i; R2 = j; R3 = h

```
if (i==j)
  h = i+j;

else
  h = i-j;

...
```

```
  bne R1, R2, DoElse
  add R3, R1, R2
  j SkipElse
DoElse:
  sub R3, R1, R2
SkipElse:
...
```

Always skip the subtraction if we did the addition.

Question: checking conditions

Q: How do we jump to label done if R3 is equal to 7?

A: Store 7 into a register (R2 in this case) and then compare if R3 is equal to that register.
Remember that we do not have a branch instruction that compares against an immediate value, so we have to do this in two instructions. (We'll see why in when we talk about how instructions are encoded into 32-bit values.)

```
...
addi R3, R0, 7
beq  R3, R0, done
...
```

```
...
j    done
...
```

```
...
beq R3, 7, done
...
```

```
...
addi R2, R0, 7
beq  R2, R3, done
...
```

Question: checking conditions

Q: How do we jump to label done if R3 is equal to 7?

A: Store 7 into a register (R2 in this case) and then compare if R3 is equal to that register.
Remember that we do not have a branch instruction that compares against an immediate value, so we have to do this in two instructions. (We'll see why in when we talk about how instructions are encoded into 32-bit values.)

```
...
addi R3, R0, 7
beq  R3, R0, done
...
```

```
...
j    done
...
```

```
...
beq R3, 7, done
...
```

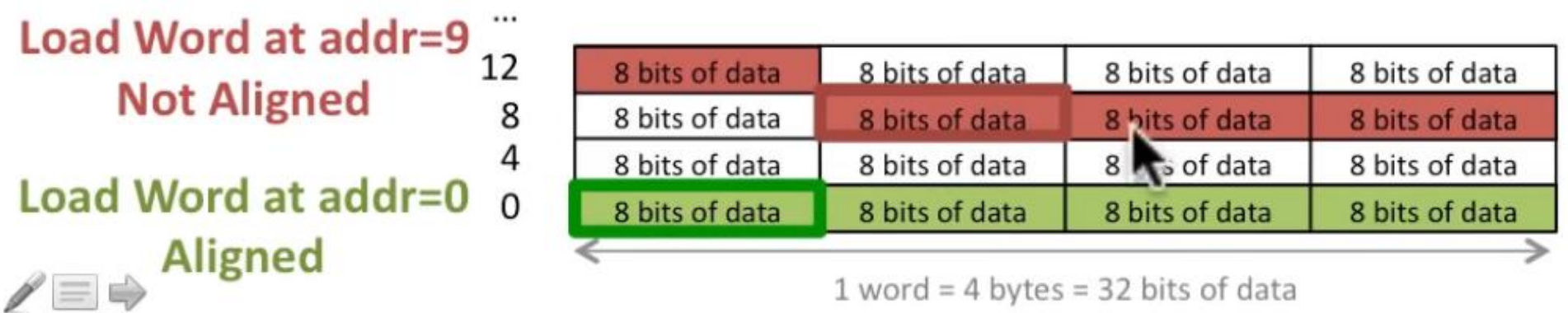
```
...
addi R2, R0, 7
beq  R2, R3, done
...
```

Remember: beq and bne only compare registers with registers.
To compare a register with a constant we have to:

1. Load the constant into a register (addi)
2. Compare the registers (bne/beq)

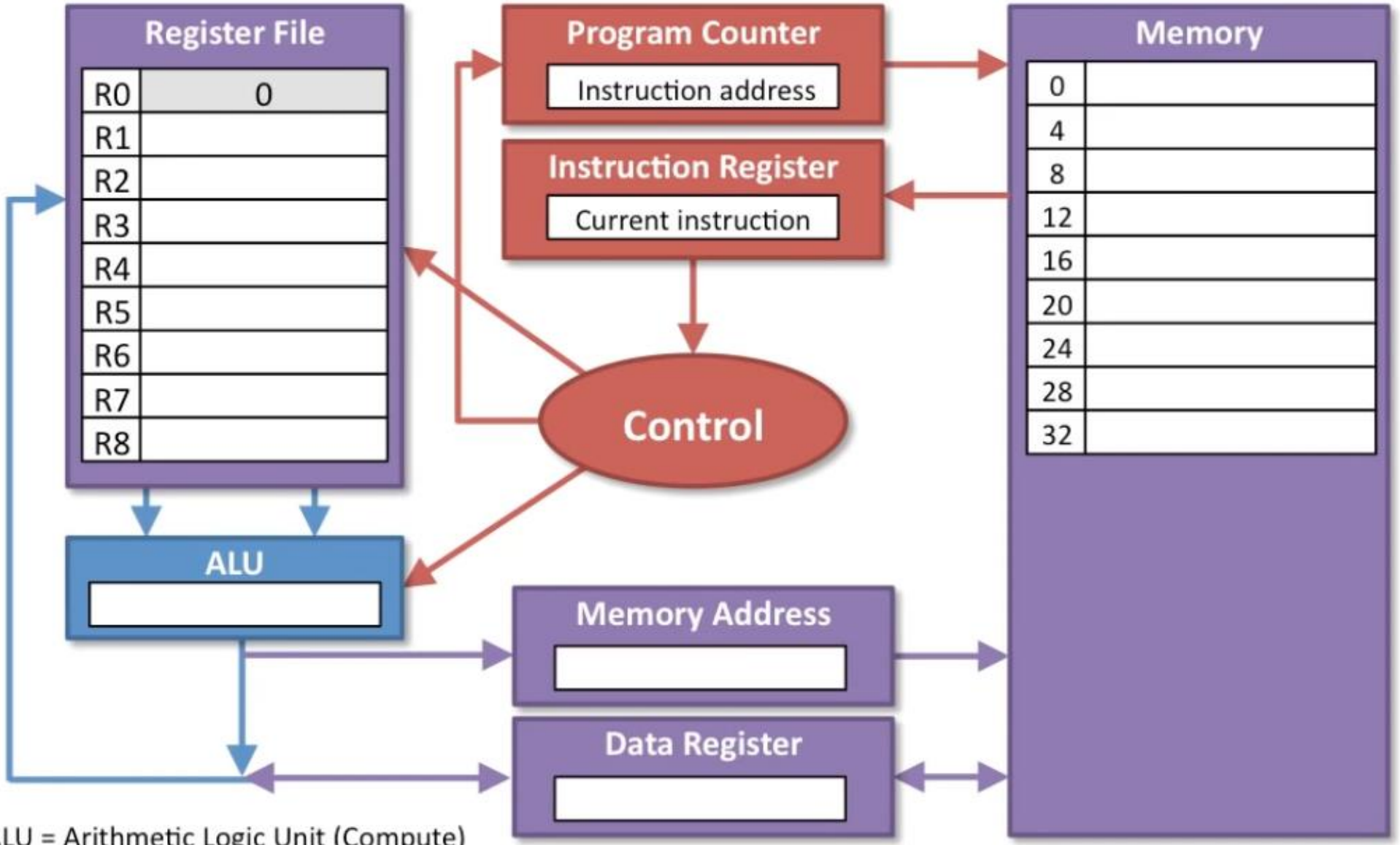
ISA 1 Summary 1: Instructions and Memory

- **Instructions**
  - op dest, src1, src2 — add R2, R3, R4 — R2 ← R3 + R4
  - Some can have constant values: — addi R2, R3, 12 — lw R4, 26(R2)
- **Memory**
  - Register file is 32 32-bit **words**, R0 is always 0
  - Memory is 4 billion **bytes**
  - MIPS uses word-aligned loads to load 32 bits (1 word) from memory at a time



ISA 1 Summary 2: Execution

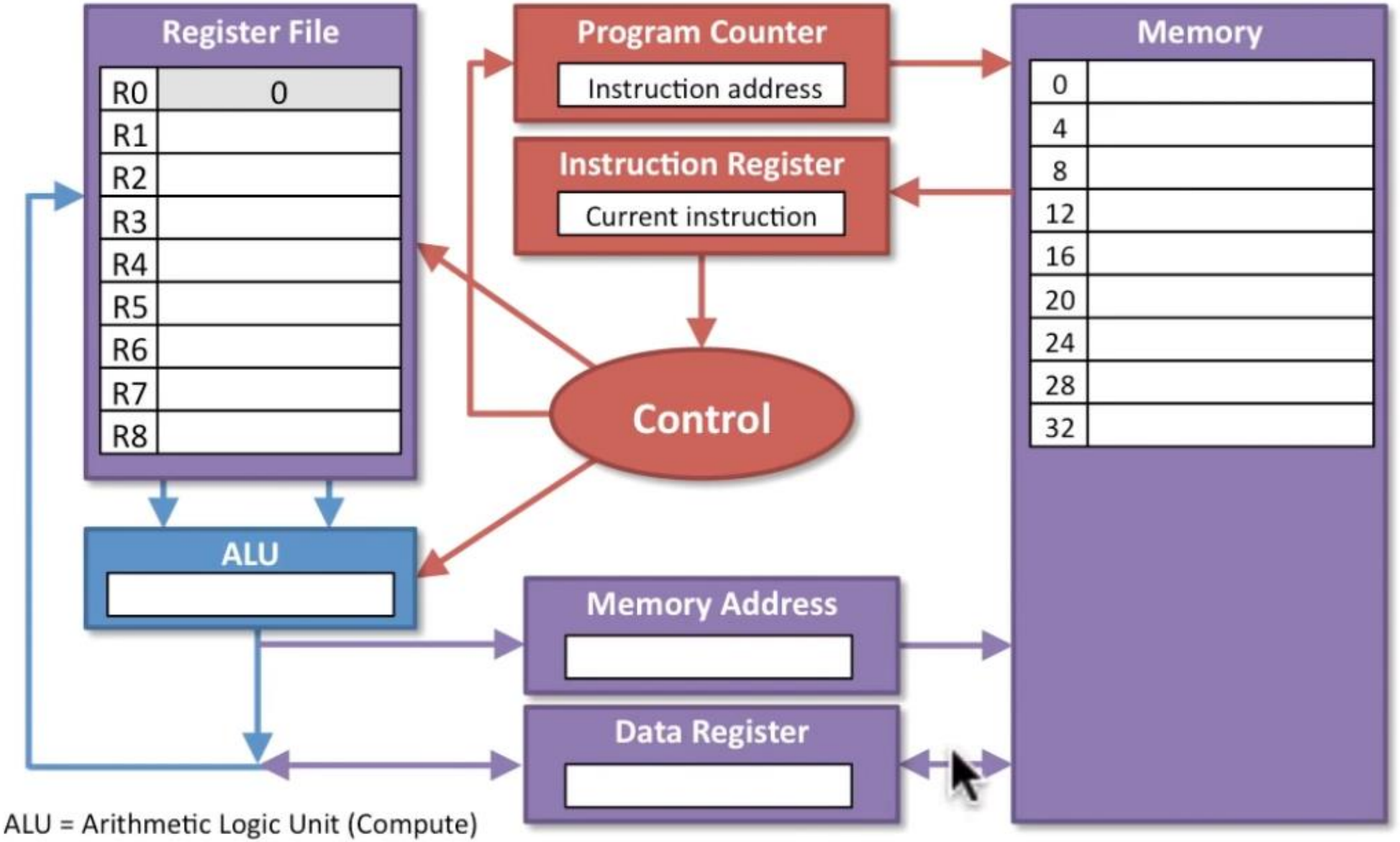
1. **Program Counter** holds the instruction address.
2. Instructions are ***fetched*** from memory into the **Instruction Register**.
3. **Control** logic ***decodes*** the instruction and tells the **ALU** and **Register File** what to do.
4. **ALU** ***executes*** the instruction and results flow back to the **Register File**.
5. The **Control** logic ***updates*** the **Program Counter** for the next instruction.
6. The **Memory Address register** and **Data Register** are used to load and store to/from **Memory**.



ALU = Arithmetic Logic Unit (Compute)

Review: execution

1. **Program Counter** holds the instruction address.
2. Instructions are ***fetched*** from memory into the **Instruction Register**.
3. **Control** logic ***decodes*** the instruction and tells the **ALU** and **Register File** what to do.
4. **ALU** ***executes*** the instruction and results flow back to the **Register File**.
5. The **Control** logic ***updates*** the **Program Counter** for the next instruction.
6. The **Memory Address register** and **Data Register** are used to load and store to/from **Memory**.



