CSE261: Computer Architecture

4. Instruction Set Architecture (3)

Seongil Wi

Department of Computer Science and Engineering

Notification: Quiz1

- Date: 10/08 (TUE.), Class time

- Scope:
  - Instruction Set Architecture – ISA Overview
  - Instruction Set Architecture – MIPS ISA

- T/F problems + 2~4 computation problems

Notification: HW1

- HW1 will be released (in this week)

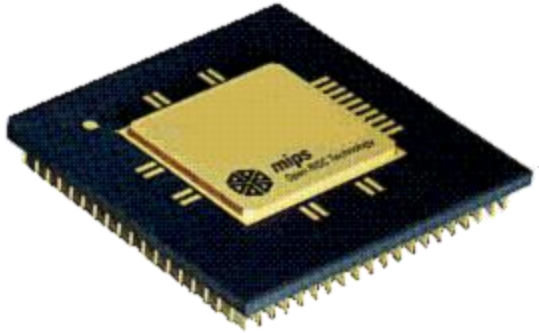
- Due date: 10/10, 11:59PM

- MIPS assembly programming
- Environment: any OS, we will use a MIPS simulator QtSpim 9.1.24 (https://spimsimulator.sourceforge.net/)

Recap: Instruction Set

- The commands understood by a given architecture

Today's topic



MIPS's Instructions

```
slt $t0, $s0, $s1
add $s2, $s0, $s1
sub $t2, $s1, $zero
lw  $t0, 8($s3)
```

Recap: MIPS General Purpose Registers

#	Name	Usage
0	$zero	The constant value 0
1	$at	Assembler temporary
2	$v0	Values for results and expression evaluation
3	$v1	
4	$a0	Arguments
5	$a1	
6	$a2	
7	$a3	
8	$t0	Temporaries (Caller-save registers)
9	$t1	
10	$t2	
11	$t3	
12	$t4	
13	$t5	
14	$t6	
15	$t7	

#	Name	Usage
16	$s0	Saved temporaries (Callee-save registers)
17	$s1	
18	$s2	
19	$s3	
20	$s4	
21	$s5	
22	$s6	
23	$s7	
24	$t8	More temporaries (Caller-save registers)
25	$t9	
26	$k0	Reserved for OS kernel
27	$k1	
28	$gp	Global pointer
29	$sp	Stack pointer
30	$fp	Frame pointer
31	$ra	Return address

Recap: MIPS Design Principles

1. Simplicity favors regularity

2. Smaller is faster

3. Make common case fast

4. Good design demands a compromise

Principle #1: Simplicity Favors Regularity

- Most of arithmetic/logic instructions have **three operands**
  - Order is fixed (destination first)

```
add   a, b, c   // a = b + c
sub   a, b, c   // a = b - c
```

One destination

Two sources

- *Design Principle 1:* Simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Principle #2: Smaller is Faster

- MIPS provides **only 32 registers** available to programmers

- Most of the operands of MIPS arithmetic/logic instructions are restricted to "registers" (*register addressing mode*)
  - E.g., `int a = b + c` → `add $s0,$s1,$s2`
  - Compiler associates the variables with the registers

Design Principle 2: Smaller is Faster

Principle #3: Make Common Case Fast

- **Observation**: constants are used quite frequently as operands

```
i ++
```

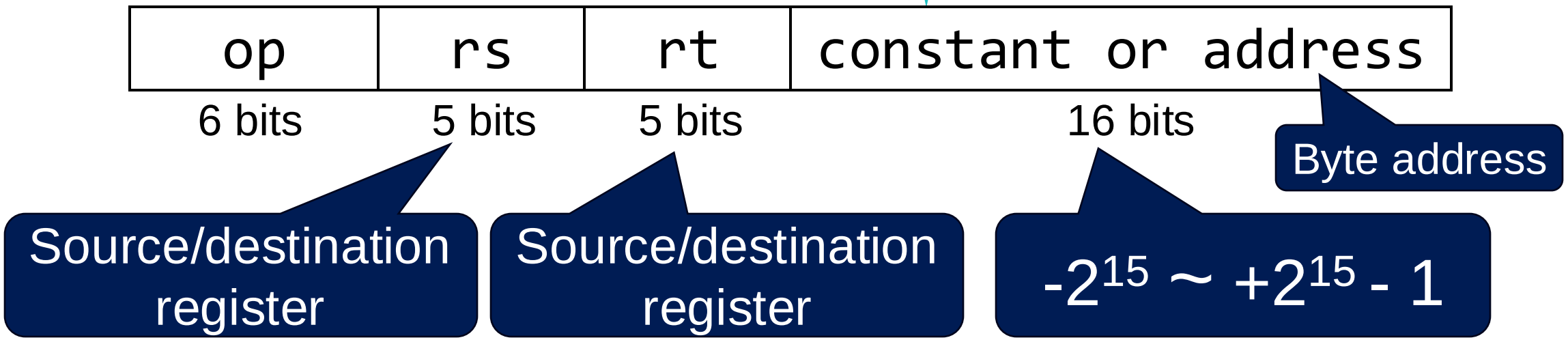
```
a = b + 3
```

- **Solution**: make *constants part* of arithmetic instructions
  - E.g., `addi $s3, $s3, 4`

(Loading a constant from memory into a register can slow down the speed)

- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

Principle #4: Good Design demands Good Compromises



- *Design Principle 4:* Good design demands good compromises
  - Different formats complicate decoding, but <u>keep 32-bit instructions same length (principle 1)</u>
  - Keep formats as similar as possible

Let’s deep dive into MIPS instruction set

MIPS Instruction Set

1. Arithmetic/Logic

2. Data Transfer

3. Control

1. Arithmetic/Logical Instructions

- **Arithmetic instructions**
  - `add $s0, $s1, $s2` # $s0 ⟵ $s1 + $s2
  - `addi $s0, $s1, 100` # $s0 ⟵ $s1 + 100
  - `sub $s0, $s1, $s2` # $s0 ⟵ $s1 - $s2
  - `subi?`
    - Just use a negative constant (`addi $s2, $s1, -1`)
  - `mult $s1, $s2` # $hi, $lo ⟵ $s1 x $s2
  - `div $s1, $s2` # $hi, $lo ⟵ $s1 / $s2
  - `mfhi $s1` # $s1 ⟵ $hi
  - `mflo $s1` # $s1 ⟵ $lo

1. Arithmetic/Logical Instructions

- **Logical instructions**
  - `and $s0, $s1, $s2 # $s0⟵$s1 bitwise-AND $s2`

Truth table for and

Input A	Input B	Output
0	0	0
1	0	0
0	1	0
1	1	1

```
$s1 0000 0000 0000 0000 0011 1100 0000 0000
$s2 0000 0000 0000 0000 0000 1101 1100 0000
$s0 0000 0000 0000 0000 0000 1100 0000 0000
```

1. Arithmetic/Logical Instructions

- **Logical instructions**
  - `and $s0, $s1, $s2 # $s0⟵$s1 bitwise-AND $s2`
  - `or $s0, $s1, $s2  # $s0⟵$s1 bitwise-OR $s2`

Truth table for `or`

Input A	Input B	Output
0	0	0
1	0	1
0	1	1
1	1	1

```
$s1 0000 0000 0000 0000 0011 1100 0000 0000
$s2 0000 0000 0000 0000 0000 1101 1100 0000
-------------------------------------------
$s0 0000 0000 0000 0000 0011 1101 1100 0000
```

1. Arithmetic/Logical Instructions

- **Logical instructions**
  - `and $s0, $s1, $s2 # $s0⟵$s1 bitwise-AND $s2`
  - `or $s0, $s1, $s2  # $s0⟵$s1 bitwise-OR $s2`
  - `andi $s0, $s1, 31 / ori $s0, $s1, 32`
  - `nor $s0, $s1, $s2 # $s0⟵$s1 bitwise-NOR $s2`

Truth table for nor

Input A	Input B	Output
0	0	1
1	0	0
0	1	0
1	1	0

```
$s1  0000 0000 0000 0000 0011 1100 0000 0000
$s2  0000 0000 0000 0000 0000 1101 1100 0000
-----------------------------------------------
or   0000 0000 0000 0000 0011 1101 1100 0000
nor  1111 1111 1111 1111 1100 0010 0011 1111
($s0)
```

1. Arithmetic/Logical Instructions

- **Logical instructions**
  - `and $s0, $s1, $s2 # $s0⟵$s1 bitwise-AND $s2`
  - `or $s0, $s1, $s2  # $s0⟵$s1 bitwise-OR $s2`
  - `andi $s0, $s1, 31 / ori $s0, $s1, 32`
  - `nor $s0, $s1, $s2 # $s0⟵$s1 bitwise-NOR $s2`

Truth table for nor

Input A	Input B	Output
0	0	1
1	0	0
0	1	0
1	1	0

Q. How to implement **bit-wise NOT operation** with above instructions?

```
nor $s1, $s2, $zero
```

1. Arithmetic/Logical Instructions

- **Logical instructions**
  - `and $s0, $s1, $s2 # $s0⟵$s1 bitwise-AND $s2`
  - `or $s0, $s1, $s2  # $s0⟵$s1 bitwise-OR $s2`
  - `andi $s0, $s1, 31 / ori $s0, $s1, 32`
  - `nor $s0, $s1, $s2 # $s0⟵$s1 bitwise-NOR $s2`

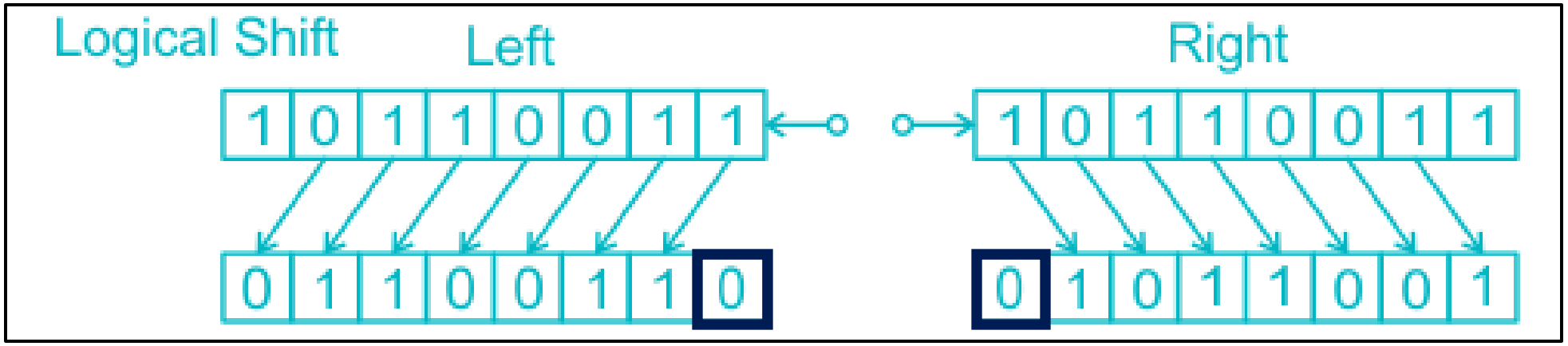
  - `sll $s0, $s1, 2  # $s0⟵$s1 << 2`
    `(shift left logical)`
  - `srl $s0, $s1, 3  # $s0⟵$s1 >> 3`
    `(shift right logical)`

sll and srl

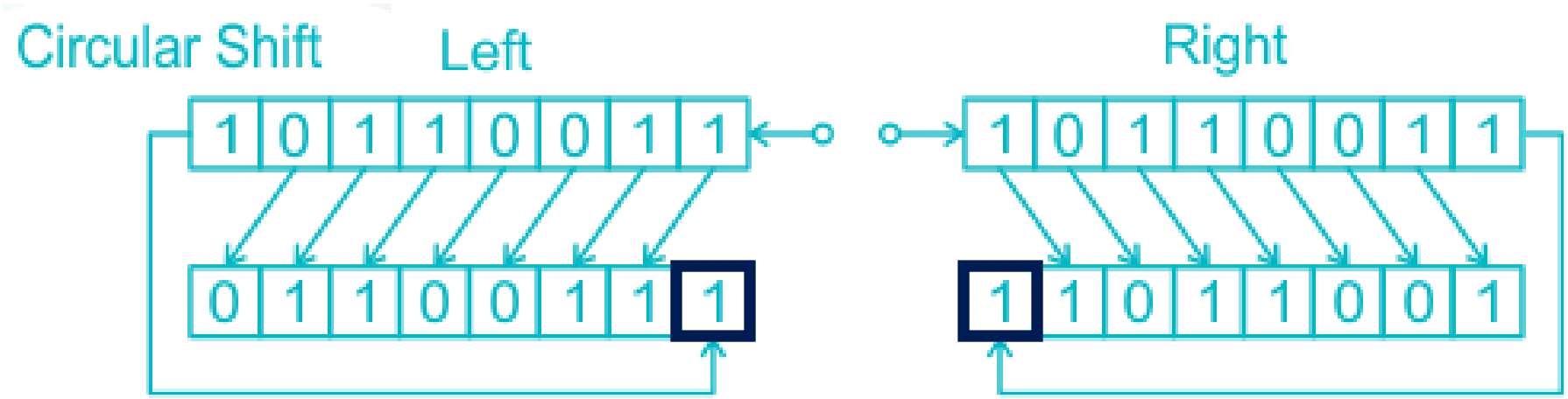
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

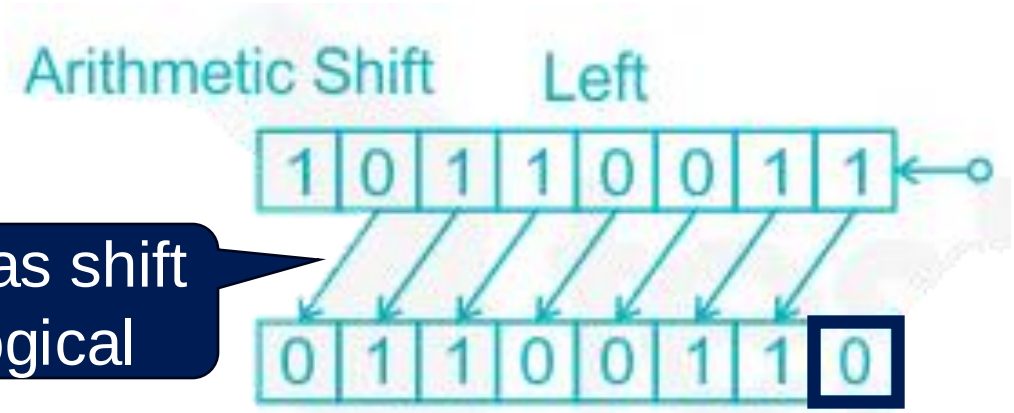
How many positions to shift

- **Shift left logical (`sll`)**
  - Shift left and fill with 0 bits
- **Shift right logical (`srl`)**
  - Shift right and fill with 0 bits



FYI: Circular Shift and Arithmetic Shift





Same as shift left logical

$00001011_2 = 11_{10}$

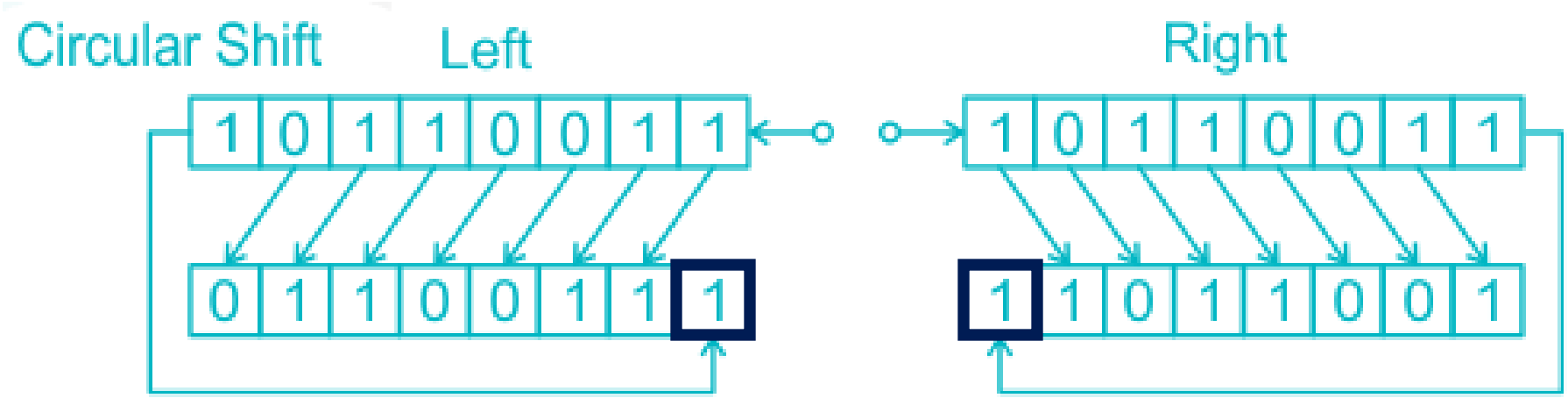
Arithmetic shift left by 1 bit

$00010110_2 = 22_{10}$

- Shift left by 1 bit → multiply by 2
- Shift left by 2 bits → multiply by 2^2
- …