ALU = Arithmetic Logic Unit (Compute)

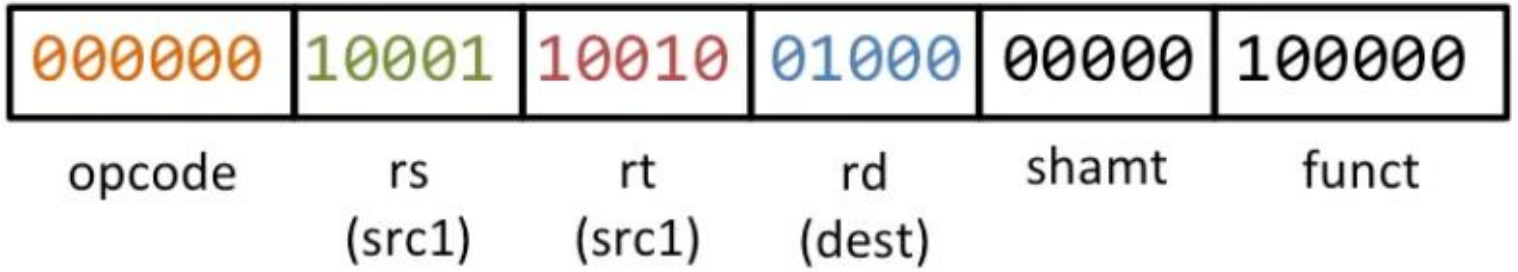
Instruction format (machine language)

- **Machine Language**
  - Computers do not understand “add R8, R17, R18”
  - Instructions are translated to machine language (1s and 0s)

- **Example:**

```
add R8, R17, R18 →
00000010 00110010 01000000 00100000
```

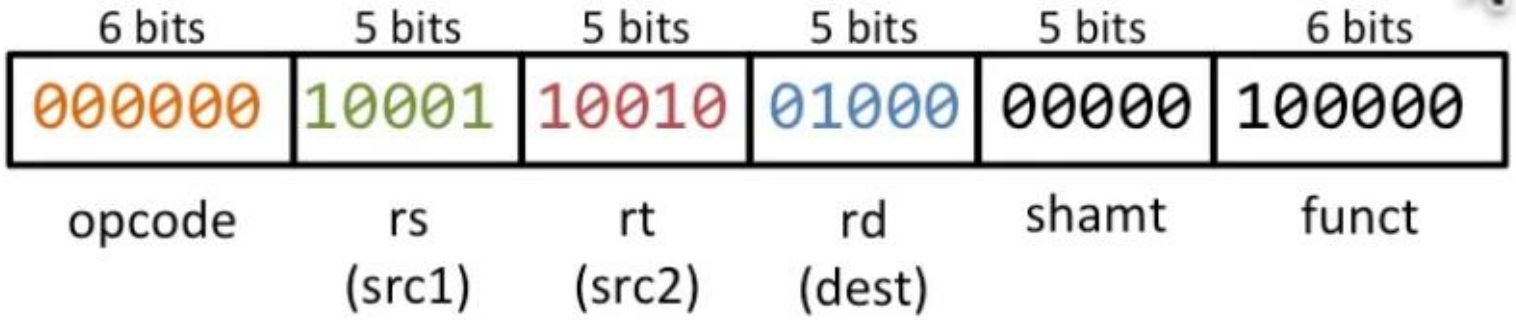
- MIPS instructions have logical fields:



Instruction fields

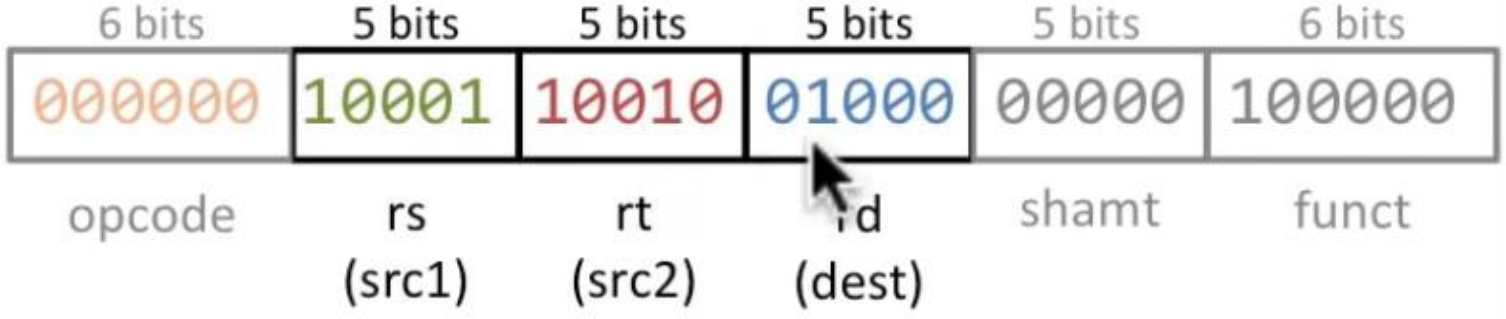
- **opcode** Operation (e.g., “add” “lw”)
- **rs** First source register
- **rt** Second source register
- **rd** Destination register
- **shamt** Shift amount
- **funct** Function selector (add = 32, sub =34)

Remember from 2’s complement: subtraction is basically addition, so it makes sense to share an opcode.



Question: number of bits

- **opcode** Operation (e.g., “add” “lw”)
- **rs** First source register
- **rt** Second source register
- **rd** Destination register
- **shamt** Shift amount
- **funct** Function selector (add = 32, sub =34)



Q: Why are there 5 bits for each of the registers rs, rt, and rd?

- Need 5 bits to specify the memory address
- Need 5 bits to specify the register
- Need 4 bits to specify the register and 1 bit to specify if it is read or write

A: Need 5 bits to specify the register

We have 32 registers and 2^5=32. Whether we read or write the register is defined by the field (rd vs. rs/rt) and the opcode.

Constants (immediate)

- **Small constants (immediates) are used all over code (~50%)**

```
if (a==b) c=1;
else      c=2;
```

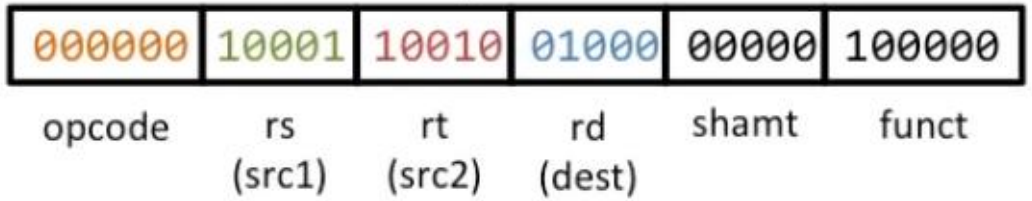
- **How can we support this in the processor?**
  - Put the "typical constants" in memory and load them (slow)
  - Create hard-wired registers (like R0) for them (how many?)

Need lots of bits to choose among lots of constant registers.

- **MIPS does something in between:**
  - Some instructions can have **constants inside the instruction**
  - The control logic then sends the constants to the ALU
  - `addi R29, R29, 4` ← value 4 is inside the instruction

Store the constant data in the instruction, not the register file.

- **But there's a problem:**
  - Instructions have only 32 bits. Need some for opcode and registers.



  - **How do we tradeoff space for constants and instructions?**

MIPS instruction formats

- Different formats for different kids of instructions
- **MIPS has 3 instruction formats:**
  - R: operation **3** registers **no** immediate
  - I: operation **2** registers **16 bit** immediate
  - J: jump **0** registers **26 bit** immediate
- Formats use the instruction bits differently
  - Trade-off immediate space and registers

Name	Bit Fields						Notes
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	(32 bits total)
R-Format	op	rs	rt	rd	shmt	funct	Arithmetic, logic
I-format	op	rs	rt	**address/immediate (16)**			Load/store, branch, immediate
J-format	op	**target address (26)**					Jump

Question: how big is an immediate?

Q: How large an immediate can you have for addi? (The immediate bits are interpreted as two's complement.)

- 0 to 65,535 (0 to $+2^{16}$)
- -32,768 to 32,767 (-2^{16-1} to $+2^{16-1}-1$)
- -32,767 to 32,767 (-2^{16-1} to $+2^{16-1}$)
- 0 to 31 (0 to $+2^{5}-1$)

A: -32,768 to 32,767 (-2^{16-1} to $+2^{16-1}-1$)
The immediate field for the I-format instructions is 5+5+6 bits = 16 bits. A Two's complement 16-bit number goes from $-(2^{16-1})$ to $+(2^{16-1}-1)$. E.g.,:
1000 0000 0000 0000 = -2^{15} to
0111 1111 1111 1111 = $+2^{15}-1$

Name	Bit Fields						Notes
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	(32 bits total)
R-Format	op	rs	rt	rd	shmt	funct	Arithmetic, logic
I-format	op	rs	rt	**address/immediate (16)**			Load/store, branch, immediate
J-format	op	**target address (26)**					Jump

Sign extension

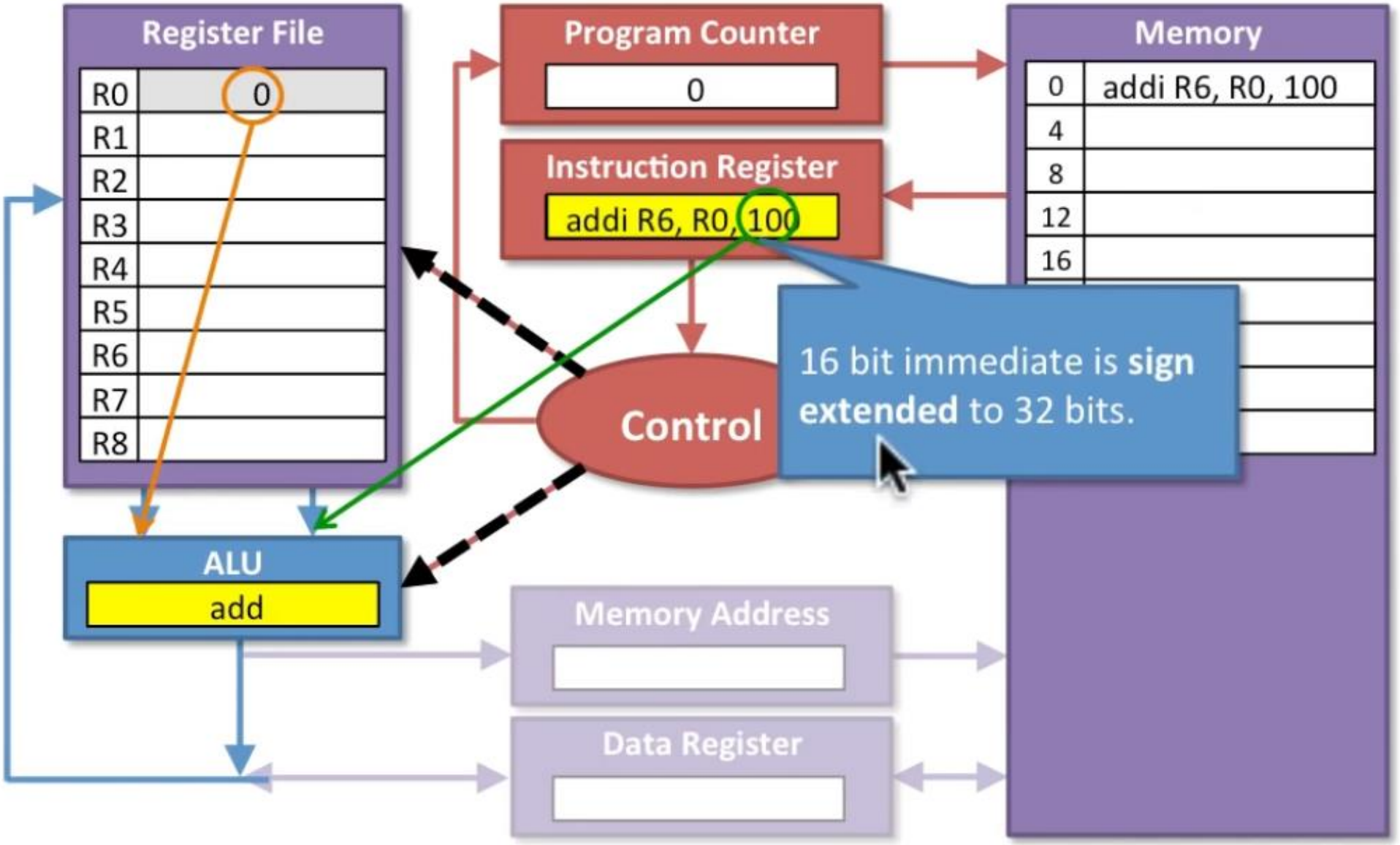
- How do we convert the 16-bit immediate field for use with the 32-bit register values?
- **Sign extension:**
  - Take the leftmost bit and repeat it

- Example: sign extend 8 bits to 16 bits
  - 100_{10} = **0**110 0100_2
  - Sign-extend to 16 bits: 0000 0000 **0**110 0100_2
- Example: sign extend 16 bits to 32 bits
  - -212_{10} = **1**111 1111 0010 1100_2
  - Sign-extend to 32 bits: 1111 1111 1111 1111 **1**111 1111 0010 1100_2

- Preserves the value of two's complement values

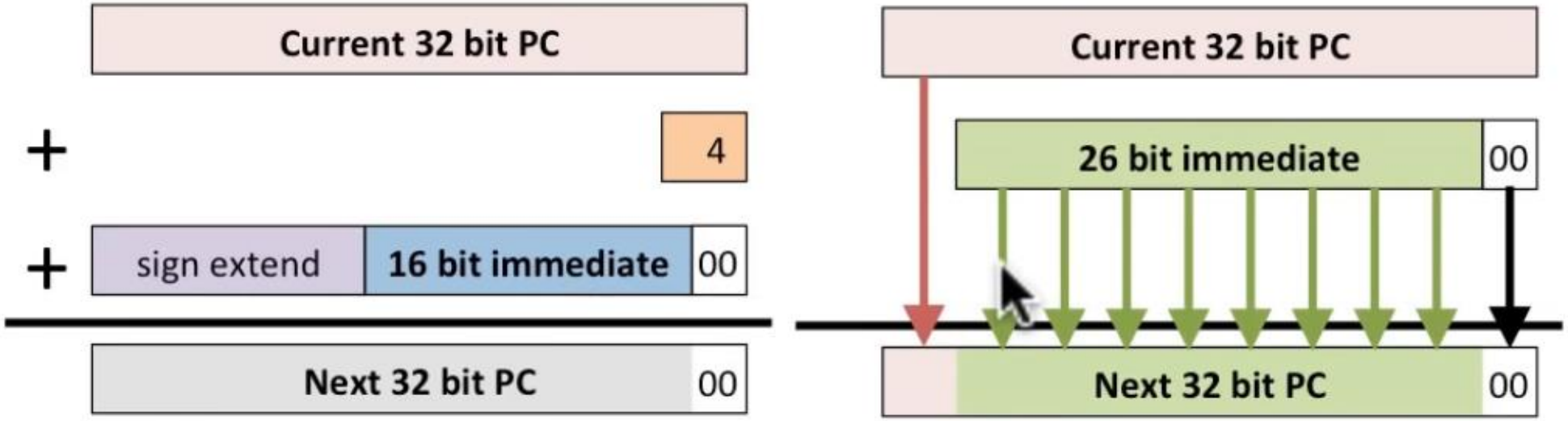
Loading immediate values (constants)

Control tells the ALU to take one operand from the **Register File** and the other from the **Instruction**.



Addresses in branches and jumps

- **Branch instructions**
  - **bne/beq** I-format **16 bit immediate**
  - **j** J-format **26 bit immediate**
- **But addresses are 32 bits!** How do we handle this?
  - Treat bne/beq as **relative offsets** (**add** to current PC)
  - Treat j as an **absolute value** (**replace** 26 bits of the PC)



Question: why 00s at the end?

Q: Why do we have 00 at the end of the immediate when we calculate the new instruction for both the branch and jump instructions?

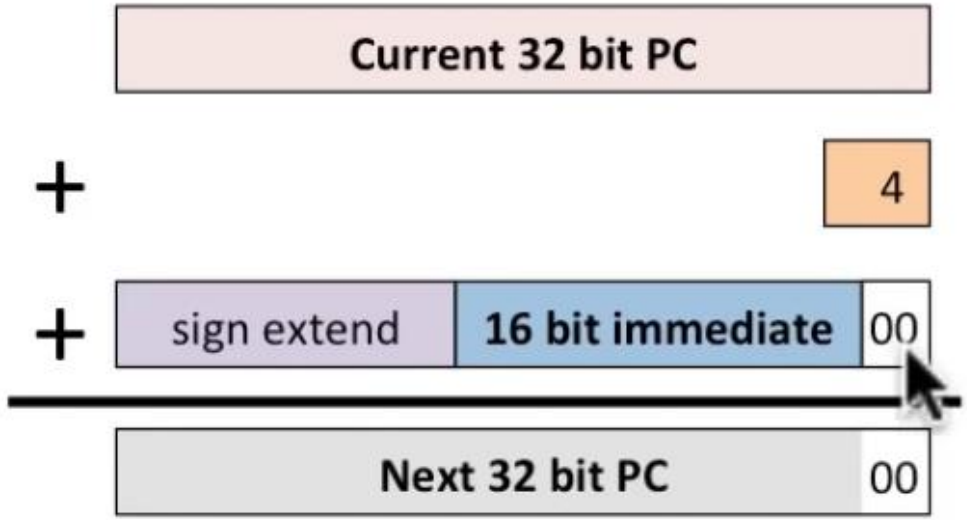
- Instructions are word-aligned so their addresses always end in 00
- We always jump by a full instruction, which is 4 bytes, so we multiply the jump offset by 4 by shifting it left two places
- We can jump further if the offset is in multiple of 4 bytes instead of 1
- All of the above

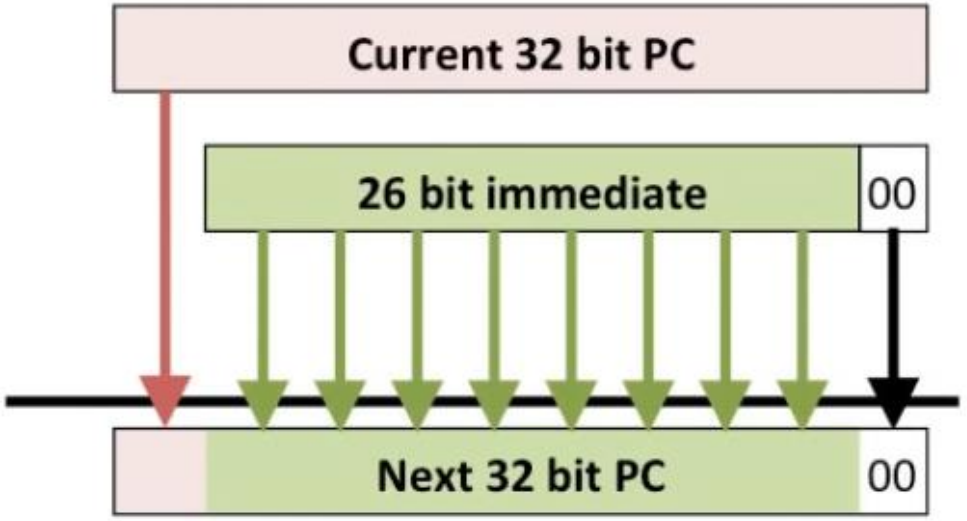
A: All of the above

Instructions are word-aligned (they are 4 bytes each) so we will never have anything other than 00 in the rightmost bits.

We always jump by full instructions (it doesn't make sense to jump into the middle of an instruction) and we can make sure of this by interpreting the jump offset as full instructions.

To convert to full instructions we multiply by 4, because each instruction is 4 bytes and memory addresses are in bytes.



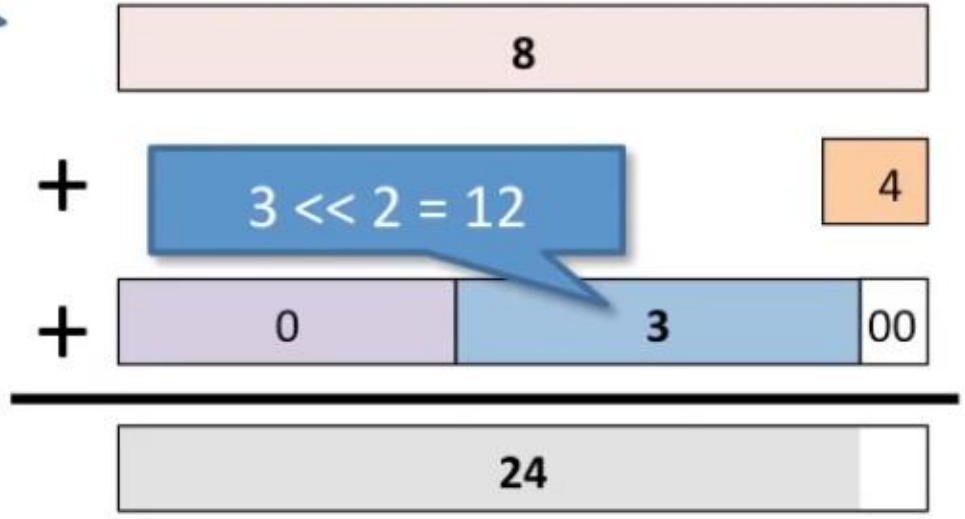


Jump addresses example: loops

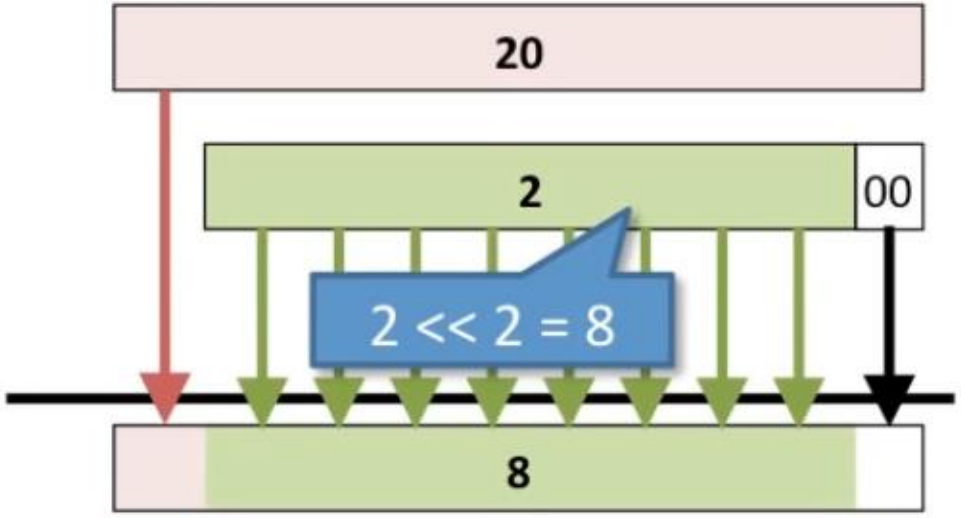
```
for (j=0; j<10; j++)
{
  b = b + j;
}
```

```
R5 = j;   R6 = b;
Addr      Instruction          Comment
0         addi R5, R0, 0       ; j ← 0 + 0
4         addi R1, R0, 10      ; R1 ← 0 + 10
8         beq  R5, R1, 3       ; if ( j == 10) goto 24
12        add  R6, R6, R5      ; b ← b + j
16        addi R5, R5, 1       ; j ← j + 1
20        j    2               ; goto 8
24        ...                  ; done with loop
```

Use labels and let the assembler figure it out.

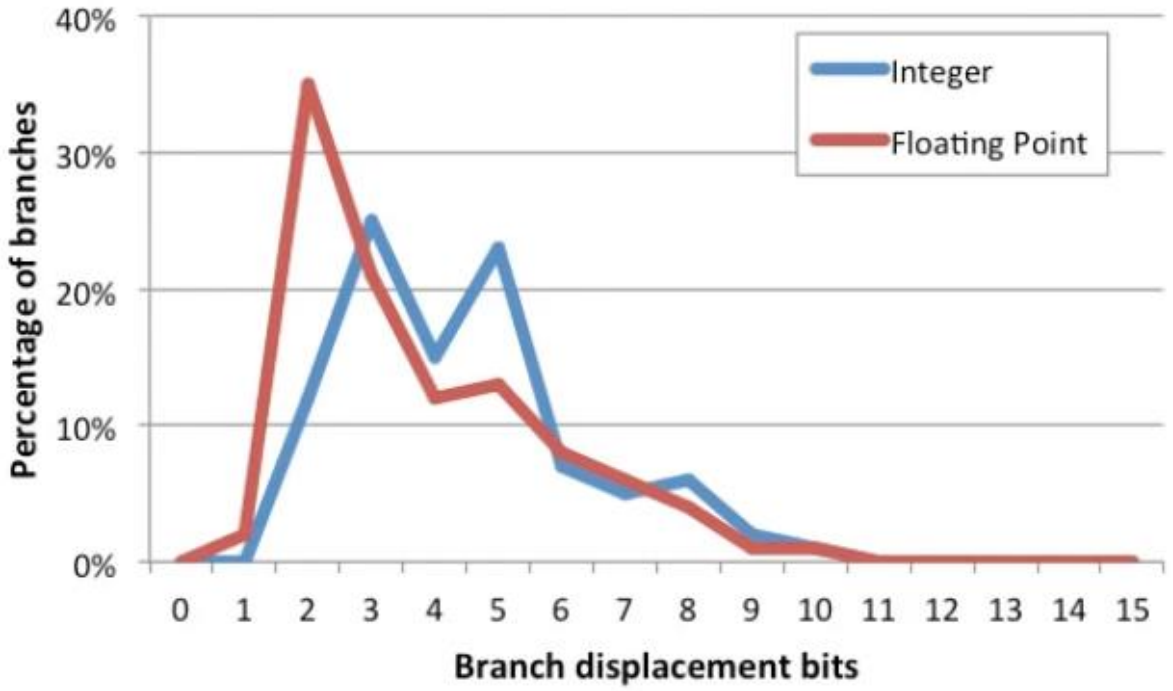


beq: PC=8+4+(3<<2)