Image from https://testbook.com/question-answer/which-of-the-following-isare-the-types-of-shifts--5f9c078f4112f193f7d991c2

FYI: Circular Shift and Arithmetic Shift



Same as shift left logical

Preserve the sign bit

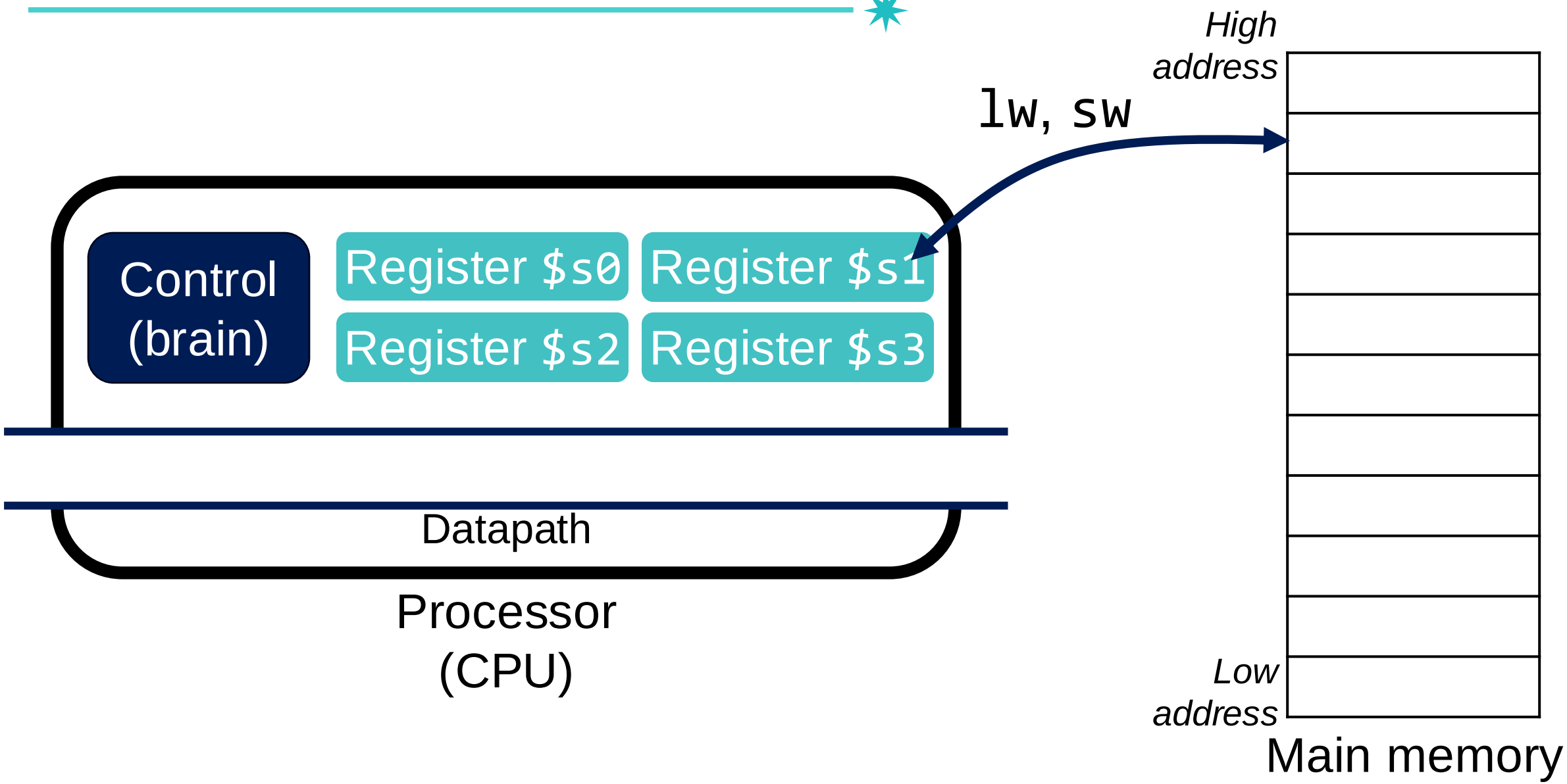
MIPS `sra` instruction

- Shift left by 1 bit → multiply by 2
- Shift left by 2 bits → multiply by 2^2
- …

- Shift right by 1 bit → divide by 2
- Shift right by 2 bits → divide by 2^2
- …

Image from https://testbook.com/question-answer/which-of-the-following-isare-the-types-of-shifts--5f9c078f4112f193f7d991c2

2. Data Transfer Instructions



2. Data Transfer Instructions

- I-format instructions

- **Read data from memory (load)**
  - `lw $s1, 8($s0)  # $s1 ⟵ Memory[$s0 + 8]`

- **Write data to memory (store)**
  - `sw $s1, 12($s0)  # Memory[$s0 + 12] ⟵ $s1`

Example of the Data Transfer Instructions

- C code:

```
A[300] = h + A[300];
        // 4-byte array A[]
        // h in $s2
        // base address of A in $t1
```

Exercise: convert the C program into MIPS assembly language

Example of the Data Transfer Instructions

- C code:

```
A[300] = h + A[300];
        // 4-byte array A[]
        // h in $s2
        // base address of A in $t1
```

- Compiled MIPS code:

```
lw  $t0, 1200($t1)  # 1200 = 300 * 4 (= word size)
add $t0, $s2, $t0
sw  $t0, 1200($t1)
```

Example of the Data Transfer Instructions

- Compiled MIPS code:

```
lw  $t0, 1200($t1)  # 1200 = 300 * 4 (= word size)
add $t0, $s2, $t0
sw  $t0, 1200($t1)
```

R-format	op	rs	rt	rd	shamt	funct
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
I-format	op	rs	rt	constant or address		
	6 bits	5 bits	5 bits	16 bits		
I-format	35	9	8	1200		
R-format	0	18	8	8	0	32
I-format	43	9	8	1200		

3. Control Instructions

- MIPS has **conditional** and **unconditional** branch instructions
  - **Conditional branch**: `beq, bne`
  - **Unconditional branch**: `j, jr, jal`

Conditional Branch Instructions

- I-format instructions

- **B**ranch if **Eq**ual

```
beq $s1, $s2, L1
 # if($s1 == $s2), pc = target address labeled L1
```

op=4	rs=17	rt=18	address=25

Recap: PC-Relative Mode Example

Assembly language → **Machine language**

```
        slt $t0, $s0, $s1                  10101010 00110010 01000000 00101010
        beq $t0, $zero, else #if $s0≥$s1   00010001 00000000 00000000 00000010
        add $s2, $s0, $s1                     ...      ...      ...      ...
        j exit                                ...      ...      ...      ...
else:   sub $s2, $s0, $s1     #else        00000100    ...
        ...                                   ...
exit:
```

There is no specific machine code for labels

Recap: PC-Relative Mode Example

Assembly language → **Machine language**

```
        slt $t0, $s0, $s1
        beq $t0, $zero, else  #if $s0≥$s1
        add $s2, $s0, $s1
        j exit
else:   sub $s2, $s0, $s1     #else
        ...
exit:
```

2 (relative offset)

```
10101010 00110010 01000000 00101010
00010001 00000000 00000000 00000010
   ...      ...       ..       ...
   ...      ..                   .
00000100    ..
   ...
```

Address field (why 2?)

Conditional Branch Instructions

- I-format instructions

- **B**ranch if **Eq**ual

```
beq $s1, $s2, L1
 # if($s1 == $s2), pc = target address labeled L1
```

op=4	rs=17	rt=18	address=25

- **B**ranch if **N**ot **E**qual

```
bne $s1, $s2, L1
 # if($s1 != $s2), pc = target address labeled L1
```

op=5	rs=17	rt=18	address=25

Conditional Branch Instructions

- We have `beq` and `bne`
- What about branch-if-less-than?

Combine `slt` and `beq/bne`

Set Less Than (slt)

- Set Less Than (R-format)

```
slt $s1, $s2, $s3   # if($s2 < $s3) $s1 = 1 else $s1 = 0
```

op=0	rs=18	rt=19	rd=17	shamt=0	funct=42

- Set Less Than immediate (I-format)

```
slti $s1, $s2, 100  # if($s2 < 100) $s1 = 1 else $s1 = 0
```

op=10	rs=17	rt=18	address=100

Branching Less Than

```
if x ≥ y then
  z = x + y
else:
  z = x - y
```

```
$s0: x
$s1: y
$s2: z
```

Assembly?

Branching Less Than

```
if x ≥ y then
  z = x + y
else:
  z = x - y
```

```
$s0: x
$s1: y
$s2: z
```

```
slt $t0, $s0, $s1
bne $t0, $zero, else
add $s2, $s0, $s1
j exit
else: sub $s2, $s0, $s1
exit:
```

Unconditional branch

Branching Less Than

```
if x ≥ y then
  z = x + y
else:
  z = x - y
```

```
$s0: x
$s1: y
$s2: z
```

```
slt $t0, $s0, $s1
bne $t0, $zero, else
add $s2, $s0, $s1
j exit
else: sub $s2, $s0, $s1
exit:
```

Assembler calculates addresses

Unconditional branch

Recap: Addressing Mode: PC-Relative Mode

The content of PC is added to the address part of instruction to obtain the *effective address* (branch type instructions)

- *Effective address*: PC + address field of instruction*4
- Operand value: memory[effective address]

Example: MIPS instruction

```
beq $t0,$zero,else
```

Effective address = 200 + Address field value × 4

Register PC

Recap: Why not `bge`, `bgt`, `ble`, `blt`?

- Hardware for <, ≥, … slower than =, ≠
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions penalized!

- `beq` and `bne` are the common cases!
- This is a good design compromise

Recap: RISC vs. CISC

- **Reduced Instruction Set Computer (RISC)**
  - Example: MIPS, ARM, PowerPC
  - Small and simple instruction set **=> Simple hardware**
  - Fixed-size instruction format

Example instruction set:

`slt, beq(=), bne(≠)`

- **Reduced Instruction Set Computer (CISC)**
  - Example: Intel x86, AMD
  - A large number of instruction set **=> Complex hardware**
  - Variable-size instruction format

Example instruction set:

`slt, beq(=), bne(≠)`
`bge(≥), bgt(>), ble(≤), blt(<)`

Compiling Loop Statements

- **C code:**

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

  - i in $s3
  - k in $s5
  - Address of save[] in $s6
  - 4-byte array save[]

- **Compiled MIPS code:**

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

Compiling Loop Statements

- **C code:**

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

  - i in $s3
  - k in $s5
  - Address of save[] in $s6
  - 4-byte array save[]

- **Compiled MIPS code:**

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

$t1 ← i * 4 (word size)

$t1 ← addr(save) + $t1

$t0 = save[i]

Signed vs. Unsigned

- **Signed comparison:** `slt, slti`
- **Unsigned comparison:** `sltu, sltui`

- Example

```
$s0 = 1111 1111 1111 1111 1111 1111 1111 1111
$s1 = 0000 0000 0000 0000 0000 0000 0000 0001
-slt  $t0, $s0, $s1  # signed
```

  - `-1 < +1 ⇒ $t0 = 1`

```
-sltu $t0, $s0, $s1  # unsigned
```

  - `+4,294,967,295 > +1 ⇒ $t0 = 0`

Unconditional Branch Instructions

- J-format instructions

- **J**ump

```
j L1 # pc = target address labeled L1
```

J-Format Instructions

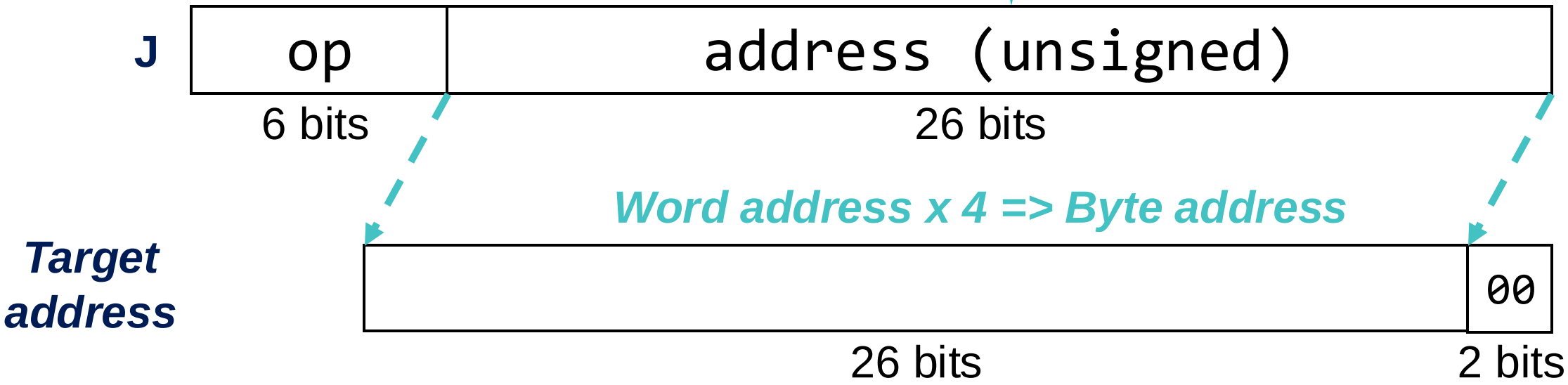
op	address (unsigned)
6 bits	26 bits

Operation code

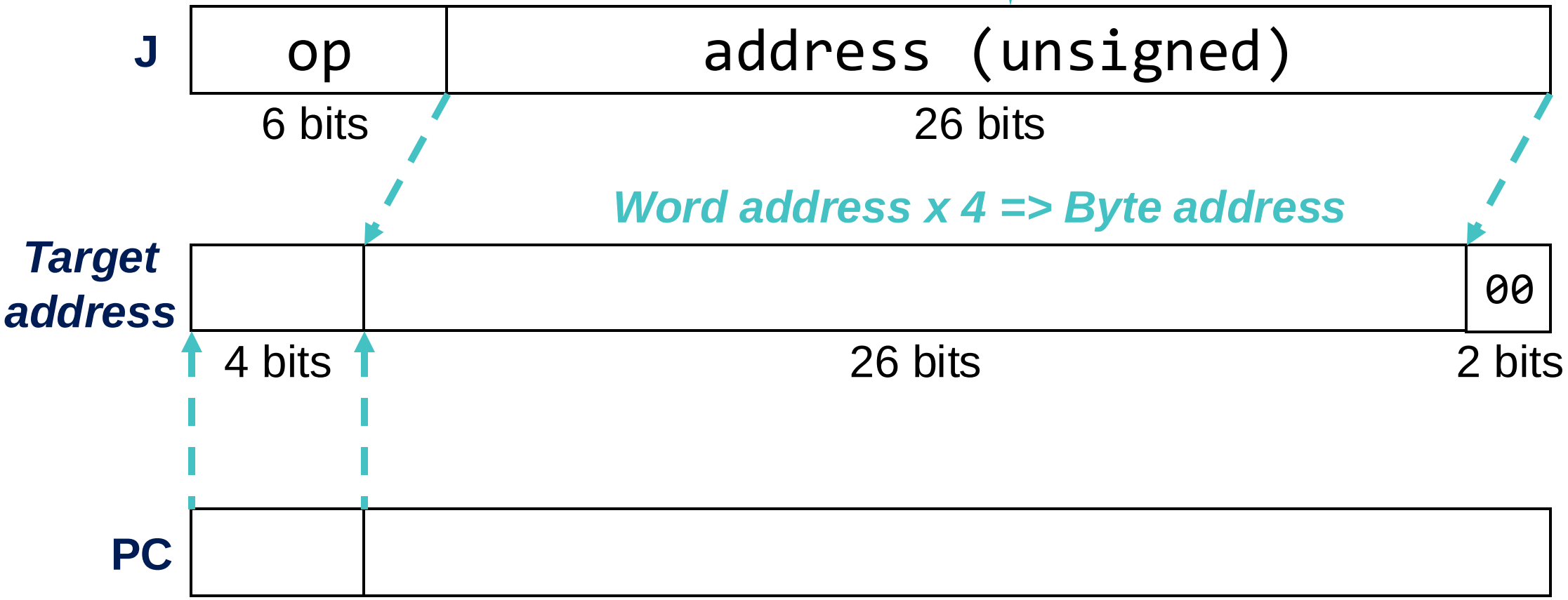
Word address

It has a 26-bit long address.
How to get the effective address from J-format instructions?