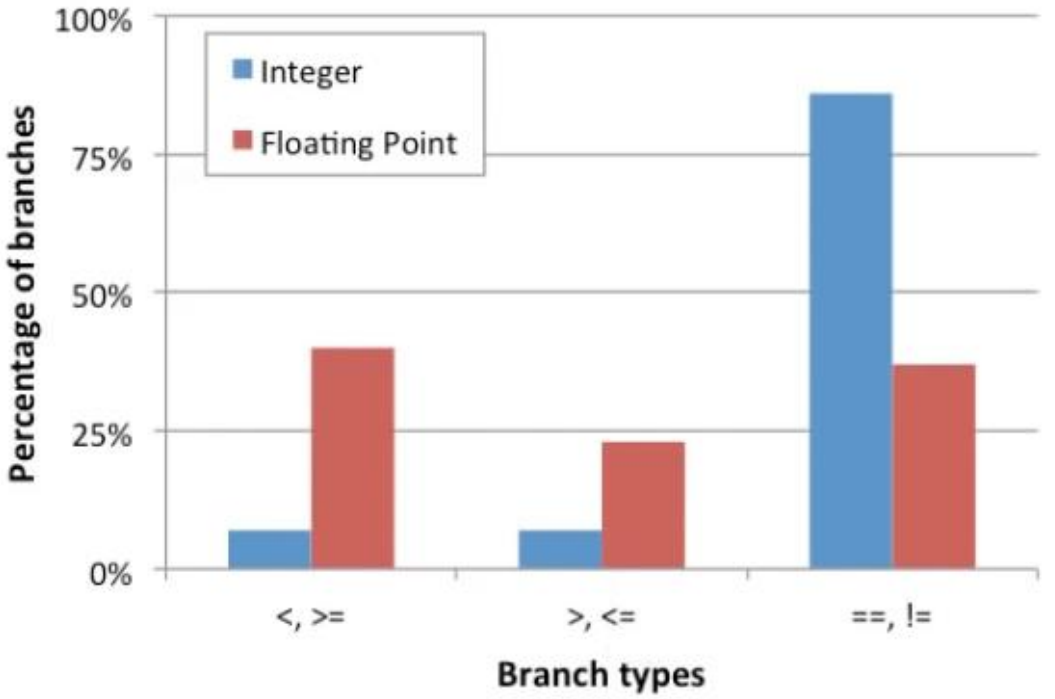


j: PC=[PC(31:28):2]<<2

What kind of branches are in programs?



16 bits of displacement is enough for almost everything.



Floating point (scientific) programs use greater than/less than a lot.

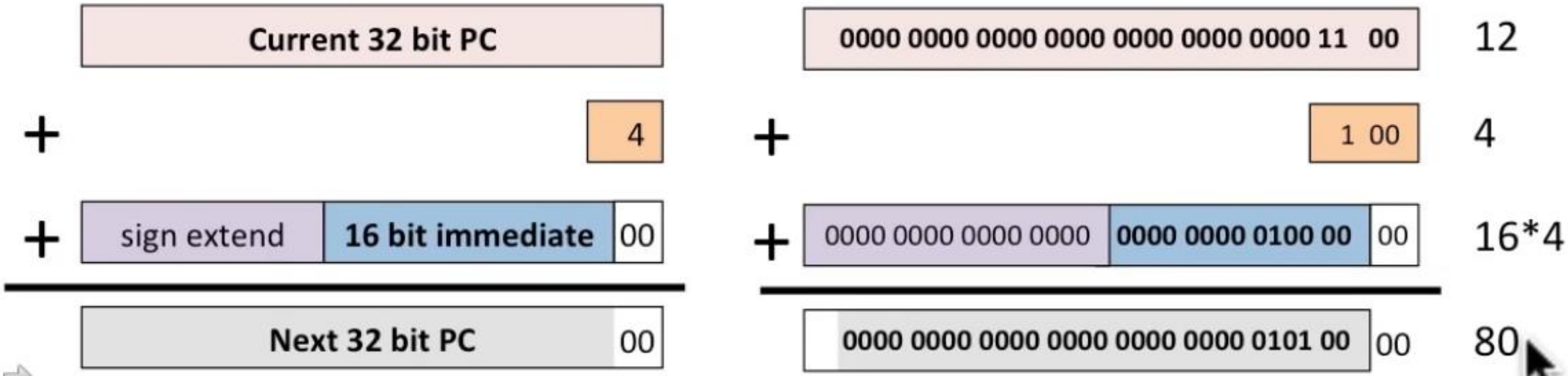
Question: jump destination

Q: To what address will the following instruction jump when the branch is taken? The instruction is at address 12.
bne R0, R1, 16

- 76
- 32
- 80
- 28

A: 80
The branch will jump to the current PC plus the immediate*4 (to convert instructions to bytes) plus the 4 we always add on.

12 + 16*4 + 4 = 80.



Loading larger values

- The immediate field is limited to 16 bits (-32,768 to +32,767)
  - How do we load larger values?
- Use two instructions to combine two 16 bit immediates
  - **Load Upper Immediate** (`lui`): Loads **upper** 16 bits
  - **Or Immediate** (`ori`): Loads **lower** 16 bits
- Example: `10101010 10101010 11110000 11110000`

```
lui R2, 10101010 10101010
```

puts zeros in the lower bits

```
R2:     10101010 10101010 00000000 00000000
```

```
ori R2, 11110000 11110000
```

```
R2:     10101010 10101010 11110000 11110000
```

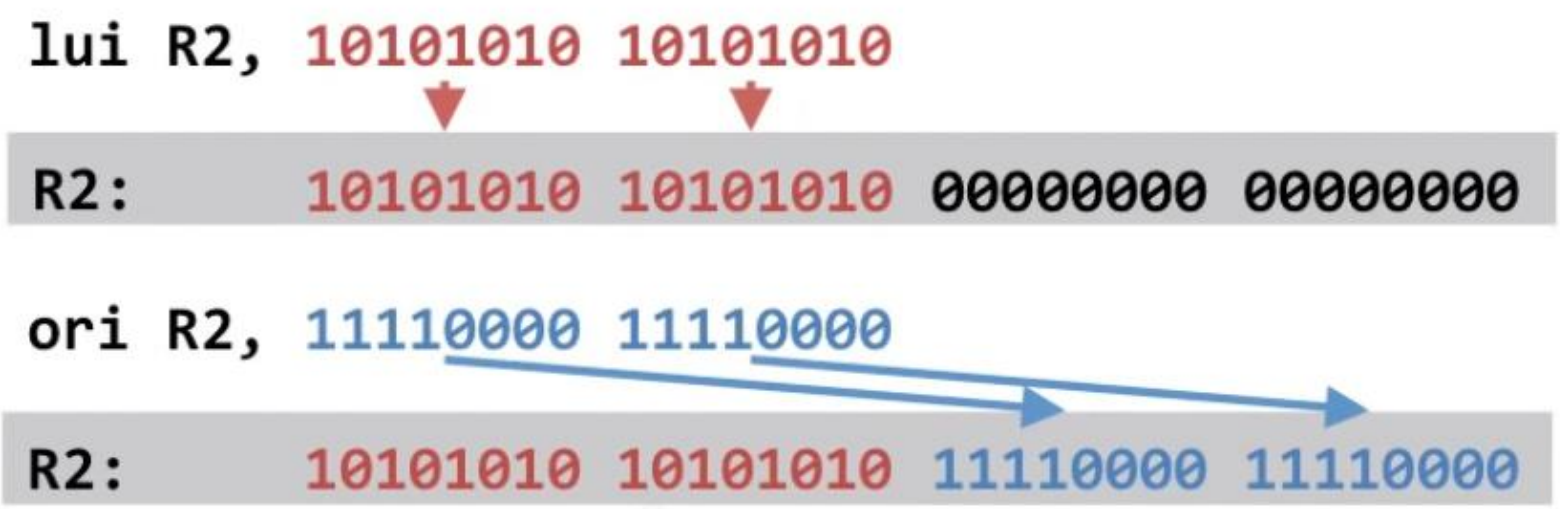
Question: why lui and ori?

Q: Why do we need both the lui and ori instructions to load a 32 bit constant?

- The ori loads the lower 16 bits and the lui shifts them to the upper half
- You need two instructions to get 32 bits of data
- The lui loads the upper 16 bits and the ori inserts the lower 16 bits
- You need to load data that isn't sign-extended

A: The lui loads the upper 16 bits and the ori inserts the lower 16 bits

lui = load **upper** immediate
Loads 16 bits into the **upper** half and puts zeros in the bottom
ori = or immediate
Ors the constant value into the **lower** 16 bits



Summary: machine code and immediates

- Instructions have **different encodings** to store **different types of data** (3 register vs. immediate)
- MIPS has 3 types, for different uses:

Name	Bit Fields						Notes
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	(32 bits total)
R-Format	op	rs	rt	rd	shmt	funct	Arithmetic, logic
I-format	op	rs	rt	**address/immediate (16)**			Load/store, branch, immediate
J-format	op	**target address (26)**					Jump

- Encodings limit how much data we can have
- **These are tradeoffs in design:**
  - **Optimize for the common case** (short immediates in 1 instruction)
  - **Support the general case** (long immidiates in 2 instructions)

Question: 16 bit to 32 bit immediate

Q: How is the 16 bit immediate field in addi converted to a 32 bit immediate for adding to a 32 bit register value?

- You can't get a 32 bit value from addi
- The top 16 bits are filled with zeros
- The data is replicated into the top 16 bits
- The 16 bits are sign-extended

A: The 16 bits are sign-extended

Arithmetic operations sign-extend the 16 bit immediates to 32 bits by replicating the leftmost bit. This does not allow you load a 32 bit value (you need the ori for that) but it allows you to do 32 bit math with a 16 bit immediate.

```
R3 = 12
addi  R4, R3, -12
```

R3:	0000 0000 0000 0000 0000 0000 0001 1000 = 12
16 bit immediate:	1111 1111 1111 0100 = -12
Sign Extend:	1111 1111 1111 1111 1111 1111 1111 0100 = -12
Add → R4:	0000 0000 0000 0000 0000 0000 0000 0000 = 0

Question: instruction format tradeoffs

Q: What does the I-format instruction tradeoff relative to the R-format?

- The 16 bit immediate field for more register operands
- All register fields for a long immediate
- The third register and shift bits for a 16 bit immediate

A: The third register and shift bits for a 16 bit immediate

The I-format uses the bits from the rd, shmt, and funct fields in the R-format for a 16-bit immediate.

Name	Bit Fields						Notes
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	(32 bits total)
R-Format	op	rs	rt	rd	shmt	funct	Arithmetic, logic
I-format	op	rs	rt	**address/immediate (16)**			Load/store, branch, immediate
J-format	op	**target address (26)**					Jump

Procedure calls

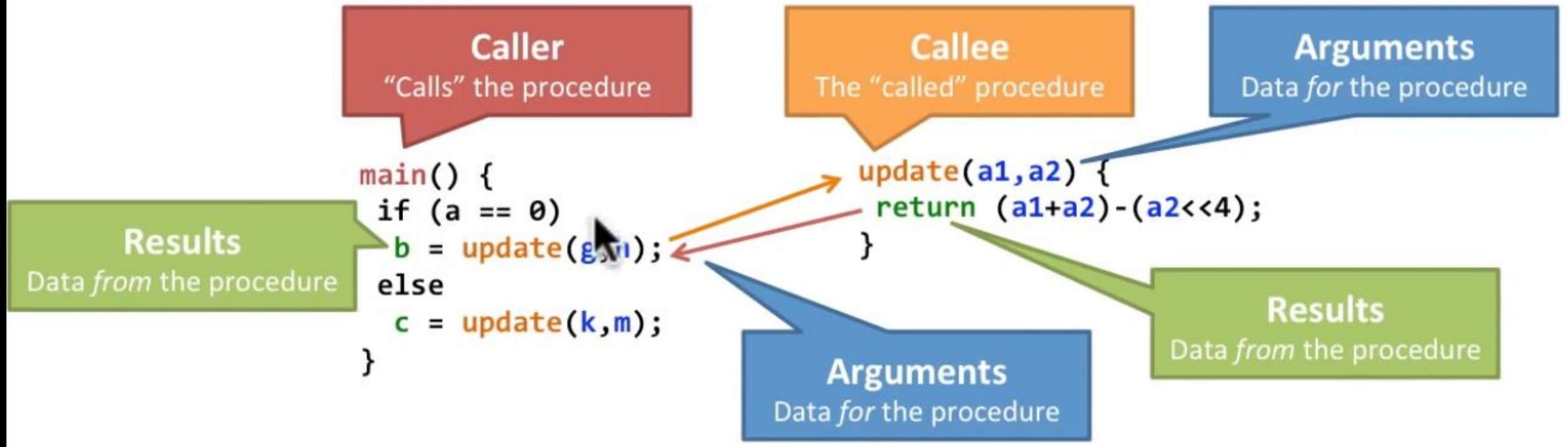
```
main() {
 if (a == 0)
  b = (g+h)-(g*16);
 else
  c = (k+m)-(k*16);
}
```

```
main() {
 if (a == 0)
  b = update(g,h);
 else
  c = update(k,m);
}
```

```
update(a1,a2) {
 return (a1+a2)-(a2<<4);
}
```

- **Procedures (functions/subroutines) are needed for structured programming**
  - Avoid repeated code
  - Call functions you didn't write (libraries)
- **What needs to happen:**
  - **Put data** where the procedure can access it
  - Start the **procedure**
    - Calculate
    - Put the **results** where the **caller** can access them
  - Return to the right place in the **caller**

Procedure call terminology



- The Caller calls the procedure
- The procedure is the Callee
- The Caller gives the Callee Arguments (data)
- The Callee returns Results (data) to the Caller

How to do a procedure call

- **Transfer** control to the **callee** to start the procedure:

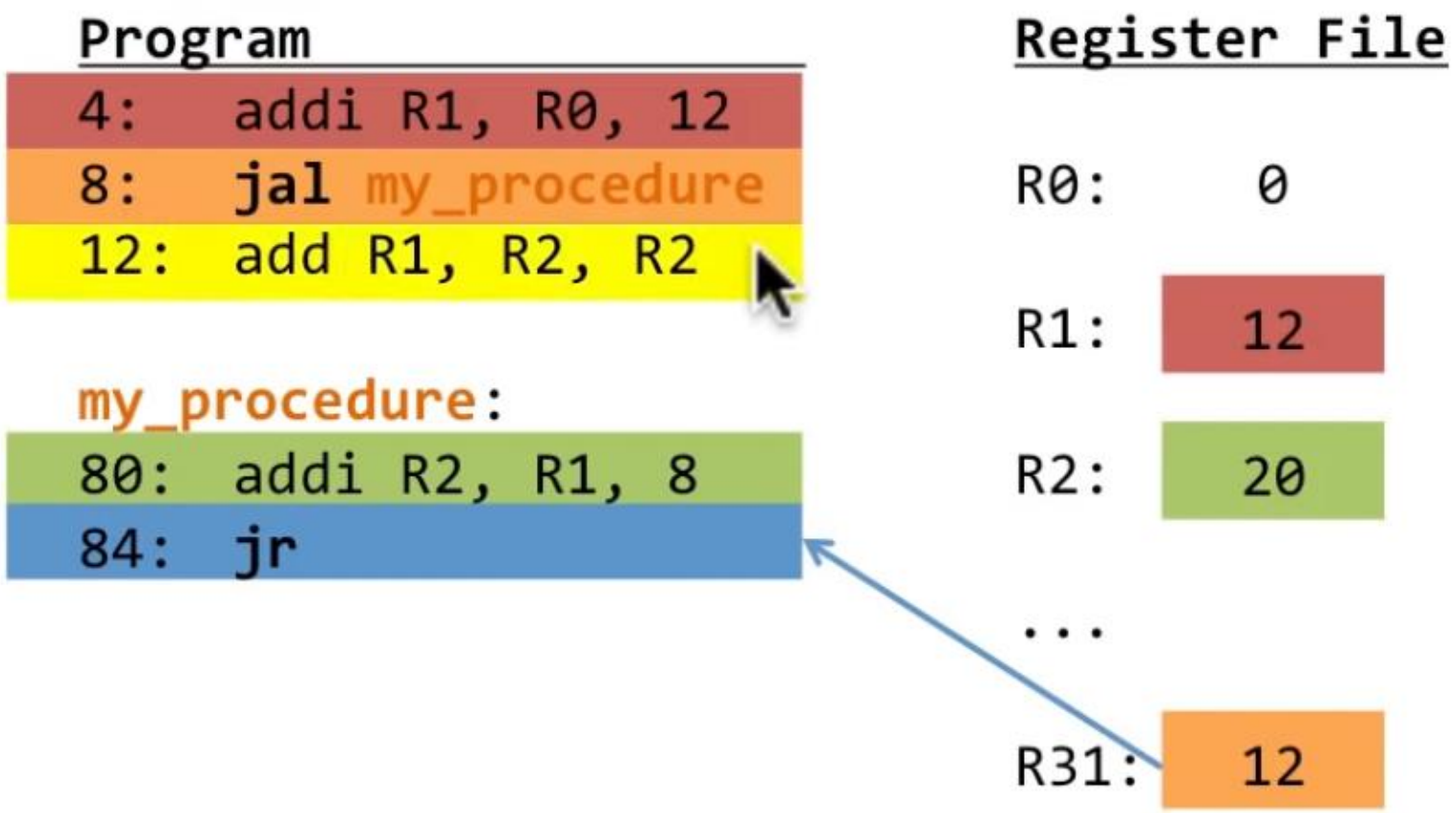
```
jal ProcedureAddress        ; jump-and-link to the procedure
```

  - Keeps track of the instruction *after* the **jal** so we can continue in the right place when we are done with the procedure.
  - **Stores the return address (PC+4) in $ra (R31)**

- **Return** control to the **caller** when the procedure is done:

```
jr $ra                      ; jump-return to the address in $ra
```

Jump-and-link



Question: jump-and-link

Q: Is the code below correct?

- No, you should only add 1 to $ra to go to the next instruction.
- No, jal already stores PC+4 into $ra so you don't need to add 4 yourself.
- Yes, you need to add 4 to $ra so you return to the next instruction (not the current one).

A: No, jal already stores PC+4 into $ra so you don't need to add 4 yourself.

The jal instruction stores the address of the current instruction plus 4 into $ra. (That is, PC+4.) Therefore you do not need to increment $ra.

Incrementing $ra would cause you to return to the instruction after the one you want, which would probably cause a crash.

```
jal Procedure          ; Call the procedure
. . .

Procedure:
 . . .                 ; Do the procedure's work
 addi $ra, $ra, 4      ; Make sure we return to the next instruction!
 jr $ra                ; Return
```

Procedure call in detail

```
main() {
 if (a == 0)
  b = update(g,h);
 else
  c = update(k,m);
}
```

Main Registers:
R20=a, R21=b, R22=c
R16=g, R17=h, R18=k, R19=m

```
main:
bne R20, R0, DoElse
 jal update (use R16, R17)
 Expect results in R21
 j SkipElse
DoElse:
 jal update (use R18, R19)
 Expect results in R22
SkipElse:
```

Q: What is going to happen to the variable "a" when main calls update?

- Nothing. update doesn't't use a.
- It will be changed because update is supposed to change "a".
- It will be changed because update uses the same register as "a" for a temporary register.

```
update(a1,a2) {
 return (a1+a2)-(a2<<4);
}
```

Update Registers:
Arguments: R4=a1, R5=a2
R20=temp0, R21=temp1, R2=result

```
update:
 add R20, R4, R5
 sll R21, R5, 4
 sub R2, R20, R21
jr $ra
```

A: It will be changed because update uses the same register as "a" for a temporary register

When we call update it will write to R16 as a temporary register. But main is using R16 to store "a". Update will corrupt main's data!

Problem: argument registers

```
main() {
 if (a == 0)
  b = update(g,h);
 else
  c = update(k,m);
}
```

Main Registers:
R20=a, R21=b, R22=c
R16=g, R17=h, R18=k, R19=m

```
update(a1,a2) {
 return (a1+a2)-(a2<<4);
}
```

Update Registers:
Arguments: R4=a1, R5=a2
R20=temp0, R21=temp1, R2=result

```
main:
bne R20, R0, DoElse
 jal update (use R16, R17)
 Expect results in R21
 j SkipElse
DoElse:
 jal update (use R18, R19)
 Expect results in R22
SkipElse:
```

```
update:
 add R20, R4, R5
 sll R21, R5, 4
 sub R2, R20, R21
jr $ra
```

But main is using R16, R17!

Problem: update expects its arguments in R4, R5

Problem: result registers

```
main() {
 if (a == 0)
  b = update(g,h);
 else
  c = update(k,m);
}
```

Main Registers:
R20=a, R21=b, R22=c
R16=g, R17=h, R18=k, R19=m

```
main:
bne R20, R0, DoElse
 jal update (use R16, R17)
 Expect results in R21
 j SkipElse
DoElse:
 jal update (use R18, R19)
 Expect results in R22
SkipElse:
```

But main expects the results in R21!

```
update(a1,a2) {
 return (a1+a2)-(a2<<4);
}
```

Update Registers:
Arguments: R4=a1, R5=a2
R20=temp0, R21=temp1, R2=result

```
update:
 add R20, R4, R5
 sll R21, R5, 4
 sub R2, R20, R21
jr $ra
```

Problem: update returns its result in R2

The real problem: lack of coordination

```
main() {
 if (a == 0)
  b = update(g,h);
 else
  c = update(k,m);
}
```

Main Registers:
R20=a, R21=b, R22=c
R16=g, R17=h, R18=k, R19=m

```
main:
bne R20, R0, DoElse
 jal update (use R16, R17)
 Expect results in R21
 j SkipElse
DoElse:
 jal update (use R18, R19)
 Expect results in R22
SkipElse:
```

```
update(a1,a2) {
 return (a1+a2)-(a2<<4);
}
```

Update Registers:
Arguments: R4=a1, R5=a2
R20=temp0, R21=temp1, R2=result

```
update:
 add R20, R4, R5
 sll R21, R5, 4
 sub R2, R20, R21
jr $ra
```

main() and update() are incompatible because they don't **coordinate** how they use the register file.

This is a real problem if you are calling someone else's code (e.g., a library).

Question: problems with lack of coordination

Q: Which of the following is *not* a reason we need a register convention for procedure calls?

- To know where the return value goes
- To know where the argument values go
- To know where the procedure is
- To avoid over-writing registers with temporary values

```
jal update
```

A: To know where the procedure is

The memory address of the procedure's instructions is included in the label constant when we call the procedure, e.g., "jal update". The program needs to know where it is, but the location of the code depends on the compiler.

The other resources (return value, arguments, and temporary registers) are all shared resources in the register file. As such we need to agree how to share them!

Saving and restoring registers

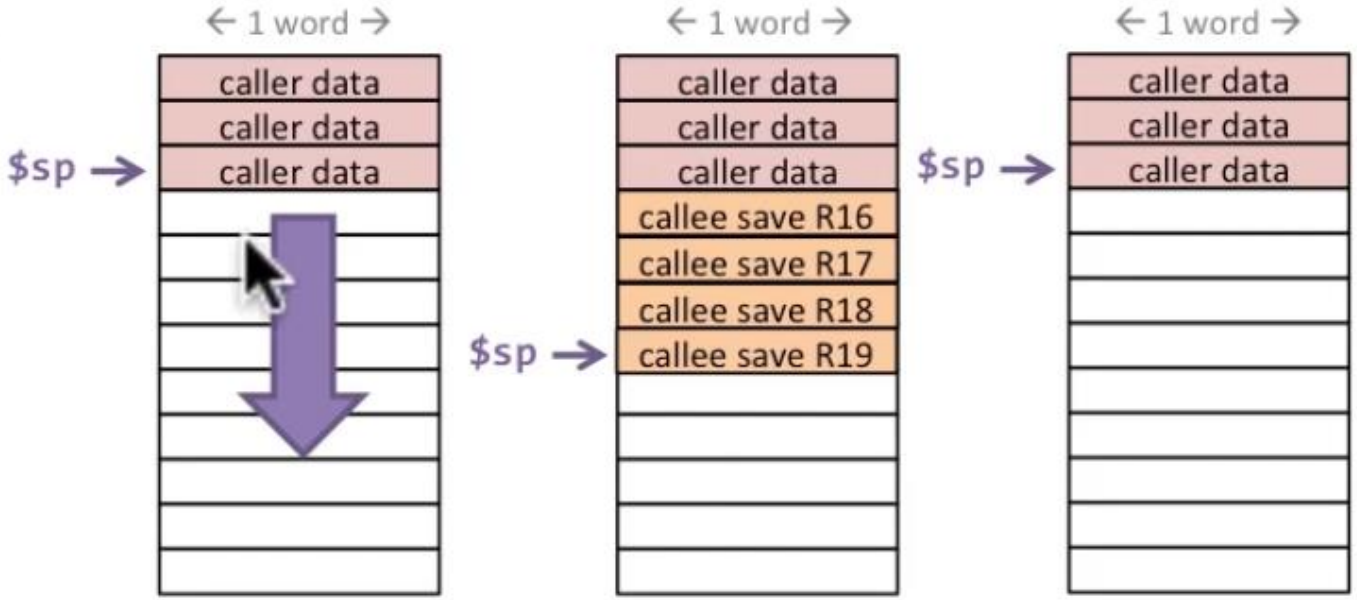
- We need a register **convention** so **callers** and **callees**:
  - Know where to expect data (**arguments** and **results**)
  - Don't **overwrite** each other's registers
    - Some registers must be **saved** by the **callee** if it uses them
    - Some registers must be **saved** by the **caller** if it uses them

- Saving and restoring:
  - Saving: *copy a register to memory* where they won't be overwritten
  - Restoring: *copy a register back from memory* to the original register

- Why do we need to save and restore?
  - The **callee** does not know which registers the **caller** is using!
    (It could have multiple different callers at different places in the code.)