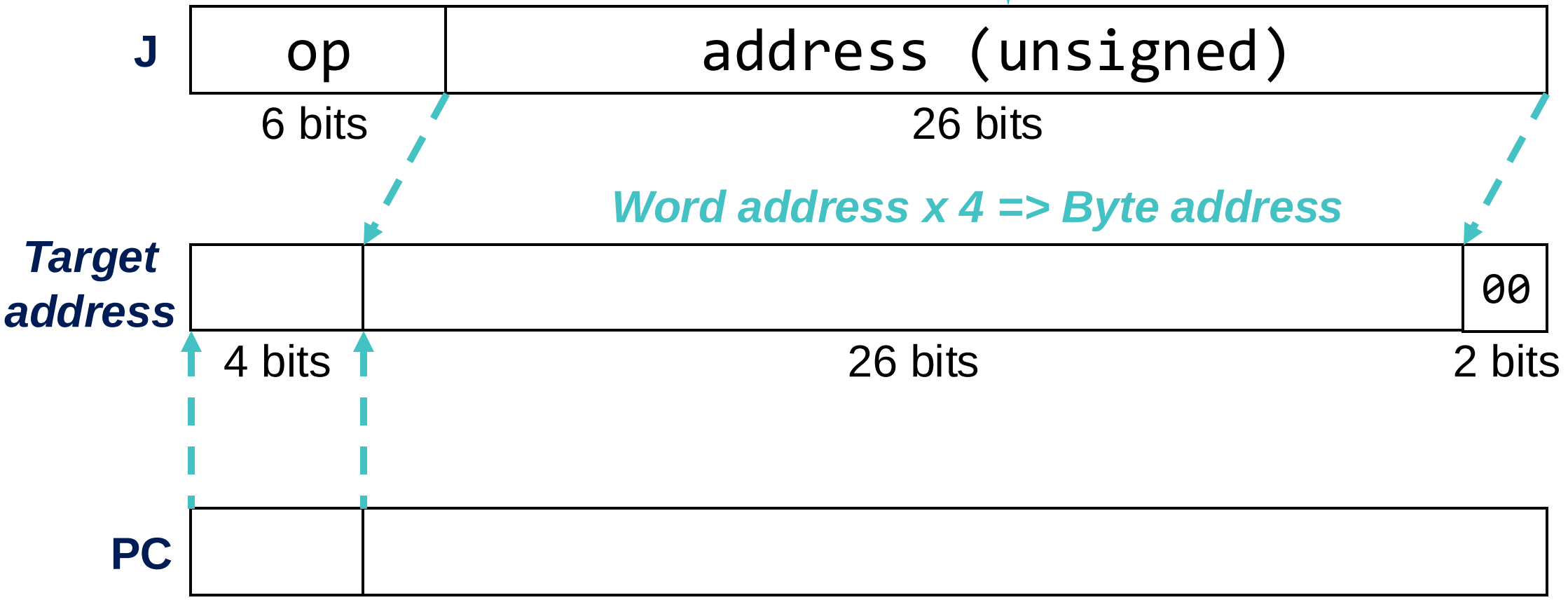
Effective Address in J-Format Instructions



Effective Address in J-Format Instructions



Effective Address in J-Format Instructions



(Pseudo) Direct Addressing Mode

Recap: Direct Mode

Effective address is equal to **the address part of the instruction**

- *Effective address*: address field of instruction
- Operand value: memory[effective address]

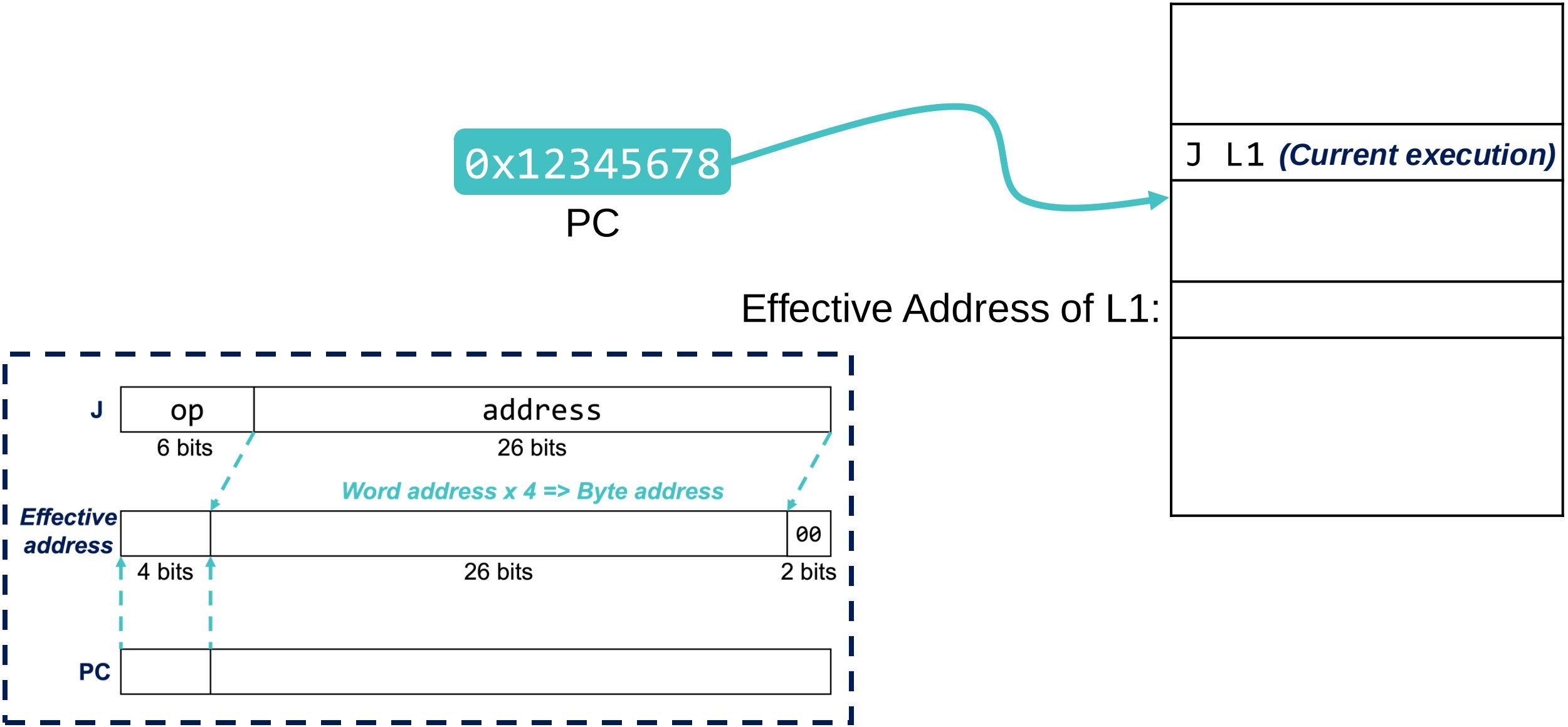
Example: Motorola 68000

```
MOVE.W 0x100,D1
```

Source

Destination

Meaning of the Pseudo Direct Mode



Meaning of the Pseudo Direct Mode

PC[31:28] = 0x1 (0001_2)

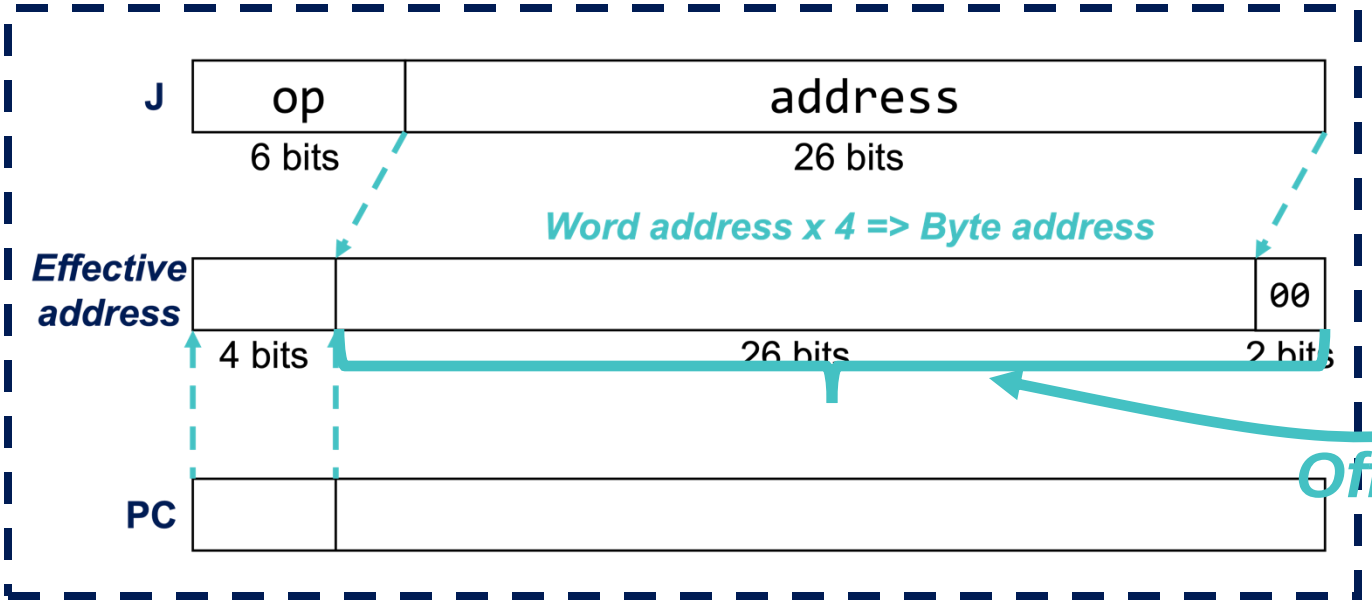
0x12345678

PC

0x10000000

J L1 *(Current execution)*

Effective Address of L1:

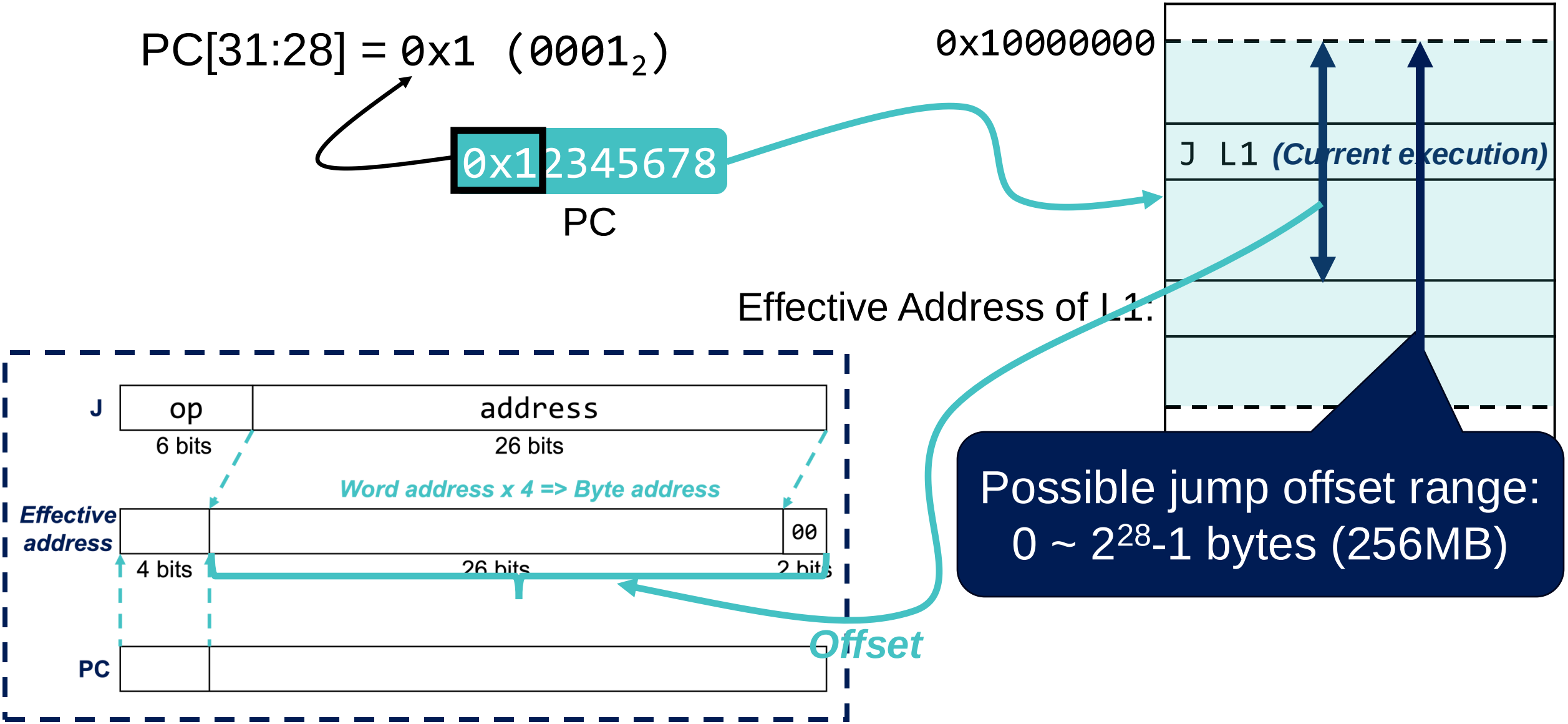


Meaning of the Pseudo Direct Mode

PC[31:28] = 0x1 (0001_2)

0x12345678

PC



Unconditional Branch Instructions

- **J**ump (J-Format)

```
j L1 # pc = pc[31:28]:(address x 4)
```

op = 2	word address (unsigned)

- **J**ump **A**nd **L**ink (J-Format)

```
jal L1  # $ra = pc; pc = pc[31:28]:(address x 4)
```

op = 3	word address (unsigned)

- **J**ump **R**egister (R-Format)

```
jr $ra # pc = $ra
```

op=0	rs=31	rt=0	rd=0	shamt=0	funct=8

Unconditional Branch Instructions

- **J**ump (J-Format)

```
j L1 # pc = pc[31:28
```

op = 2	word add (unsigned)

Used for function calls

- **J**ump **A**nd **L**ink (J-Format)

```
jal L1  # $ra = pc; pc = pc[31:28]:(address x 4)
```

op = 3	word address (unsigned)

- **J**ump **R**egister (R-Format)

```
jr $ra # pc = $ra
```

op=0	rs=31	rt=0	rd=0	shamt=0	funct=8

Branching Far Away

- **`beq $s0, $s1, L1`**
  - The range of L1: $-2^{15} \leq L1 \leq +2^{15} - 1$

  Q. What if branch target is too far to encode with 16-bit offset?

  A. Insert an *unconditional jump* to the branch target and inverts the condition (The assembler rewrites the code)

```
beq  $s0, $s1, L1
```

→

```
     bne  $s0, $s1, L2
     J    L1
L2:
```

Branching Far Away

- **`beq $s0, $s1, L1`**
  - The range of L1: $-2^{15} \leq L1 \leq +2^{15} - 1$

  Q. What if branch target is too far to encode with 16-bit offset?

  A. Insert an *unconditional jump* to the branch target and inverts the condition (The assembler rewrites the code)

- **j L1**
  - The target is anywhere within a block of 256M address where PC supplies the upper 4 bits

  Q. What if the target is out of 256 block of current PC?

  A. Use the `jr` instruction (The assembler rewrites the code)

Summary: MIPS Control Instructions

- **`beq / bne $t0, $t1, 16-bit address`**

  PC = PC + address*4

- **`j / jal 26-bit address`**

  PC = PC[31:28]:(address*4)