Saving registers on the stack

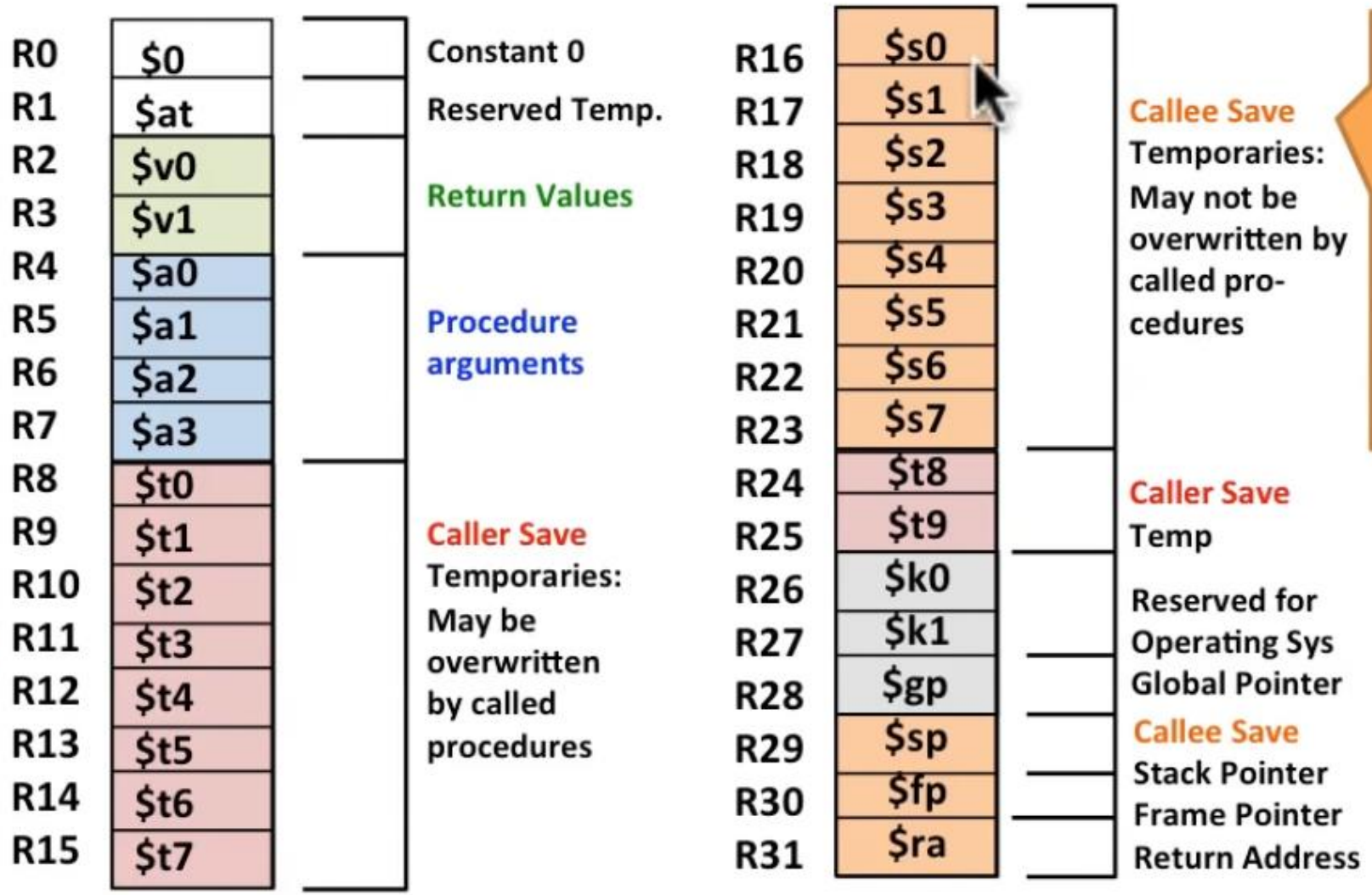
- MIPS has a special part of memory called The **Stack** for saving registers.
- The **Stack Pointer** (kept in $sp or R29) keeps the *address* of the end of the stack.
  In MIPS the stack grows **down**.



- **Move the stack pointer down** when storing more data on the stack.
- Each procedure **returns the stack** to where it was before it was called.
- Gives procedures a secure place to store data that does not fit in registers. (e.g., saved registers!)
- Each procedure manages its own stack space so they don't interfere.
- Works great as long as you return the stack to the way it was before.

During the procedure the registers that the procedure overwrites are stored safely on the stack.

Who saves what?



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

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

Question: how do we move the stack pointer?

Q: The stack pointer address is stored in R31 ($sp). How do much do we need to decrement it to make space for a register?

- 1
- 4
- 32
- Depends on the size of the register

A: 4

Each register is 4 bytes and each memory address is 1 byte. The Stack Pointer points to the last byte used on the stack, so to make room for a register (4 bytes) we need to move it down by 4 (4 bytes).

We do this by:
addi $sp, $sp, -4

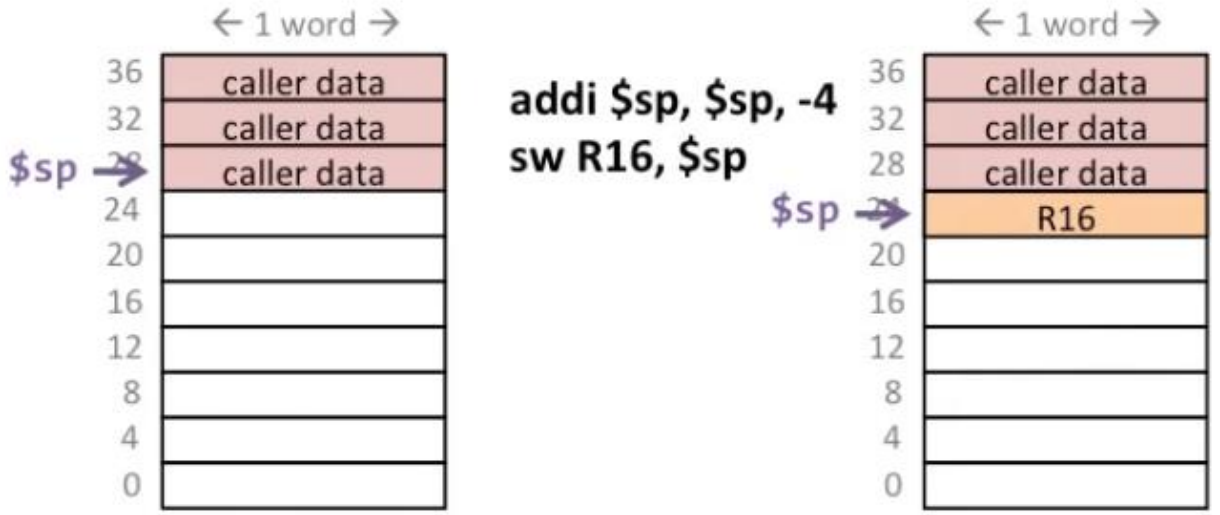
Question: how do we save to the stack?

Q: How do we store R16 to the stack?

- sw R16, $sp
- sw $sp, R16
- addi $sp, $sp, -4
  sw R16, $sp
- addi R16, $sp, -4
  sw R16, $sp

A: **addi $sp, $sp -4**
sw R16, $sp

First we move the stack pointer down by 1 word (4 bytes) then we store the register to the new address.



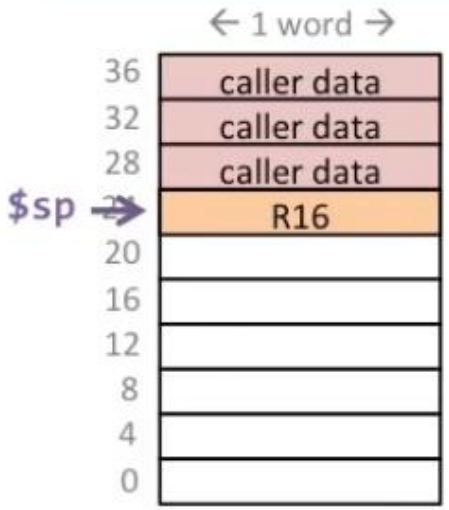
Question: how do we restore from the stack?

Q: How do we restore R16 from the stack?

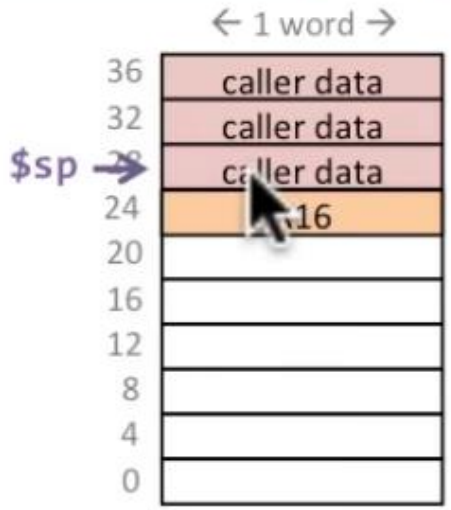
- lw R16, $sp
- lw $sp, R16
- addi $sp, $sp, 4
  lw R16, $sp
- lw R16, $sp
  addi $sp, $sp, 4

A: **lw R16, $sp**
addi $sp, $sp, 4

First we load the data from where the stack pointer currently points with load word. Then we increment the stack pointer to point to the item before since we have now restored this item.



lw R16, $sp
addi $sp, $sp, 4



Register File

R0:
R1:
…
R16 R16
R17:
…
R31:

Nested calls

main() saves some registers on the stack
main() calls B()

B() saves some registers on the stack

B() calls C()

C() saves some registers on the stack

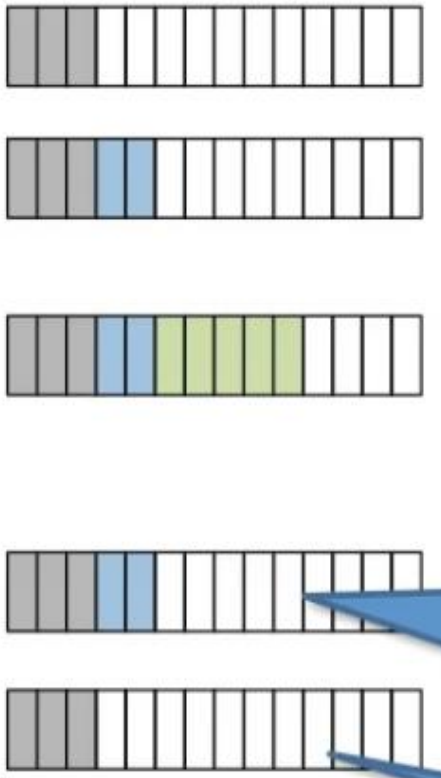
C() restores B()'s registers from the stack

C() returns

B() restores main()'s registers from the stack
B() returns

main() is back to where it was before!

```
main()
{
    B() jal B
    {
        C() jal C
        {
        } jr
    } jr
...
```



The stack **grows** and **shrinks** as procedure **calls add data** when they are called and **remove data** when they return.

Stack is returned to the way B had it before C was called.

Stack is returned to the way main had it before B was called.

- Some machines provide stacks as part of the architecture (VAX, JVM)
- Others (like MIPS) implement them in software

Example 1: which registers need to be saved?

Q: What registers need to be saved in this example?

- None
- The caller needs to save R16-R22
- The callee needs to save R20-R21
- The caller needs to save R16-R22 and the callee needs to save R20-R21

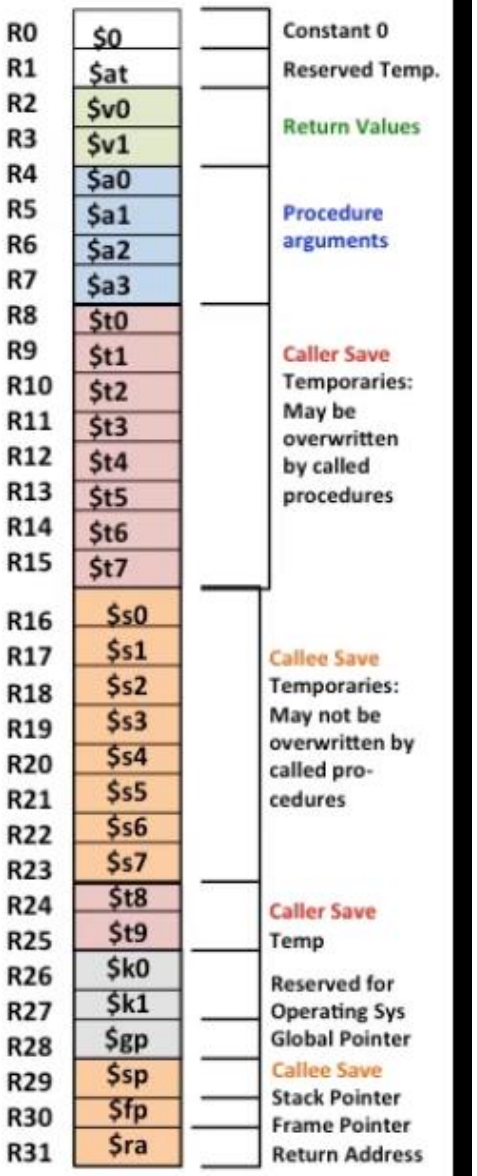
A: The callee needs to save R20-R21

R16-R23 are callee-saved registers. This means the callee must save them if it uses them, and update() does use both R20 and R21.

The caller, main(), does not need to save anything because it just uses callee-saved registers.

```
main:
bne R20, R0, DoElse
 add R4, R16, R0 ; move g→R4
 add R5, R17, R0 ; move h→R5
 jal update
 addi R21, R2    ; move result→b
 j SkipElse
DoElse:
 add R4, R18, R0 ; move k→R4
 add R5, R19, R0 ; move m→R5
 jal update
 addi R2[illegible] R2    ; move result→c
SkipElse:

update:
 add R20, R4, R5
 sll R21, R5, 4
 sub R2, R20, R21
jr $ra
```



Example 1: saving registers

```
main() {
 if (a == 0)
  b = update(g,h);
 else
  c = update(k,m);
}
```

Main Registers:
R20=a, R21=b, R22=c
R16=g, R17=h, R18=k, R19=m

```
update(a1,a2) {
 return (a1+a2)-(a2<<4);
}
```

Update Registers:
Arguments: R4=a1, R5=a2
R20=temp0, R21=temp1, R2=result

```
main:
bne R20, R0, DoElse
 add R4, R16, R0 ; move g→R4
 add R5, R17, R0 ; move h→R5
 jal update
 addi R21, R     ; move result→b
 j SkipElse
DoElse:
 add R4, R18, R0 ; move k→R4
 add R5, R19, R0 ; move m→R5
 jal update
 addi R22, R2    ; move result→c
SkipElse:
```

```
update:
 addi $sp, $sp -8
 sw  R20, 4($sp)
 sw  R21, 0($sp)
 add R20, R4, R5
 sll R21, R5, 4
 sub R2, R20, R21
 lw  R21, 0($sp)
 lw  R20, 4($sp)
 addi $sp, $sp, 8
jr $ra
```

3. Need to save them

1. Callee uses R20 and R21

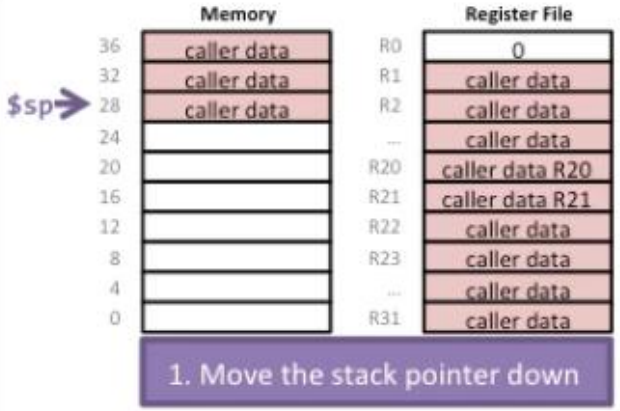
2. Which are callee-saved.

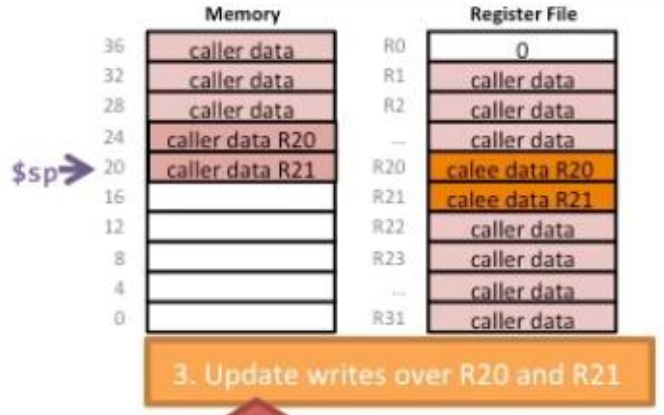
4. And restore them

5. So we don't corrupt the caller

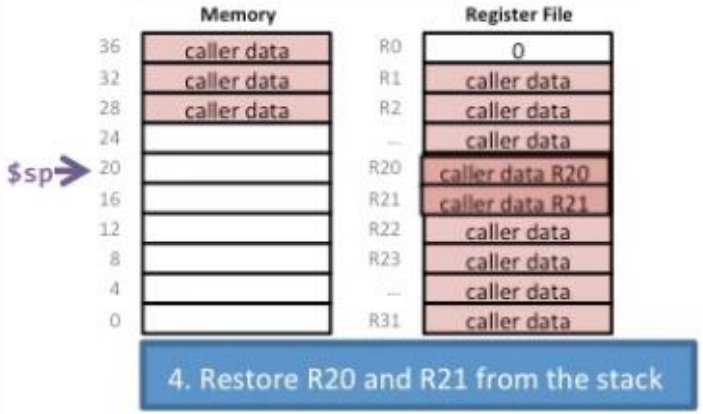
R0	$0	Constant 0
R1	$at	Reserved Temp.
R2	$v0	Return Values
R3	$v1	
R4	$a0	Procedure arguments
R5	$a1	
R6	$a2	
R7	$a3	
R8	$t0	Caller Save Temporaries: May be overwritten by called procedures
R9	$t1	
R10	$t2	
R11	$t3	
R12	$t4	
R13	$t5	
R14	$t6	
R15	$t7	
R16	$s0	Callee Save Temporaries: May not be overwritten by called pro-cedures
R17	$s1	
R18	$s2	
R19	$s3	
R20	$s4	
R21	$s5	
R22	$s6	
R23	$s7	
R24	$t8	Caller Save Temp
R25	$t9	
R26	$k0	Reserved for Operating Sys
R27	$k1	
R28	$gp	Global Pointer
R29	$sp	Callee Save Stack Pointer
R30	$fp	Frame Pointer
R31	$ra	Return Address

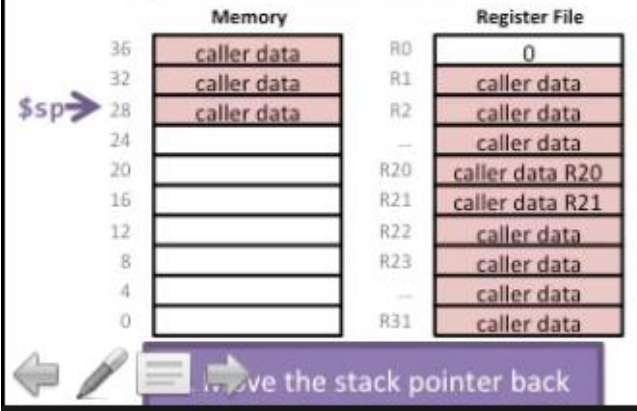
Callee-saving registers in detail





Good thing we saved the caller's R20 and R21 on the stack!






```
update:
 addi $sp, $sp -8
 sw  R20, 4($sp)
 sw  R21, 0($sp)
 add R20, R4, R5
 sll R21, R5, 4
 sub R2, R20, R21
 lw  R2,  0($sp)
 lw  R20, 4($sp)
 addi $sp, $sp, 8
jr $ra
```

1. Move the stack pointer down
2. Save R20 at $sp+4
2. Save R21 at $sp+0
3. Execute update
4. Restore R21 from $sp+0
4. Restore R20 from $sp+4
5. Move the stack pointer back

R0	$0	Constant 0
R1	$at	Reserved Temp.
R2	$v0	Return Values
R3	$v1	
R4	$a0	Procedure arguments
R5	$a1	
R6	$a2	
R7	$a3	
R8	$t0	Caller Save Temporaries: May be overwritten by called procedures
R9	$t1	
R10	$t2	
R11	$t3	
R12	$t4	
R13	$t5	
R14	$t6	
R15	$t7	
R16	$s0	Callee Save Temporaries: May not be overwritten by called pro-cedures
R17	$s1	
R18	$s2	
R19	$s3	
R20	$s4	
R21	$s5	
R22	$s6	
R23	$s7	
R24	$t8	Caller Save Temp
R25	$t9	
R26	$k0	Reserved for Operating Sys
R27	$k1	
R28	$gp	Global Pointer
R29	$sp	Callee Save Stack Pointer
R30	$fp	Frame Pointer
R31	$ra	Return Address

Example 2: which registers to save?

Caller

```
add $a0, $t0  2        ; set up the arguments
add $a1, $s0  $zero
add $a2, $s1  $t0
add $a3, $t0  3

jal update_func        ; call the update function procedure

add $t2, $v0, $zero    ; move the result into $t2
```

Callee

```
update_func:           ; calculates f=(g+h)-(i+j)
                       ; g, h, i, and j are in $a0, $a1, $

add $t0 $a0,$a1        ; g = $a0, h = $a1
add $t1 $a2,$a3        ; i = $a2, j = $a3
sub $s0 $t0,$t1

add $v0,$s0,$zero      ; return f in the result register $v0

jr  $ra                ; jump back to the calling routine
```

Caller needs to save anything not in the $s registers.

Questions:

1. What resources does the caller use?
   $t0, $s0, $s1
2. What resources does the callee use?
   $t0, $t1, $s0
3. What does the caller need to save?
   $t0
4. What does the callee need to save?
   $s0

Callee needs to save anything that is in the $s registers.

Example 2: saving to the stack

Move the stack pointer down by 4 bytes (1 word) **Then store the register to that location.**

Read the register from the stack. Then move the stack pointer up by 4 bytes (1 word) back to where it was before.

Caller

```
add $a0, $t0  2        ; set up the arguments
add $a1, $s0  $zero
add $a2, $s1  $t0
add $a3, $t0  3

addi $sp, $sp, -4      ; adjust the stack to make room for one item
sw  $t0, 0($sp)        ; save $t0 in case the callee uses it

jal update_func        ; call the update function procedure

lw  $t0, 0($sp)        ; restore $t0 from the stack
addi $sp, $sp, 4       ; adjust the stack to delete one item

add $t2, $v0, $zero    ; move the result into $t2
```

Callee

```
update_func:           ; calculates f=(g+h)-(i+j)
                       ; g, h, i, and j are in $a0, $a1, $a2, $a3

 addi $sp, $sp, -4     ; adjust the stack to make room for one item
 sw  $s0, 0($sp)       ; save $s0  for the caller

 add $t0 $a0,$a1       ; g = $a0, h = $a1
 add $t1 $a2,$a3       ; i = $a2, j = $a3
 sub $s0 $t0,$t1

 add $v0,$s0,$zero     ; return f in the result register $v0

 lw  $s0, 0($sp)       ; restore $s0 for the caller
 addi $sp, $sp, 4      ; adjust the stack to delete one item
 jr  $ra               ; jump back to the calling routine
```

Questions:

1. What resources does the caller use?
   $t0, $s0, $s1
2. What resources does the callee use?
   $t0, $t1, $s0
3. What does the caller need to save?
   $t0
4. What does the callee need to save?
   $s0

Example 2: what not to save

Caller

```
add $a0, $t0, 2        ; set up the arguments
add $a1, $s0, $zero
add $a2, $s1, $t0
add $a3, $t0, 3

addi $sp, $sp, -4      ; adjust the stack to make room for one item
sw  $t0, 0($sp)        ; save $t0 in case the callee uses it

jal update_func        ; call the update function procedure

lw  $t0, 0($sp)        ; restore $t0 from the stack
addi $sp, $sp, 4       ; adjust the stack to delete one item

add $t2, $v0, $zero    ; move the result into $t2
```

Callee

```
update_func:           ; calculates f=(g+h)-(i+j)
                       ; g, h, i, and j are in $a0, $a1, $a2, $a3

 addi $sp, $sp, -4     ; adjust the stack to make room for one item
 sw  $s0, 0($sp)       ; save $s0  for the caller

 add $t0,$a0,$a1       ; g = $a0, h = $a1
 add $t1,$a2,$a3       ; i = $a2, j = $a3
 sub $s0,$t0,$t1

 add $v0,$s0,$zero     ; return f in the result register $v0

 lw  $s0, 0($sp)       ; restore $s0 for the caller
 addi $sp, $sp, 4      ; adjust the stack to delete one item
 jr  $ra               ; jump back to the calling routine
```

Q: Why does the callee not save $s1?