- **`jr $t0`**

  PC = $t0

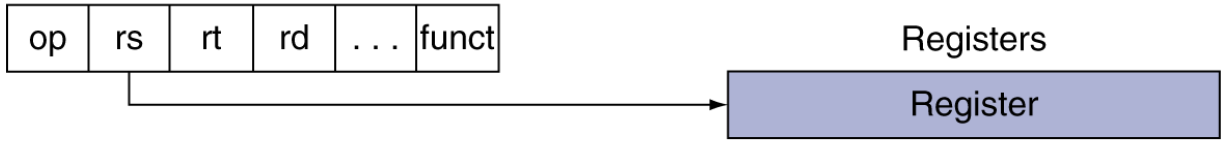
Summary: MIPS Instruction Formats

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

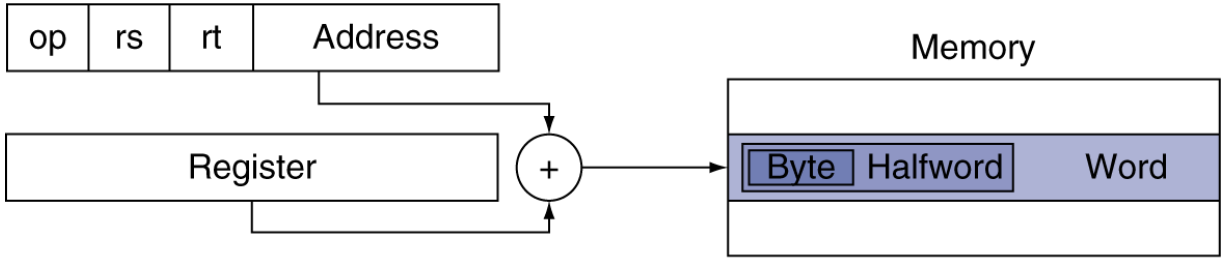
Summary: MIPS Addressing Mode

1. Immediate addressing

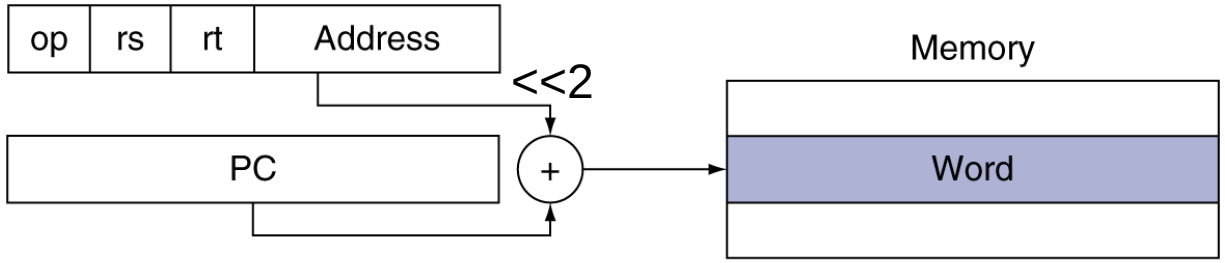
2. Register addressing



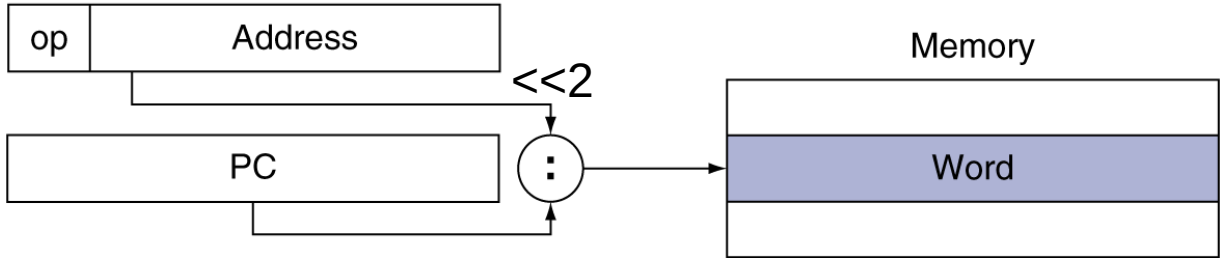
3. Base addressing



4. PC-relative addressing



5. Pseudodirect addressing



32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient

- For the occasional 32-bit constant, use the `lui` instruction
- **L**oad **U**pper **I**mmediate (I-Format)

```
lui $t0, 255
```

  - Copies 16-bit constant (255) to left 16 bits of `rt`
  - Clears right 16 bits of `rt` to 0

`lui:`	`00111`	`00000`	`01000`	`0000 0000 1111 1111`
	op	rs	rt	constant

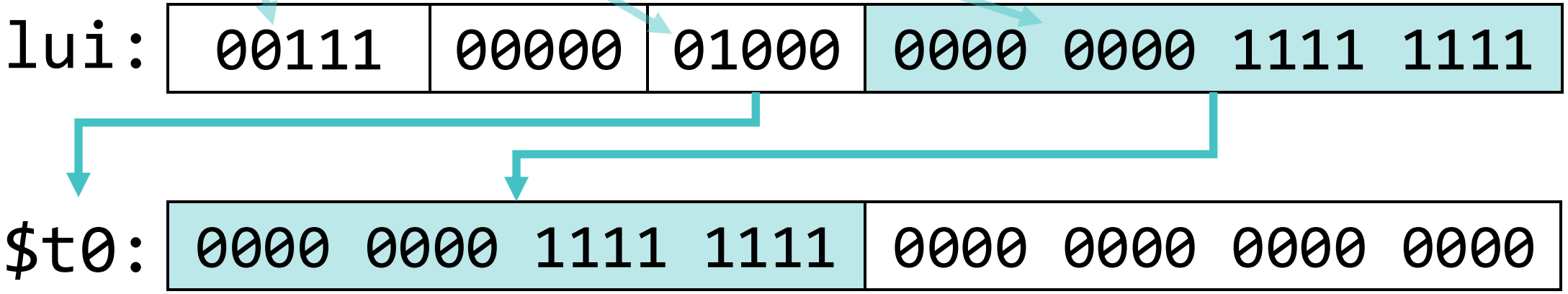
32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient

- For the occasional 32-bit constant, use the `lui` instruction
- **L**oad **U**pper **I**mmediate (I-Format)

```
lui $t0, 255
```

  - Copies 16-bit constant (255) to left 16 bits of `rt`
  - Clears right 16 bits of `rt` to 0



Assembler Pseudoinstructions

- Legal MIPS assembly language instructions that <u>do not have a direct hardware implementation</u>

Assembler Pseudoinstructions

- Legal MIPS assembly language instructions that <u>do not have a direct hardware implementation</u>

```
move $t0, $t1      →    add $t0, $zero, $t1

li   $s0, 10       →    ori $s0, $zero, 10

blt  $t0, $t1, L   →    slt $at, $t0, $t1
                        bne $at, $zero, L
```

  - Similar for ble, bgt, and bge

The assembler translates them into equivalent real MIPS instructions

Supporting Procedure Call

MIPS Call and Return Instructions

- **Call: J**ump **A**nd **L**ink (J-Format)

```
jal L1  # $ra = pc; pc = pc[31:28]:(address x 4)
```

- **Return: J**ump **R**egister (R-Format)

```
jr $ra # pc = $ra
```

Function Call (jal): Before Execution

```
     ...
     jal foo
     add $s0,$s1,$s2
     ...
foo: sub $s0,$s1,$s2
     ...
```

Function Call (jal): Before Execution

```
      ...
      jal foo
      add $s0,$s1,$s2
      ...
foo:  sub $s0,$s1,$s2
      ...
```

$ra = pc
pc = target address

Address of
add $s0,$s1,$s2

PC

$ra

Function Call (jal): Execution (1)

```
      ...
      jal foo
      add $s0,$s1,$s2
      ...
foo:  sub $s0,$s1,$s2
      ...
```

$ra = pc
pc = target address

Address of
add $s0,$s1,$s2
PC

Address of
add $s0,$s1,$s2
$ra

Save the return address for later

Function Call (jal): Execution (2)

```
        ...
        jal foo
        add $s0,$s1,$s2
        ...
foo:    sub $s0,$s1,$s2
        ...
```

$ra = pc
pc = target address

Address of
sub $s0,$s1,$s2
PC

Address of
add $s0,$s1,$s2
$ra

Function Call (jal): After Execution

```
      ...
      jal foo
      add $s0,$s1,$s2
      ...
foo:  sub $s0,$s1,$s2
      ...
```

$ra = pc
pc = target address

Control flow

Address of
sub $s0,$s1,$s2
PC

Address of
add $s0,$s1,$s2
$ra

Function Return (jr): Before Execution

```
      ...
      jal foo
      add $s0,$s1,$s2
      ...
foo:  sub $s0,$s1,$s2
      ...
      jr $ra
      ...
```

pc = $ra

Address of
...
PC

Address of
add $s0,$s1,$s2
$ra

Function Return (jr): Execution

```
      ...
      jal foo
      add $s0,$s1,$s2
      ...
foo:  sub $s0,$s1,$s2
      ...
      jr $ra
      ...
```

pc = $ra

Address of
add $s0,$s1,$s2
PC

Address of
add $s0,$s1,$s2
$ra

Function Return (jr): After Execution

```
      ...
      jal foo
      add $s0,$s1,$s2
      ...
foo:  sub $s0,$s1,$s2
      ...
      jr $ra
      ...
```

Control flow

pc = $ra

Address of
add $s0,$s1,$s2
PC

Address of
add $s0,$s1,$s2
$ra

Stack Frame

- When a function is invoked, a new ***stack frame*** is allocated at the top of the stack memory

- Also, called as <u>procedure frame</u> or <u>activation record</u>

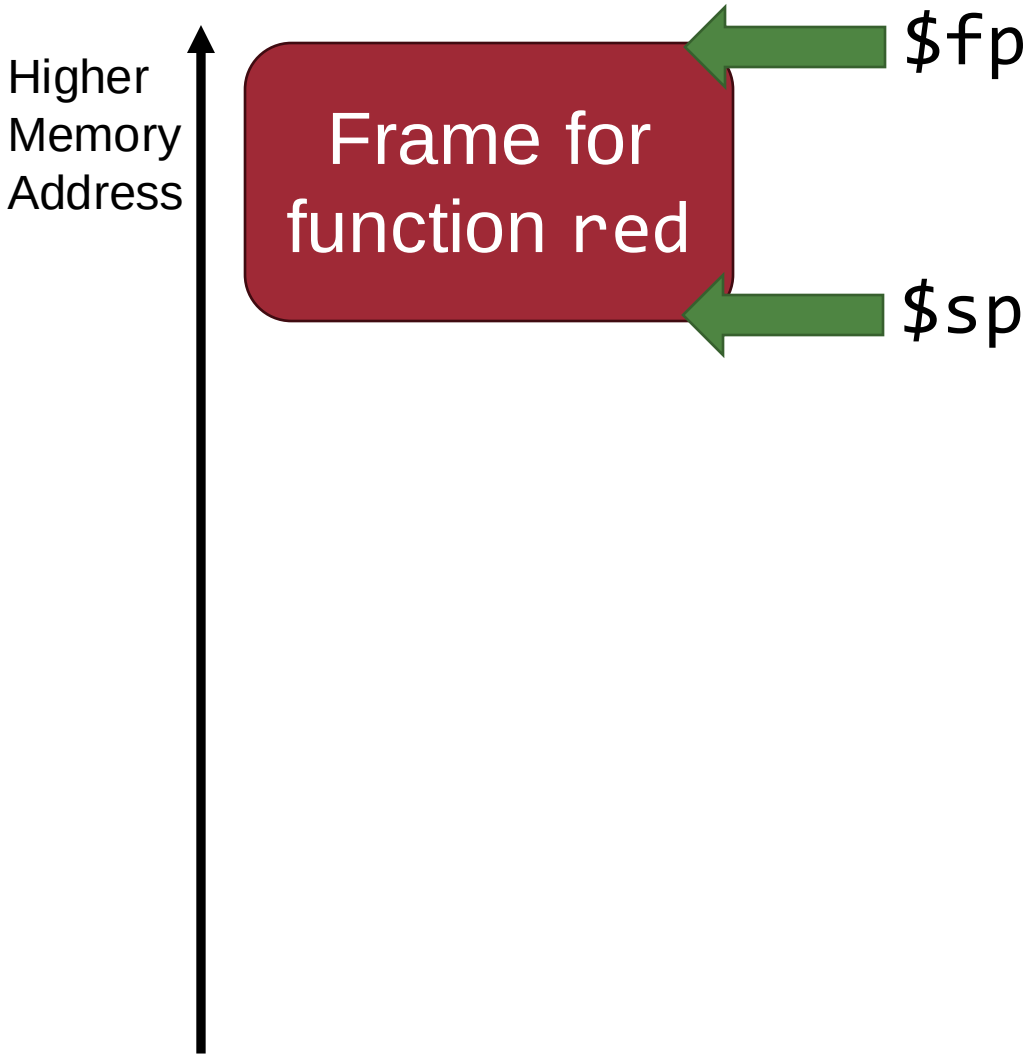
Stack Frame Example

```
int purple(int g, h) {
    int f;
    f = g + h;
    return f;
}

int blue(int a) {
    int b = 3;
    int c = purple(4, 3)
    return a + b + c;
}

int red(int a) {
    return blue(5);
}
```

Stack Frame Example

```
int purple(int g, h) {
    int f;
    f = g + h;
    return f;
}

int blue(int a) {
    int b = 3;
    int c = purple(4, 3)
    return a + b + c;
}

int red(int a) {
    return blue(5);
}
```

Start



$sp and $fp

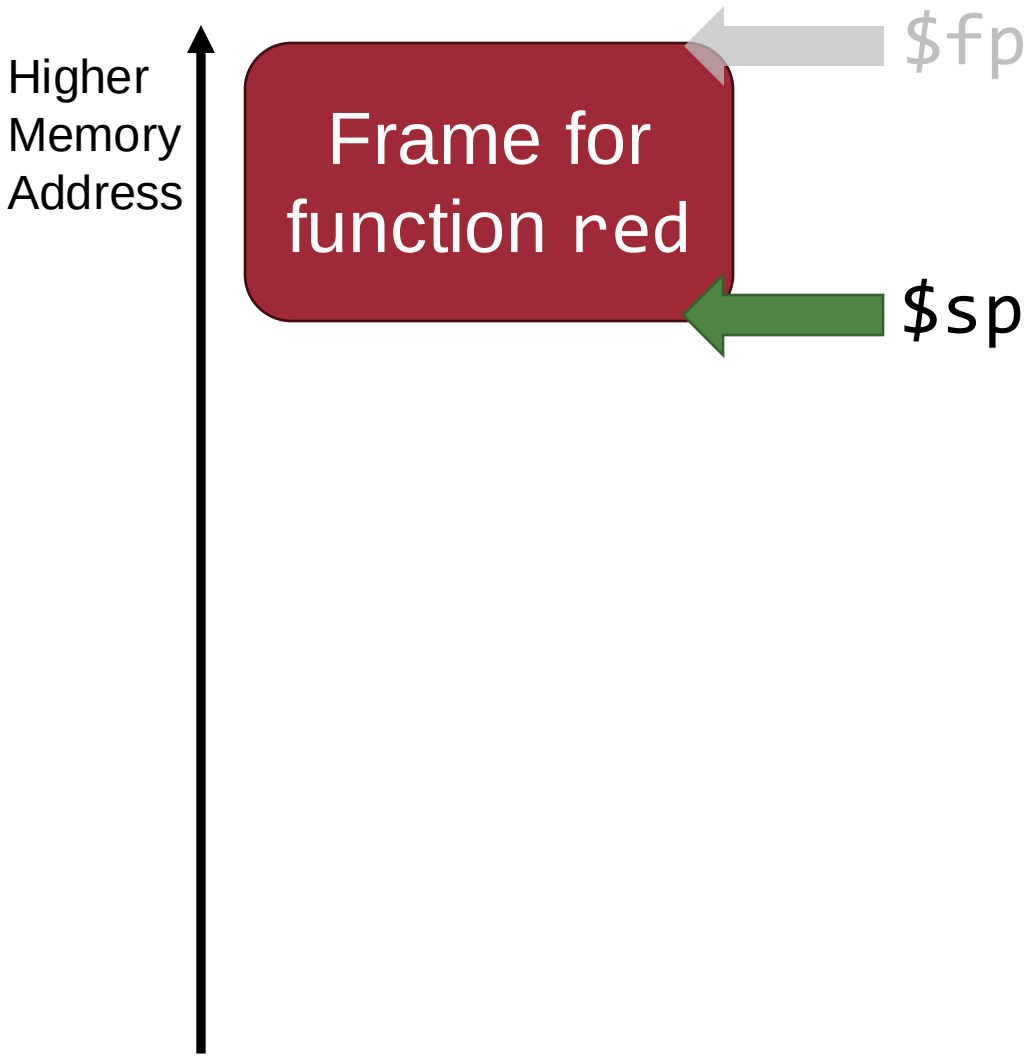
- `$sp`: stack pointer (reg 29)
  - Point to the top of each stack frame

- `$fp`: frame pointer (reg 30)
  - Point to the bottom of each stack frame, i.e., the beginning of the current stack frame
  - In MIPS architecture, `$fp` is optional and in practice rarely used

Stack Frame Example

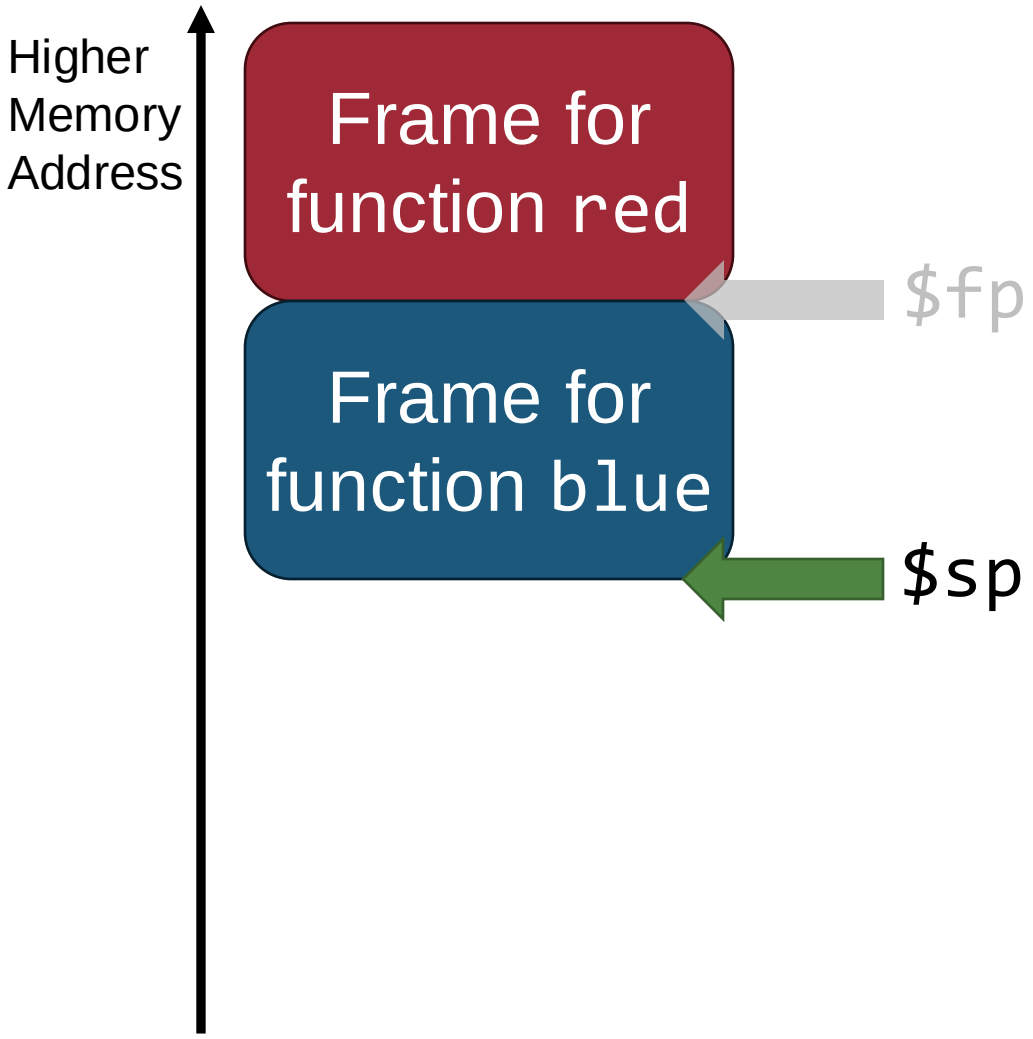
```
int purple(int g, h) {
    int f;
    f = g + h;
    return f;
}

int blue(int a) {
    int b = 3;
    int c = purple(4, 3)
    return a + b + c;
}

int red(int a) {
    return blue(5);
}
```

Start



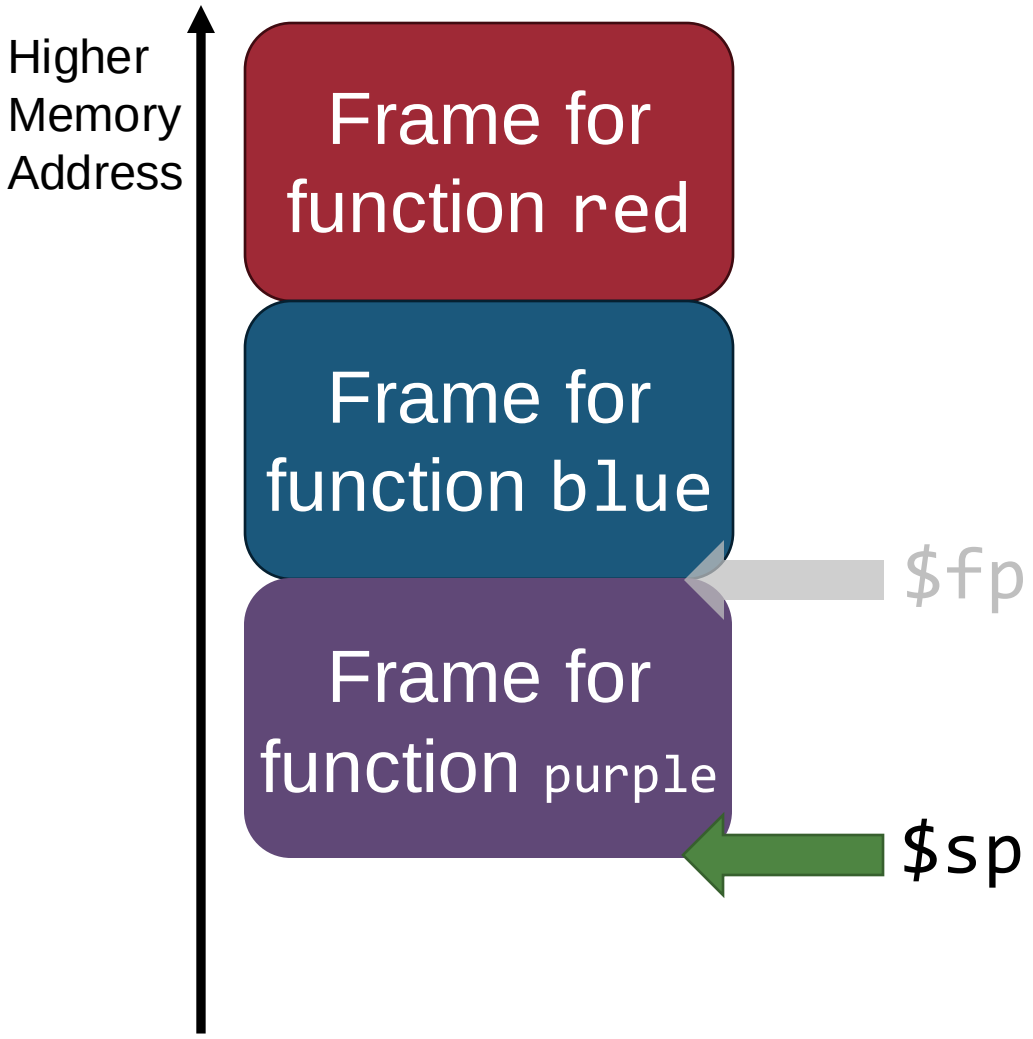
Stack Frame Example

```
int purple(int g, h) {
    int f;
    f = g + h;
    return f;
}

int blue(int a) {
    int b = 3;
    int c = purple(4, 3)
    return a + b + c;
}

int red(int a) {
    return blue(5);
}
```



Stack Frame Example

```
int purple(int g, h) {
    int f;
    f = g + h;
    return f;
}

int blue(int a) {
    int b = 3;
    int c = purple(4, 3)
    return a + b + c;
}

int red(int a) {
    return blue(5);
}
```



Calling Convention

A scheme for **how functions receive parameters** from their caller and **how they return a result**

MIPS Calling Convention

A scheme for **how functions receive parameters** from their caller and **how they return a result**

The registers
`$v0, $v1`

The registers
`$a0, $a1, $a2, $a3`

Procedure Calling

1. **Caller** places parameters in registers for the procedure (callee) to access them
   `$a0–$a3`: argument registers to pass parameters
2. **Caller** transfers control to the callee (`jal CalleeAddress`)
3. **Callee** acquires storage for procedure
4. **Callee** performs the desired task
5. **Callee** places the result value in register for **caller** to access it
   `$v0–$v1`: value registers to return result values
6. **Callee** returns control to the **caller** (`jr $ra`)

Leaf and Non-leaf Procedures

```
int purple(int g, h) {
    int f;
    f = g + h;
    return f;
}
```

Leaf procedure

```
int blue(int a) {
    int b = 3;
    int c = purple(4, 3)
    return a + b + c;
}

int red(int a) {
    return blue(5);
}
```

Higher Memory Address

Frame for function red

Frame for function blue

Frame for function purple

Non-leaf procedures

Leaf and Non-leaf Procedures

```
int purple(int g, h) {
    int f;
    f = g + h;
    return f;
}
```

Leaf procedure

Higher Memory Address

Frame for function red

Frame for function purple

```
    return a + b + c;
}

int red(int a) {
    return blue(5);
}
```

Non-leaf procedures

The contents stored in the stack
differ between leaf procedures and non-leaf procedures

Leaf Procedure Example

```
int purple(int g, h) {
    int f; // f in $s0
    f = g + h;
    return f;
}

int blue(int a) {
    int b = 3; // $s0=3
    int c = purple(4, 3)
    return a + b + c;
}

int red(int a) {
    return blue(5);
}
```

Leaf procedure

Leaf Procedure Example

0xbffeffee

Main memory

$ra: 0x50 (blue's ret. addr.)
$a1: 0x3 (purple's arg)
$a0: 0x4 (purple's arg)
pc: 0x14
$sp: 0xbffeffee
$s0: 0x3 (blue's $s0)

Execution context

```
purple:
14: addi $sp, $sp, -4
18: sw   $s0, 0($sp)
1c: add  $s0, $a0, $a1
20: add  $v0, $s0, $zero
24: lw   $s0, 0($sp)
28: addi $sp, $sp, 4
2c: jr   $ra

blue:
...
4c: jal  purple
50: add  $t0, $s0, $v0
```

Leaf Procedure Example

0xbffeffee

Main memory

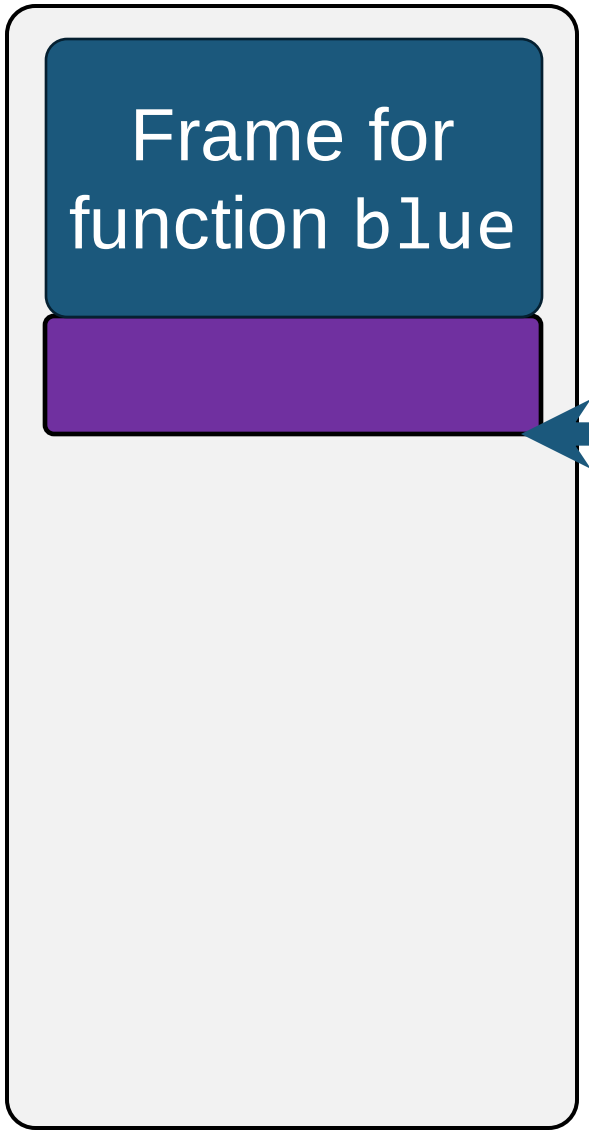
$ra: 0x50 (blue's ret. addr.)
$a1: 0x3 (purple's arg)
$a0: 0x4 (purple's arg)
pc: 0x18
$sp: 0xbffeffee
$s0: 0x3 (blue's $s0)

Execution context

Currently executed instruction

```
purple:
 14: addi $sp, $sp, -4
 18: sw   $s0, 0($sp)
 1c: add  $s0, $a0, $a1
 20: add  $v0, $s0, $zero
 24: lw   $s0, 0($sp)
 28: addi $sp, $sp, 4
 2c: jr   $ra

blue:

 ...
 4c: jal  purple
 50: add  $t0, $s0, $v0
```

Leaf Procedure Example



0xbffeffea

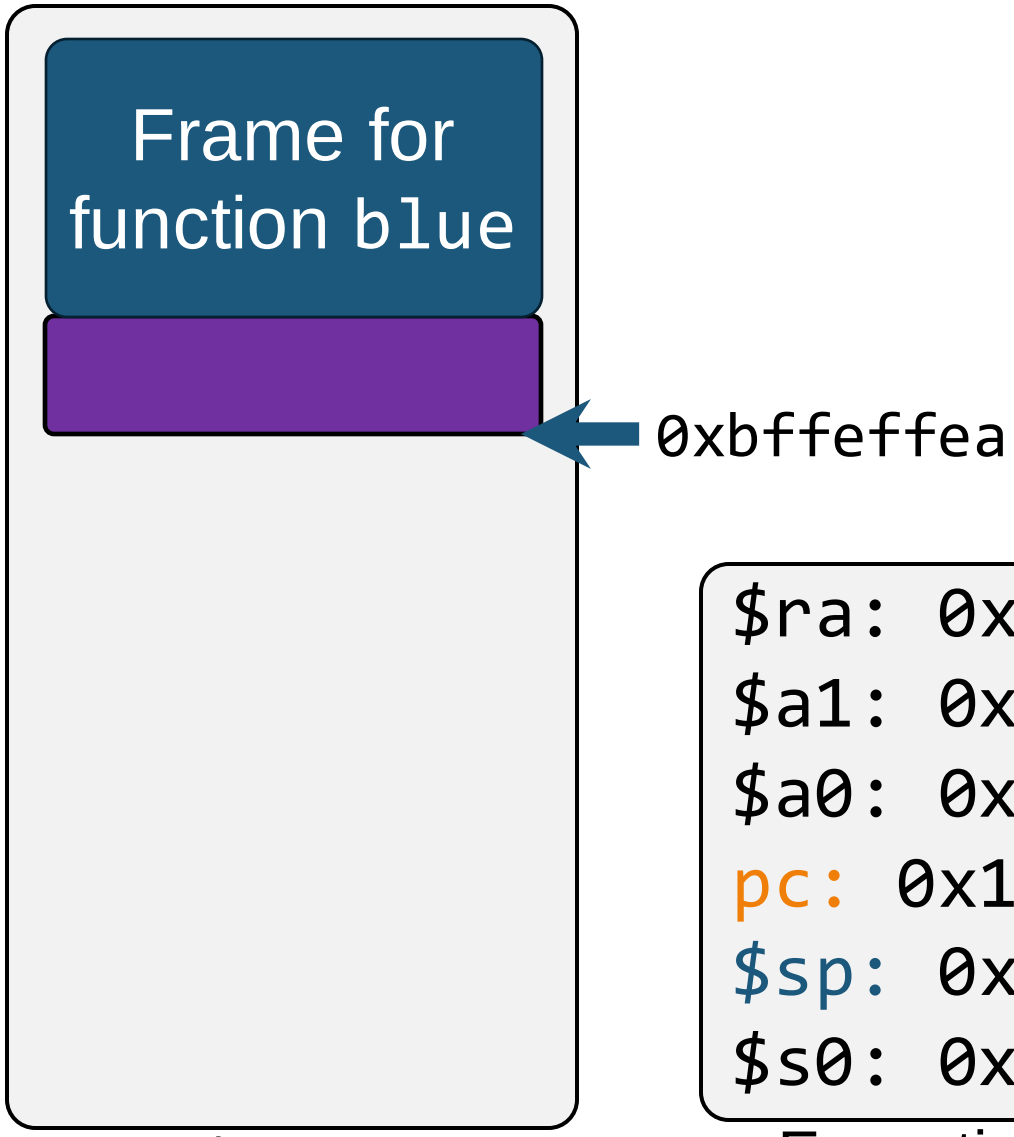
Main memory

$ra: 0x50 (blue's ret. addr.)
$a1: 0x3 (purple's arg)
$a0: 0x4 (purple's arg)
pc: 0x18
$sp: 0xbffeffea
$s0: 0x3 (blue's $s0)

Execution context

```
purple:
 14: addi $sp, $sp, -4
 18: sw   $s0, 0($sp)
 1c: add  $s0, $a0, $a1
 20: add  $v0, $s0, $zero
 24: lw   $s0, 0($sp)
 28: addi $sp, $sp, 4
 2c: jr   $ra

blue:

 ...
 4c: jal  purple
 50: add  $t0, $s0, $v0
```

Leaf Procedure Example



Main memory

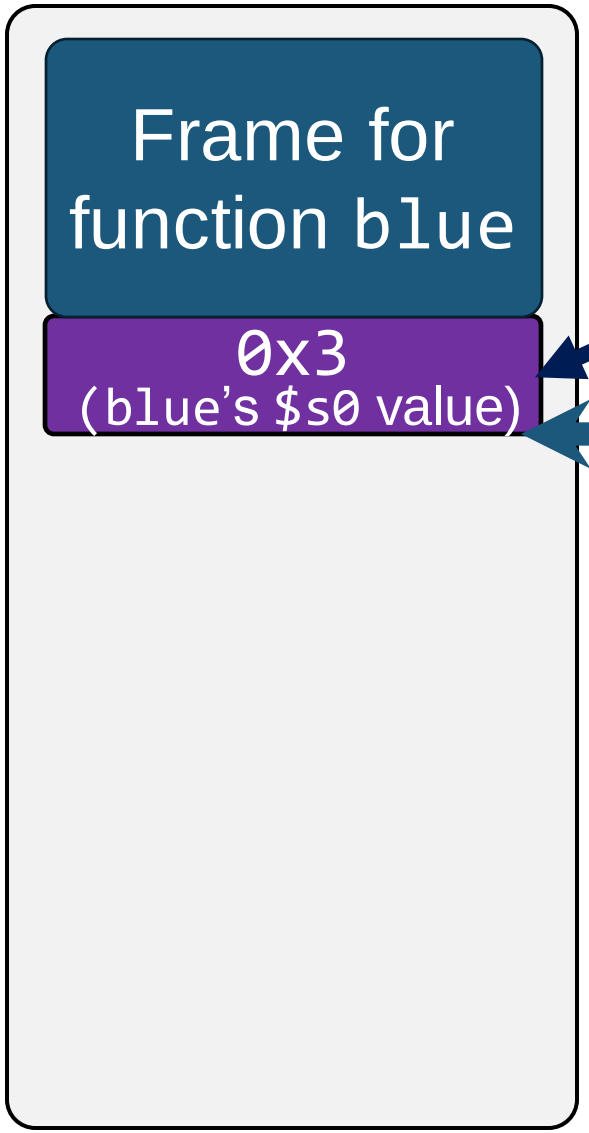
```
$ra:  0x50 (blue's ret. addr.)
$a1:  0x3 (purple's arg)
$a0:  0x4 (purple's arg)
pc:  0x1c
$sp:  0xbffeffea
$s0:  0x3 (blue's $s0)
```

Execution context

```
purple:
  14: addi $sp, $sp, -4
  18: sw   $s0, 0($sp)
  1c: add  $s0, $a0, $a1
  20: add  $v0, $s0, $zero
  24: lw   $s0, 0($sp)
  28: addi $sp, $sp, 4
  2c: jr   $ra

blue:

  ...
  4c: jal  purple
  50: add  $t0, $s0, $v0
```

Leaf Procedure Example

Register $s0 was used in blue, but since it will be **newly used in `leaf_example`**, it is backed up



Main memory

0xbffeffea

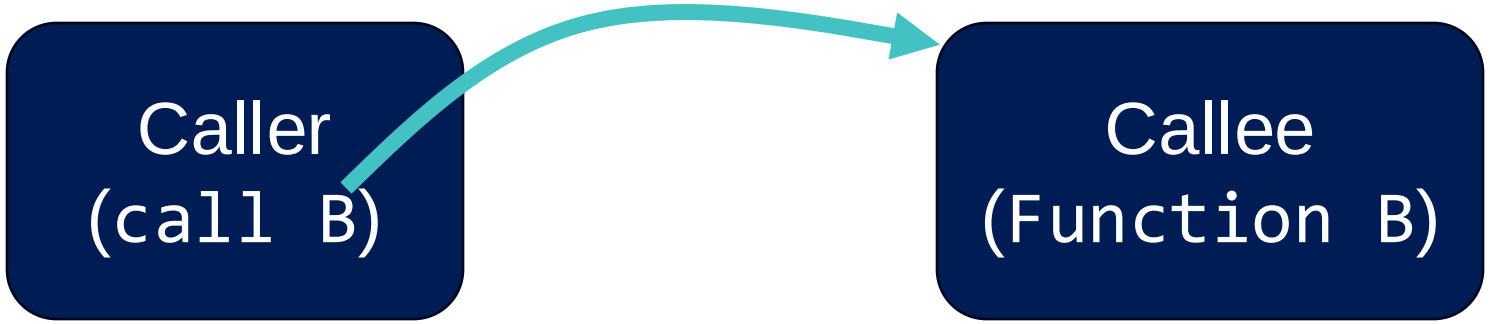
```
$ra: 0x50 (blue's ret. addr.)
$a1: 0x3 (purple's arg)
$a0: 0x4 (purple's arg)
pc: 0x1c
$sp: 0xbffeffea
$s0: 0x3 (blue's $s0)
```

Execution context

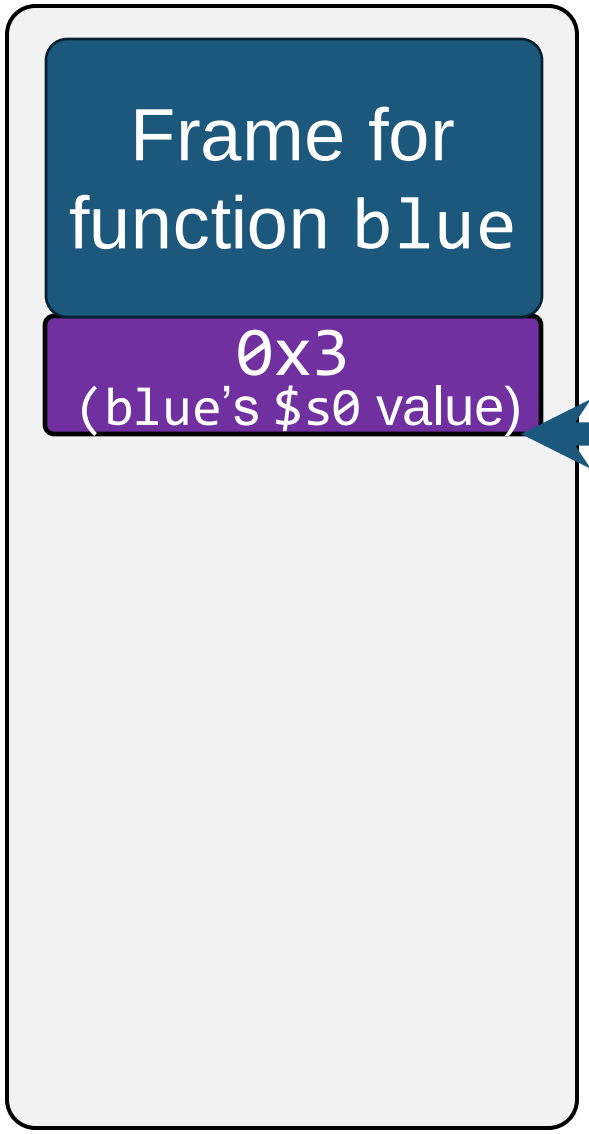
```
18: sw   $s0, 0($sp)
1c: add  $s0, $a0, $a1
20: add  $v0, $s0, $zero
24: lw   $s0, 0($sp)
28: addi $sp, $sp, 4
2c: jr   $ra

blue:

...
4c: jal  purple
50: add  $t0, $s0, $v0
```

Note: The Convention to Reduce Register Spilling

- **`$t0 – $t9`**: temporaries
  - Can be overwritten by callee
- **`$s0 – $s7`**: saved
  - Must be saved/restored (i.e., preserved) by callee



Leaf Procedure Example



0xbffeffea

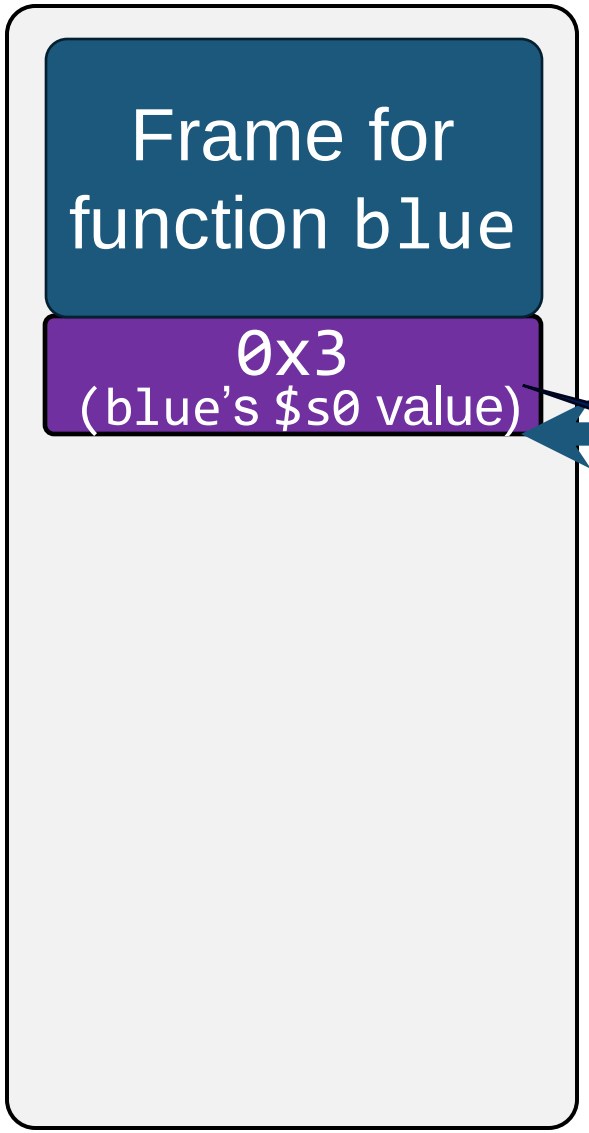
Main memory

$ra: 0x50 (blue's ret. addr.)
$a1: 0x3 (purple's arg)
$a0: 0x4 (purple's arg)
pc: 0x1c
$sp: 0xbffeffea
$s0: 0x3 (blue's $s0)

Execution context

```
purple:
  14: addi $sp, $sp, -4
  18: sw   $s0, 0($sp)
  1c: add  $s0, $a0, $a1
  20: add  $v0, $s0, $zero
  24: lw   $s0, 0($sp)
  28: addi $sp, $sp, 4
  2c: jr   $ra

blue:
  ...
  4c: jal  purple
  50: add  $t0, $s0, $v0
```

Leaf Procedure Example



Main memory

0xbffeffea

$ra: 0x50 (blue's
$a1: 0x3 (purple's arg)
$a0: 0x4 (purple's arg)
pc: 0x20
$sp: 0xbffeffea
$s0: 0x7 (purple's $s0)

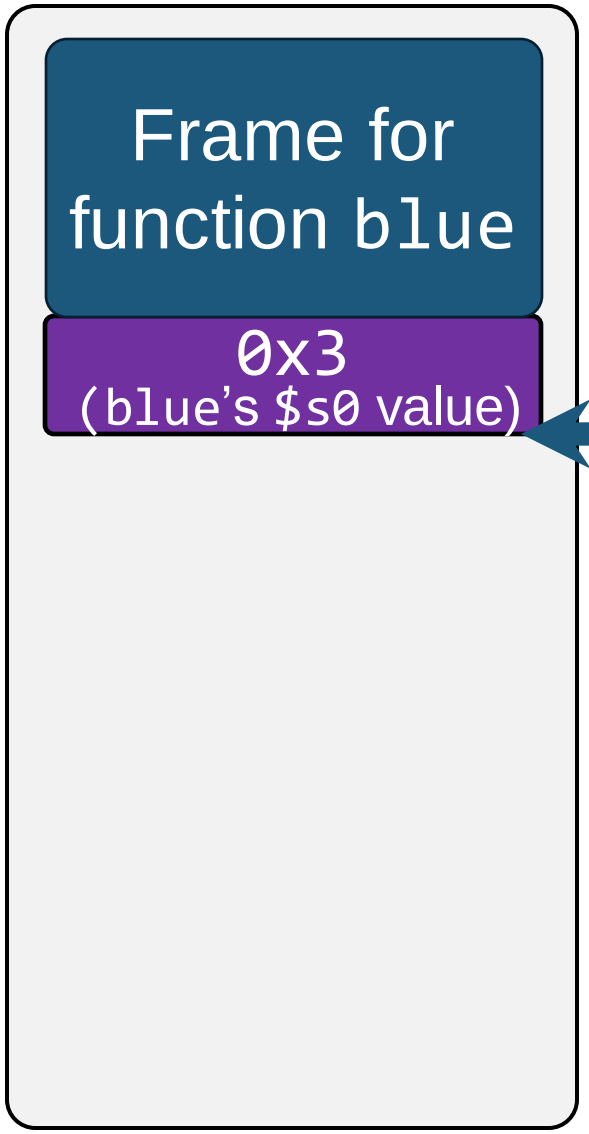
Execution context

```
purple:
 14: addi $sp, $sp, -4
 18: sw   $s0, 0($sp)
 1c: add  $s0, $a0, $a1
 20: add  $v0, $s0, $zero
 24: lw   $s0, 0($sp)
```

blue's $s0 is overwritten, but it is preserved in the stack

```
 ..
 4c: jal  purple
 50: add  $t0, $s0, $v0
```

Leaf Procedure Example



Main memory

Return value

0xbffe[illegible]ea

$v0: 0x7

$ra: 0x50 (blue's ret. addr.)

$a1: 0x3 (purple's arg)

$a0: 0x4 (purple's arg)

pc: 0x24

$sp: 0xbffeffea

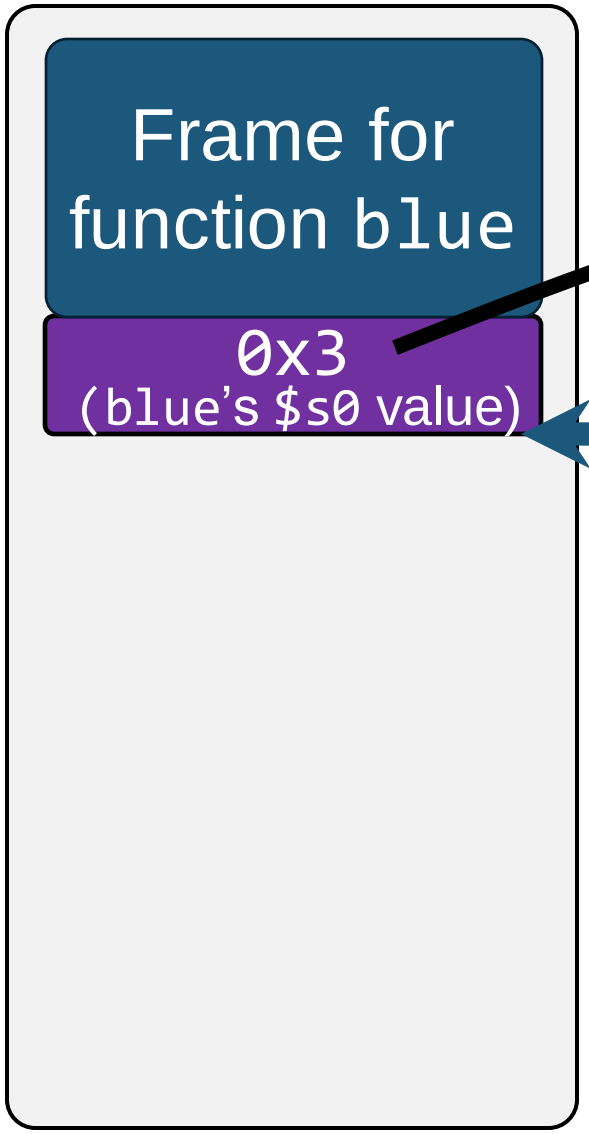
$s0: 0x7 (purple's $s0)

Execution context

```
purple:
 14: addi $sp, $sp, -4
 18: sw   $s0, 0($sp)
 1c: add  $s0, $a0, $a1
 20: add  $v0, $s0, $zero
 24: lw   $s0, 0($sp)
 28: addi $sp, $sp, 4
 2c: jr   $ra

blue:

 ...
 4c: jal  purple
 50: add  $t0, $s0, $v0
```

Leaf Procedure Example

Now, restore blue's $s0 before leaving the function



Main memory

0xbffeffea

$v0: 0x7
$ra: 0x50 (blue's ret. addr.)
$a1: 0x3 (purple's arg)
$a0: 0x4 (purple's arg)
pc: 0x28
$sp: 0xbffeffea
$s0: 0x3 (blue's $s0)

Execution context

```
14: addi $sp, $sp, -4
18: sw   $s0, 0($sp)
1c: add  $s0, $a0, $a1
20: add  $v0, $s0, $zero
24: lw   $s0, 0($sp)
28: addi $sp, $sp, 4
2c: jr   $ra

blue:

...
4c: jal  purple
50: add  $t0, $s0, $v0
```

Leaf Procedure Example

Main memory

0xbffeffee

```
$v0:  0x7
$ra:  0x50 (blue's ret. addr.)
$a1:  0x3 (purple's arg)
$a0:  0x4 (purple's arg)
pc:  0x2c
$sp:  0xbffeffee
$s0:  0x3 (blue's $s0)
```

Execution context

```
purple:
 14: addi $sp, $sp, -4
 18: sw   $s0, 0($sp)
 1c: add  $s0, $a0, $a1
 20: add  $v0, $s0, $zero
 24: lw   $s0, 0($sp)
 28: addi $sp, $sp, 4
 2c: jr   $ra

blue:

 ...
 4c: jal  purple
 50: add  $t0, $s0, $v0
```

Leaf Procedure Example

0xbffeffee

Main memory

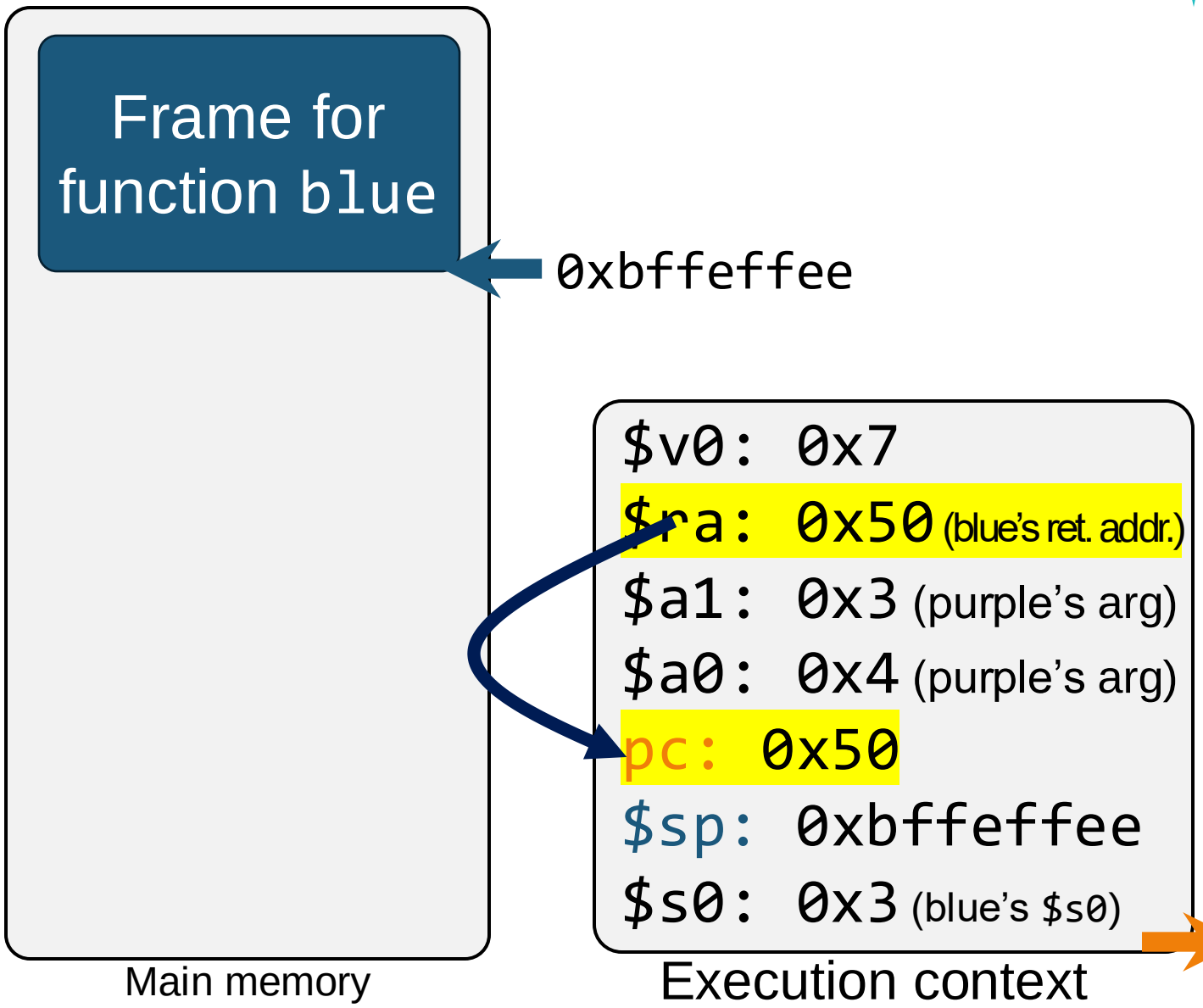
$v0: 0x7
$ra: 0x50 (blue's ret. addr.)
$a1: 0x3 (purple's arg)
$a0: 0x4 (purple's arg)
pc: 0x50
$sp: 0xbffeffee
$s0: 0x3 (blue's $s0)

Execution context

```
purple:
 14: addi $sp, $sp, -4
 18: sw   $s0, 0($sp)
 1c: add  $s0, $a0, $a1
 20: add  $v0, $s0, $zero
 24: lw   $s0, 0($sp)
 28: addi $sp, $sp, 4
 2c: jr   $ra

blue:
 ...
 4c: jal  purple
 50: add  $t0, $s0, $v0
```

pc = $ra

Leaf Procedure Example




```
purple:
 14: addi $sp, $sp, -4
 18: sw   $s0, 0($sp)
 1c: add  $s0, $a0, $a1
 20: add  $v0, $s0, $zero
 24: lw   $s0, 0($sp)
 28: addi $sp, $sp, 4
 2c: jr   $ra

blue:
 ...
 4c: jal  purple
 50: add  $t0, $s0, $v0
```

Recap: MIPS General Purpose Registers

#	Name	Usage
0	$zero	The constant value 0
1	$at	Assembler temporary
2	$v0	Values for results and expression evaluation
3	$v1	
4	$a0	Arguments
5	$a1	
6	$a2	
7	$a3	
8	$t0	Temporaries (Caller-save registers)
9	$t1	
14	$t6	
15	$t7	

#	Name	Usage
16	$s0	Saved temporaries (Callee-save registers)
17	$s1	
18	$s2	
19	$s3	
20	$s4	
21	$s5	
22	$s6	
23	$s7	
24	$t8	More temporaries (Caller-save registers)
25	$t9	
26	$k0	Reserved for OS kernel
27	$k1	
28	$gp	Global pointer
29	$sp	Stack pointer
30	$fp	Frame pointer
31	$ra	Return address

Used for function calls

Recap: MIPS General Purpose Registers

Registers primarily used as variables in programs

#	Name	Usage
0	$zero	The constant value 0
	$a1	
	$a2	
7	$a3	
8	$t0	Temporaries (Caller-save registers)
9	$t1	
10	$t2	
11	$t3	
12	$t4	
13	$t5	
14	$t6	
15	$t7	

#	Name	Usage
16	$s0	Saved temporaries (Callee-save registers)
17	$s1	
18	$s2	
19	$s3	
20	$s4	
21	$s5	
22	$s6	
23	$s7	
24	$t8	More temporaries (Caller-save registers)
25	$t9	
26	$k0	Reserved for OS kernel
27	$k1	
28	$gp	Global pointer
29	$sp	Stack pointer
30	$fp	Frame pointer
31	$ra	Return address

Life would be simple if all procedures were leaf procedures, but they aren’t ☹
Let’s consider about non-leaf procedures[1)]

1) Procedures that call others

Be Careful When Invoking Non-Leaf Func.

```
int purple(int g, h) {
    int f;
    f = g + h;
    return f;
}

int blue(int a) {
    int b = 3;
    c = purple(4, 3)
    return a + b + c;
}

int red(int a) {
    return blue(5);
}
```

$ra: return address for **red**

Be Careful When Invoking Non-Leaf Func.

```
int purple(int g, h) {
    int f;
    f = g + h;
    return f;
}

int blue(int a) {
    int b = 3;
    c = purple(4, 3)
    return a + b + c;
}

int red(int a) {
    return blue(5);
}
```

~~$ra: return address for red~~

At this time, due to `jal`...
$ra: return address for **blue**

The return address can be overwritten, losing the return to red later

Be Careful When Invoking Non-Leaf Func.

```
int purple(int g, h) {
    int f;
    f = g + h;
    return f;
}

int blue(int a) {
    int b = 3;
    c = purple(4, 3)
    return a + b + c;
}

int red(int a) {
    return blue(5);
}
```

$a0: 5 (**blue**'s argument register)

Be Careful When Invoking Non-Leaf Func.

```
int purple(int g, h) {
    int f;
    f = g + h;
    return f;
}

int blue(int a) {
    int b = 3;
    c = purple(4, 3)
    return a + b + c;
}

int red(int a) {
    return blue(5);
}
```

~~$a0: 5 (blue's argument register)~~

At this time...
$a0: 4 (purple's argument register)

The information of the arguments can be overwritten

How can we address these problems?
→ leverage stack

1) Procedures that call others

Overview of Non-leaf Procedures

```
int purple(int g, h) {
    int f;
    f = g + h;
    return f;
}

int blue(int a) {
    int b = 3;
    int c = purple(4, 3)
    return a + b + c;
}

int red(int a) {
    return blue(5);
}
```

In this `blue`'s stack frame,

- Push `blue`'s argument registers (`$a0-$a3`)
- Push `red`'s return address (`$ra`)
- Push `red`'s registers (`$s0-$s7`)
  ← Same with leaf procedures
- Push temporary registers before call `purple` (`$t0-$t9`)

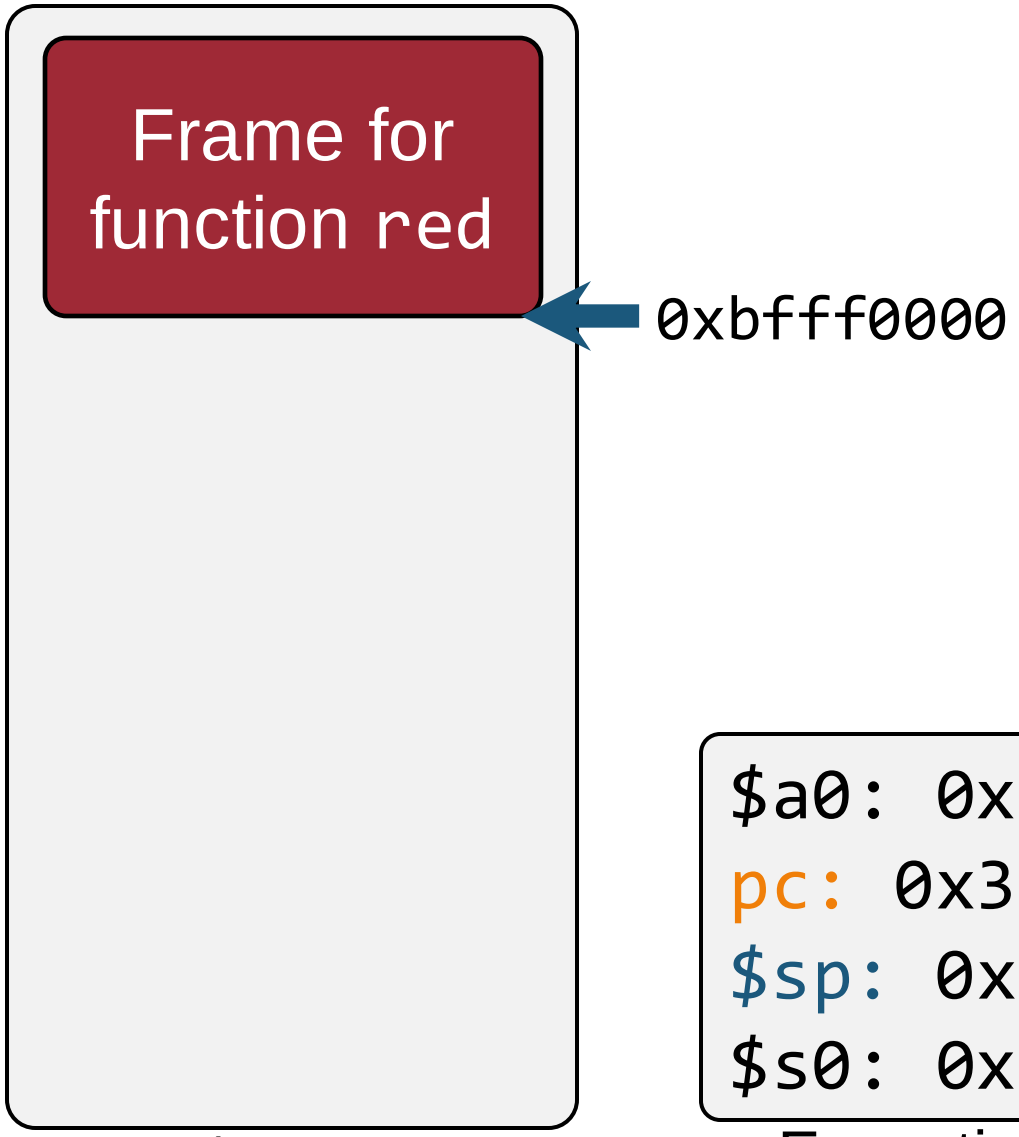
+ Arrays, structures, …

Non-leaf Procedures

```c
int purple(int g, h) {
    int f;
    f = g + h;
    return f;
}

int blue(int a) {
    int b = 3;
    int c = purple(4, 3)
    return a + b + c;
}

int red(int a) {
    return blue(5);
}
```

```
blue:
30: addi $sp, $sp, -12
34: sw   $ra, 8($sp)
38: sw   $a0, 4($sp)
3c: sw   $s0, 0($sp)
40: addi $s0, $zero, 3
44: addi $a0, $zero, 4
48: addi $a1, $zero, 3
4c: jal  purple
50: add  $t0, $s0, $v0
54: lw   $s0, 0($sp)
58: lw   $a0, 4($sp)
5c: lw   $ra, 8($sp)
60: add  $v0, $a0, $t0
64: addi $sp, $sp, 12
68: jr   $ra
```

Non-leaf Procedure Example



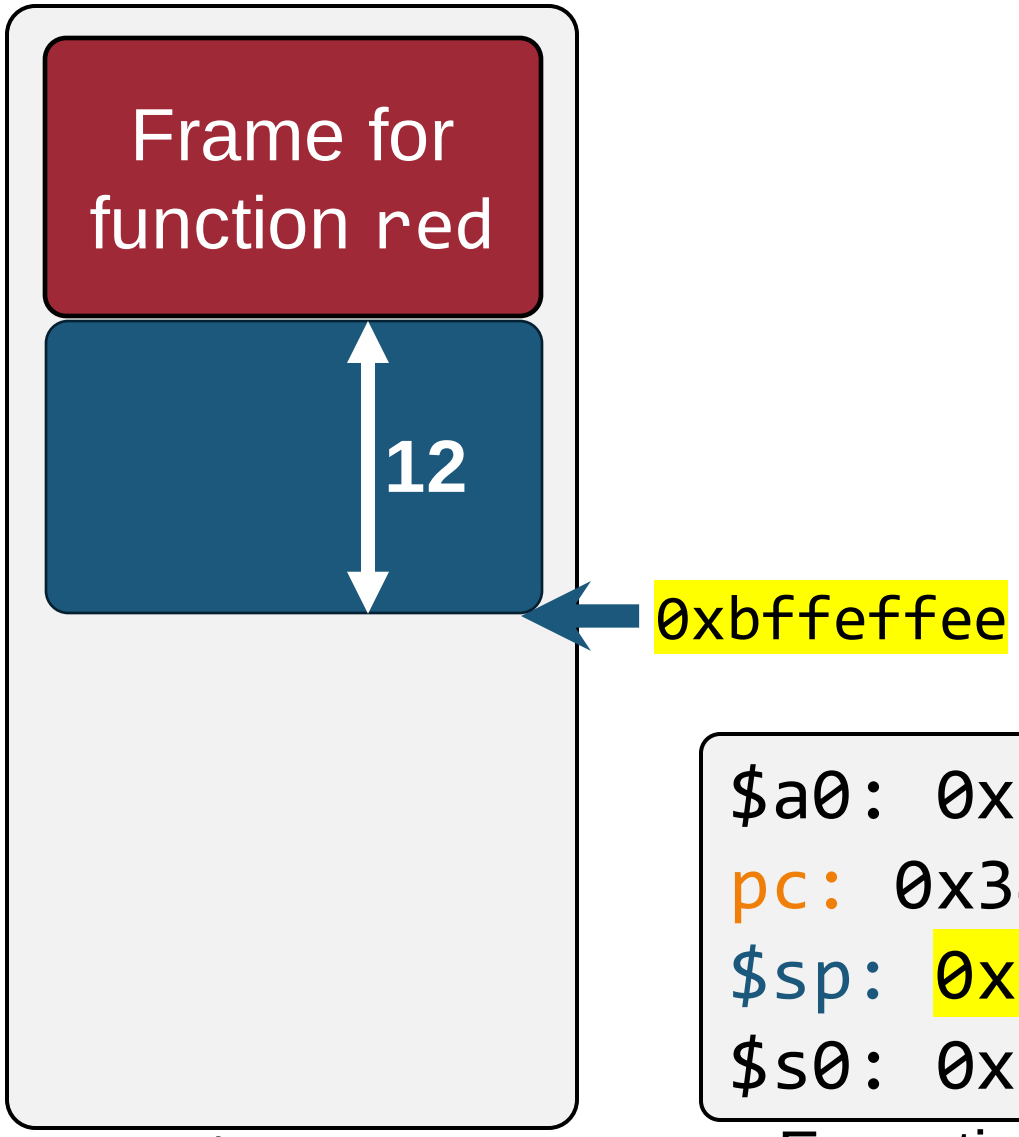
Main memory

$a0: 0x5 (blue's arg)
pc: 0x30
$sp: 0xbfff0000
$s0: 0x1 (red's $s0)

Execution context

```
blue:
 30: addi $sp, $sp, -12
 34: sw   $ra, 8($sp)
 38: sw   $a0, 4($sp)
 3c: sw   $s0, 0($sp)
 40: addi $s0, $zero, 3
 44: addi $a0, $zero, 4
 48: addi $a1, $zero, 3
 4c: jal  purple
 50: add  $t0, $s0, $v0
 54: lw   $s0, 0($sp)
 58: lw   $a0, 4($sp)
 5c: lw   $ra, 8($sp)
 60: add  $v0, $a0, $t0
 64: addi $sp, $sp, 12
 68: jr   $ra
```

Non-leaf Procedure Example



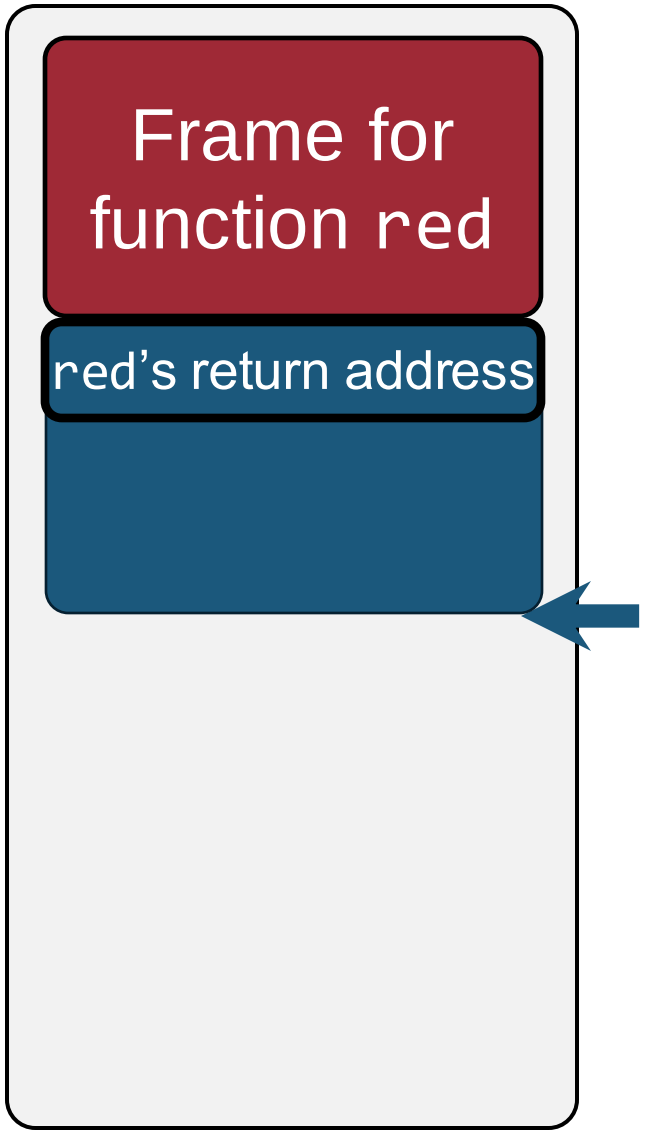
Main memory

$a0: 0x5 (blue's arg)
pc: 0x34
$sp: 0xbffeffee
$s0: 0x1 (red's $s0)

Execution context

```
blue:
30: addi $sp, $sp, -12
34: sw   $ra, 8($sp)
38: sw   $a0, 4($sp)
3c: sw   $s0, 0($sp)
40: addi $s0, $zero, 3
44: addi $a0, $zero, 4
48: addi $a1, $zero, 3
4c: jal  purple
50: add  $t0, $s0, $v0
54: lw   $s0, 0($sp)
58: lw   $a0, 4($sp)
5c: lw   $ra, 8($sp)
60: add  $v0, $a0, $t0
64: addi $sp, $sp, 12
68: jr   $ra
```

Non-leaf Procedure Example



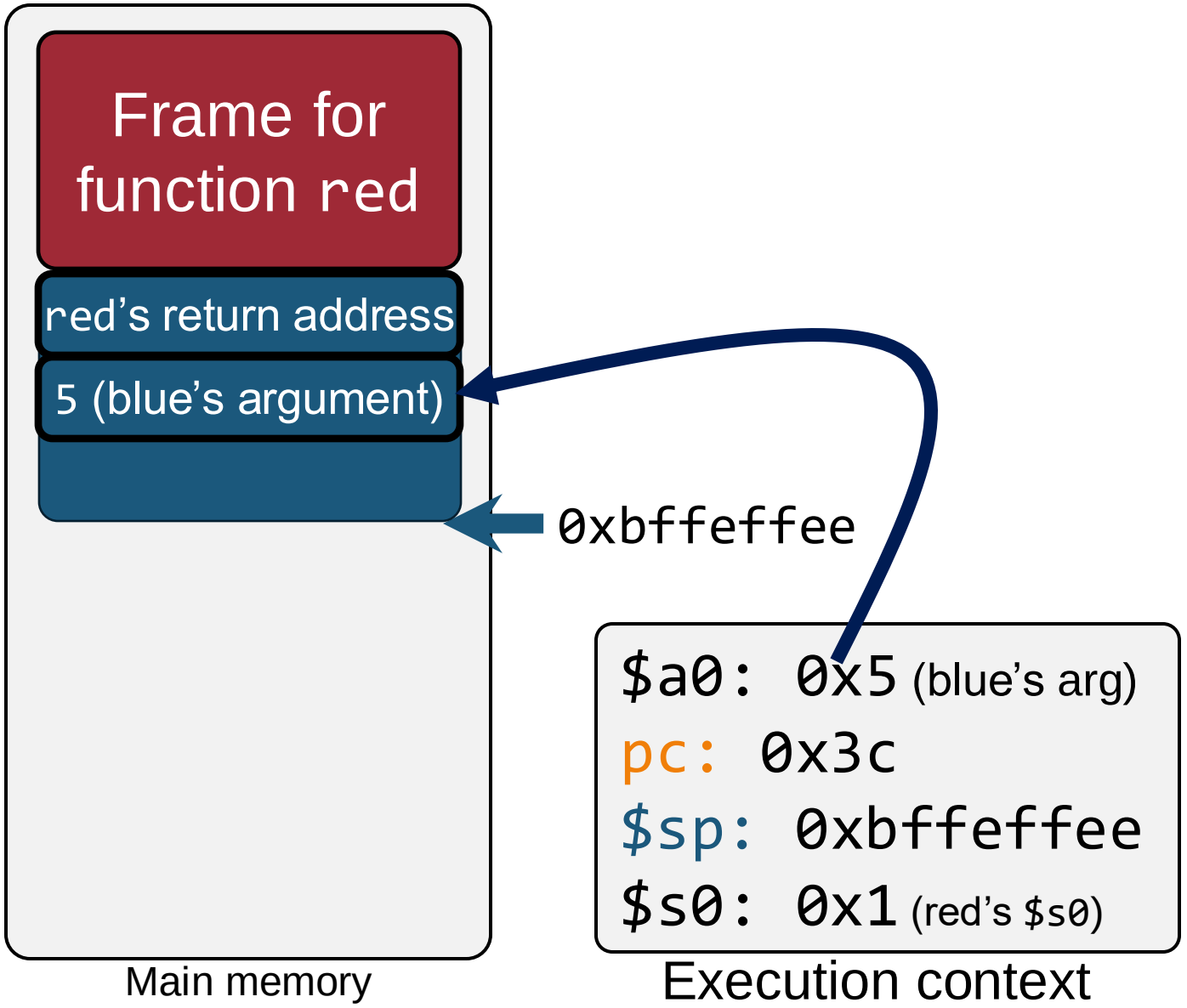
Main memory

```
$a0:  0x5 (blue's arg)
pc:  0x38
$sp:  0xbffeffee
$s0:  0x1 (red's $s0)
```

Execution context

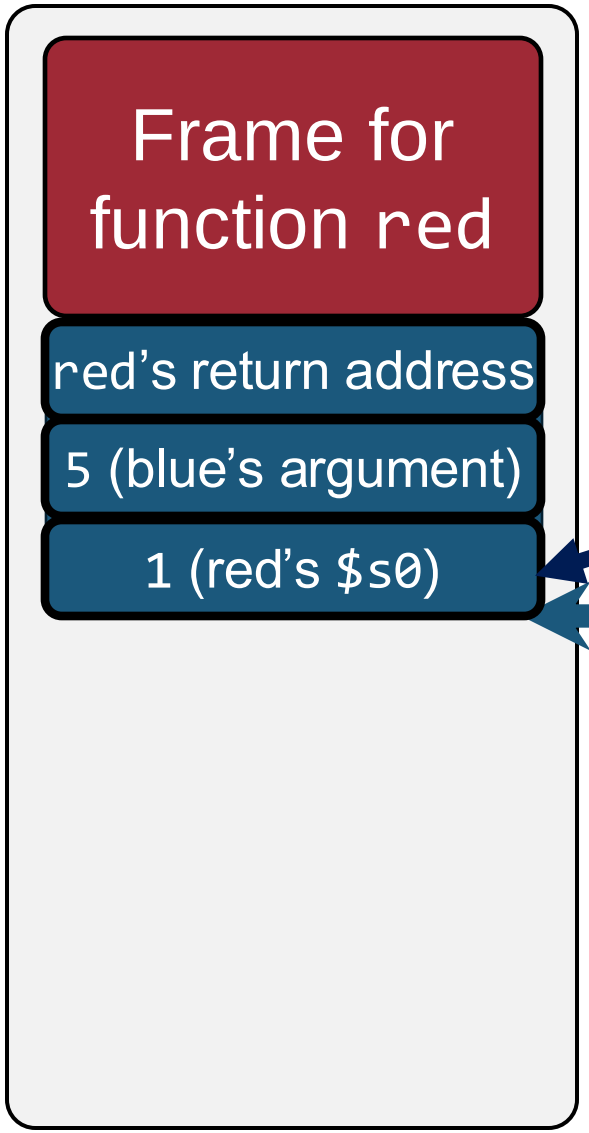
```
blue:
 30: addi $sp, $sp, -12
 34: sw   $ra, 8($sp)
 38: sw   $a0, 4($sp)
 3c: sw   $s0, 0($sp)
 40: addi $s0, $zero, 3
 44: addi $a0, $zero, 4
 48: addi $a1, $zero, 3
 4c: jal  purple
 50: add  $t0, $s0, $v0
 54: lw   $s0, 0($sp)
 58: lw   $a0, 4($sp)
 5c: lw   $ra, 8($sp)
 60: add  $v0, $a0, $t0
 64: addi $sp, $sp, 12
 68: jr   $ra
```

Non-leaf Procedure Example




```
blue:
 30: addi $sp, $sp, -12
 34: sw   $ra, 8($sp)
 38: sw   $a0, 4($sp)
 3c: sw   $s0, 0($sp)
 40: addi $s0, $zero, 3
 44: addi $a0, $zero, 4
 48: addi $a1, $zero, 3
 4c: jal  purple
 50: add  $t0, $s0, $v0
 54: lw   $s0, 0($sp)
 58: lw   $a0, 4($sp)
 5c: lw   $ra, 8($sp)
 60: add  $v0, $a0, $t0
 64: addi $sp, $sp, 12
 68: jr   $ra
```

Non-leaf Procedure Example



Main memory

0xbffeffee

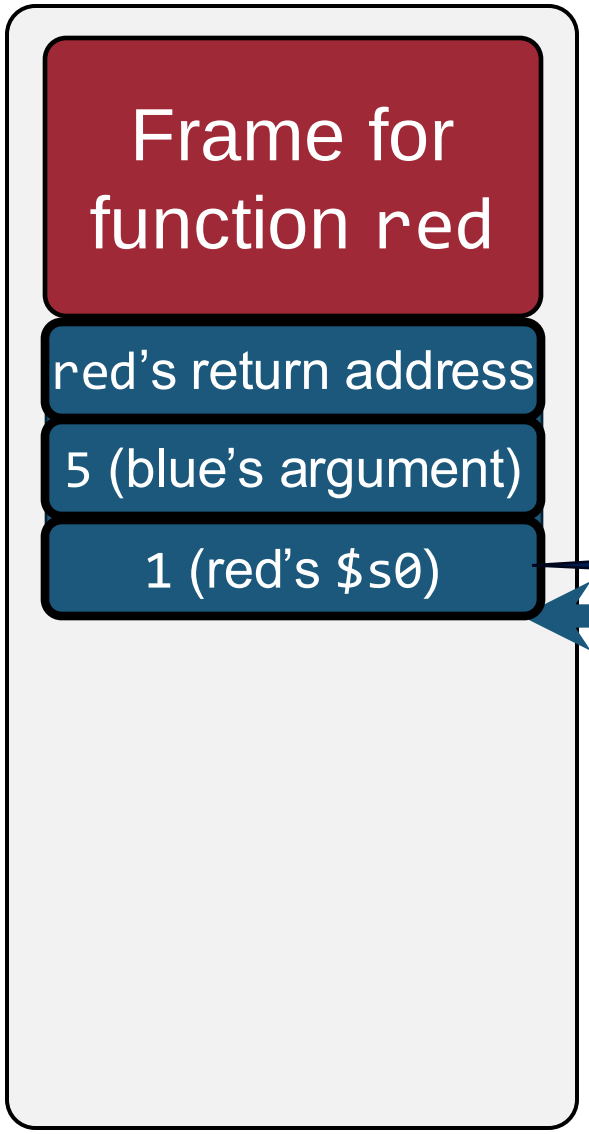
$a0: 0x5 (blue's arg)
pc: 0x40
$sp: 0xbffeffee
$s0: 0x1 (red's $s0)

Execution context

Since $s0 will be **newly used in blue**, it is backed up

```
blue:
 30: addi $sp, $sp, -12
 34: sw   $ra, 8($sp)
 38: sw   $a0, 4($sp)
 3c: sw   $s0, 0($sp)
 40: addi $s0, $zero, 3
 44: addi $a0, $zero, 4
                  ero, 3
 50: add  $t0, $s0, $v0
 54: lw   $s0, 0($sp)
 58: lw   $a0, 4($sp)
 5c: lw   $ra, 8($sp)
 60: add  $v0, $a0, $t0
 64: addi $sp, $sp, 12
 68: jr   $ra
```

Non-leaf Procedure Example



Main memory

```
int blue(int a) {
    int b = 3;
    int c = purple(4, 3)
    return a + b + c;
}
```

0xbffeffee

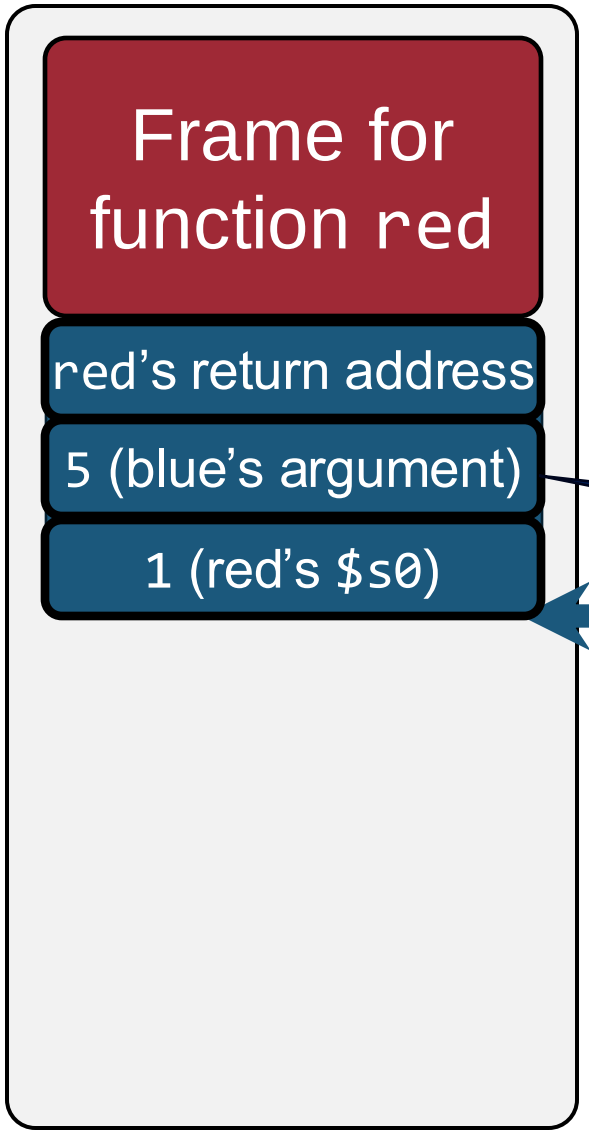
```
$a0:  0x5 (blue's arg)
pc:  0x44
$sp:  0xbffeffe
$s0:  0x3 (blue's $s0)
```

Execution context

```
blue:
30: addi $sp, $sp, -12
34: sw   $ra, 8($sp)
38: sw   $a0, 4($sp)
3c: sw   $s0, 0($sp)
40: addi $s0, $zero, 3
44: addi $a0, $zero, 4
                ro, 3
                , $v0
                sp)
    lw   $a0, 4($sp)
5c: lw   $ra, 8($sp)
60: add  $v0, $a0, $t0
64: addi $sp, $sp, 12
68: jr   $ra
```

red's $s0 is overwritten, but it is preserved from stack

Non-leaf Procedure Example



Main memory

0xbffeffee

```
int blue(int a) {
    int b = 3;
    int c = purple(4, 3)
    return a + b + c;
}
```