- It does
- The callee only saves $t registers
- It does not write to $s1, so it doesn't need to save it
- The caller saves it

A: It does not write to $s1, so it doesn't need to save it

The callee never writes to $s1 so it won't change its value. Therefore it does not need to save it.

Q: Why does the caller not save $t2?

- It does
- The caller only saves $s registers
- The caller writes over $t2 so it doesn't matter what the callee does to it
- The callee saves it

A: The caller writes over $t2 so it doesn't matter what the callee does to it

The caller copies the result value ($v0) into $t2. So even if the callee writes something into $t2 it won't matter because the caller writes over it again.

Question: $s and $t registers

Q: Who has to save the $s and $t registers?

- The callee must save all $s registers and the caller must save all $t registers
- The callee must save all $t registers and the caller must save all $s registers
- The callee must save all $s registers it uses and the caller must save all $t registers it uses
- The callee must save all $t registers it uses and the caller must save all $s registers it uses

A: The callee must save all $s registers it uses and the caller must save all $t registers it uses

The $s (save) registers are "saved" by the callee. But the callee only needs to save them if it uses them. (Otherwise it wouldn't change them.)

The $t (temporary) registers must be saved by the caller, but only if it uses them.

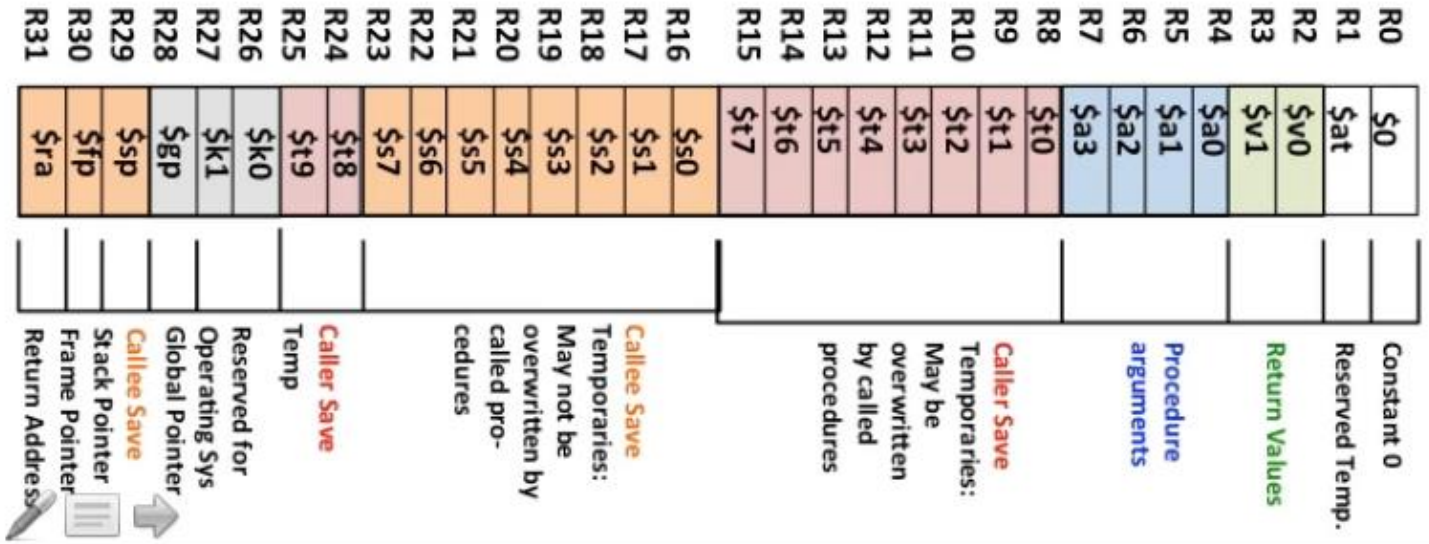
Question: saving to the stack

Q: If the caller uses $s0, $s1, $t0, and $t1, how much stack space does it need to use for saving registers before making a procedure call?

- 32 bytes
- 8 bytes
- 4 bytes
- 2 bytes

A: 8 bytes

The caller must save any registers it uses that are not in the $s (callee "save") registers. Therefore it must save $t0 and $t1 before making the procedure ca[illegible] which is 2x4 bytes, or 8 bytes.



Question: moving the stack pointer

Q: What values should X and Y have in the code below to make the code correct for saving registers $so and $s4?

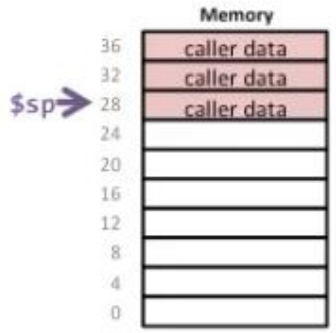
- X=8, Y=4
- X=-4, Y=4
- X= -8, Y=4
- X=4, Y=0

A: X=-8, Y=4

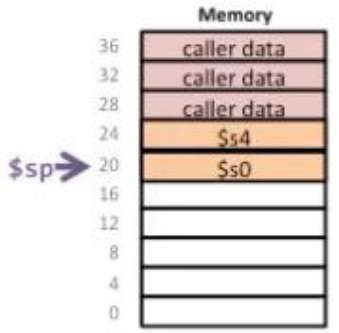
We need to store two registers, so we need to move the stack pointer down two words, or 2x4 bytes = 8 bytes. (add -8 to the stack pointer)

We then store the first register $s0 at the stack pointer, and we want to store the other one 4 above the stack pointer, since we moved it down 8.

```
addi $sp, $sp, X
sw   $s0, 0($sp)
sw   $s4, Y($sp)
```




```
addi $sp, $sp, -8
sw   $s0, 0($sp)
sw   $s4, 4($sp)
```



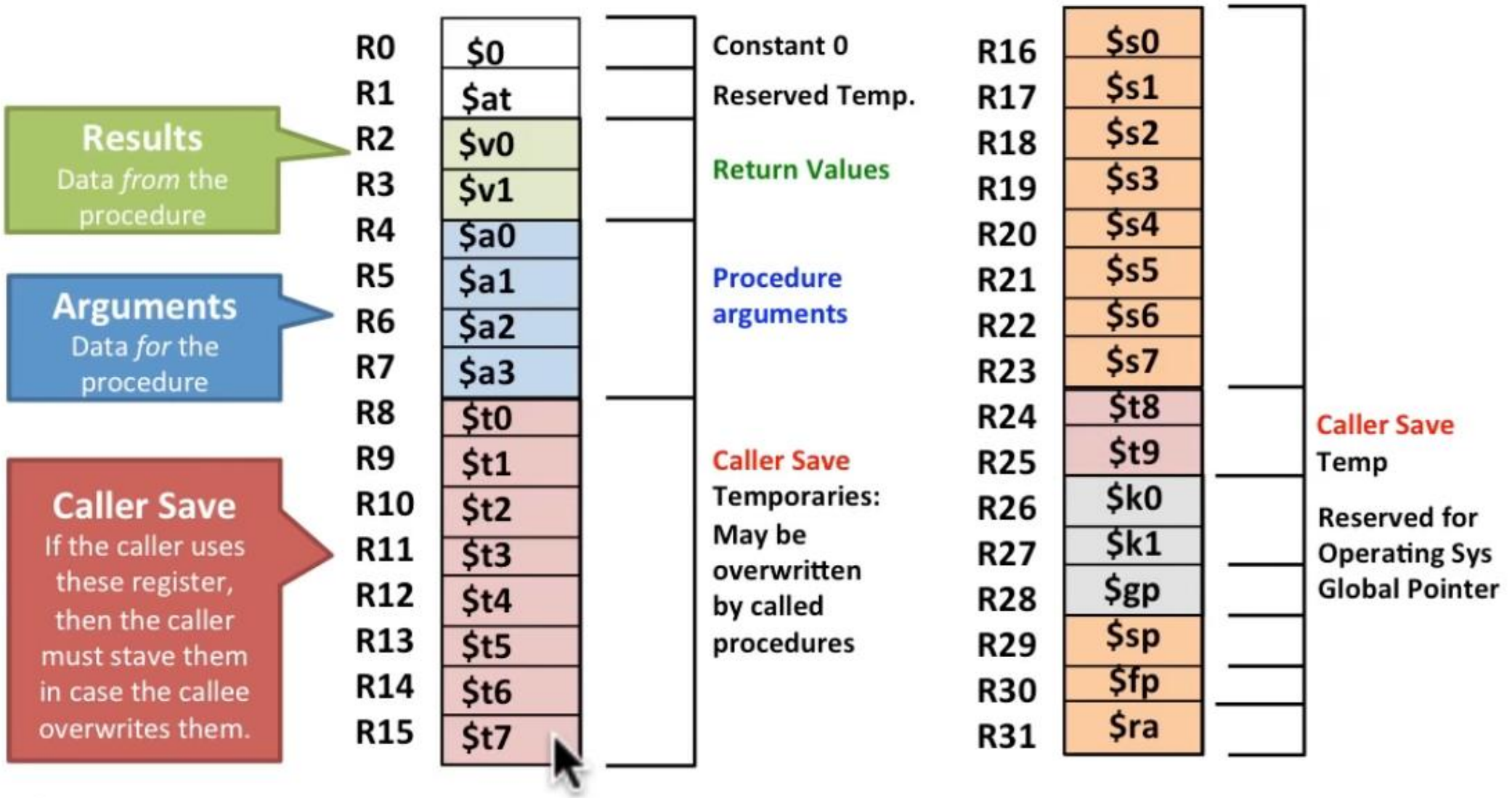
Summary: procedure calls

- Procedures need to:
  - Know where to find their **arguments** and put their **results**
  - **Save** and **restore** registers to avoid overwriting the **caller's** registers
  - Return to the right place when done

- To accomplish this we:
  - Have **conventions** for who saves registers
  - Save them in memory on the **stack**
  - Use `jal` and **jr $ra** to enter and exit procedures

- As long as everyone follows the convention we get interoperability

MIPS register names and conventions



sub $t0, $s4, $s5

Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	0	22_{hex}

opcode = 0 (hex) = 000000 (binary)

funct= 22 (hex) = 100010 (binary)

NOTE: All "R-type" instructions have an Opcode of "0"

RECAP:

- opcode = 0 (hex) = 000000 (binary)
- $rs = $s4 = 20 (decimal) = 10100 (binary)
- $rt = $s5 = 21 (decimal) = 10101 (binary)
- $rd = $t0 = 8 (decimal) = 01000 (binary)
- shamt = Not Used = 00000 (binary)
- funct = 22 (hex) = 100010 (binary)

```
0x0040310c      loop: ...
...                   ...
0x00405000            beq $t5, $s0, loop
```

Compute Byte Offset: (0x00405000 + 4) – (0x0040310c) = 0x1EF8

Compute Word Offset: 0x1EF8 / 4 = 0x7BE = 111 1011 1110 (binary)

Extend to 16 bits: 111 1011 1110 (binary) = 0000 0111 1011 1110 (16-bit binary)

Compute 2's complement:

- Complement all bits: 1111 1000 0100 0001
- Add 1: 1111 1000 0100 0010

beq $t5, $s0, loop

RECAP:

- opcode = 4 (hex) = 000100 (binary)
- $rs =$t5 = 13 (decimal) = 01101 (binary)
- $rt =$s0 = 16 (decimal) = 10000 (binary)
- immediate = Branch Offset = 1111 1000 0100 0010

0x10001A08: 000011 00000 10000 01000 00000 000111

Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	jr	R	PC=R[rs]		$0 / 08_{hex}$

So, we know this is a Jump And Link instruction.
All Jump instructions have the following format: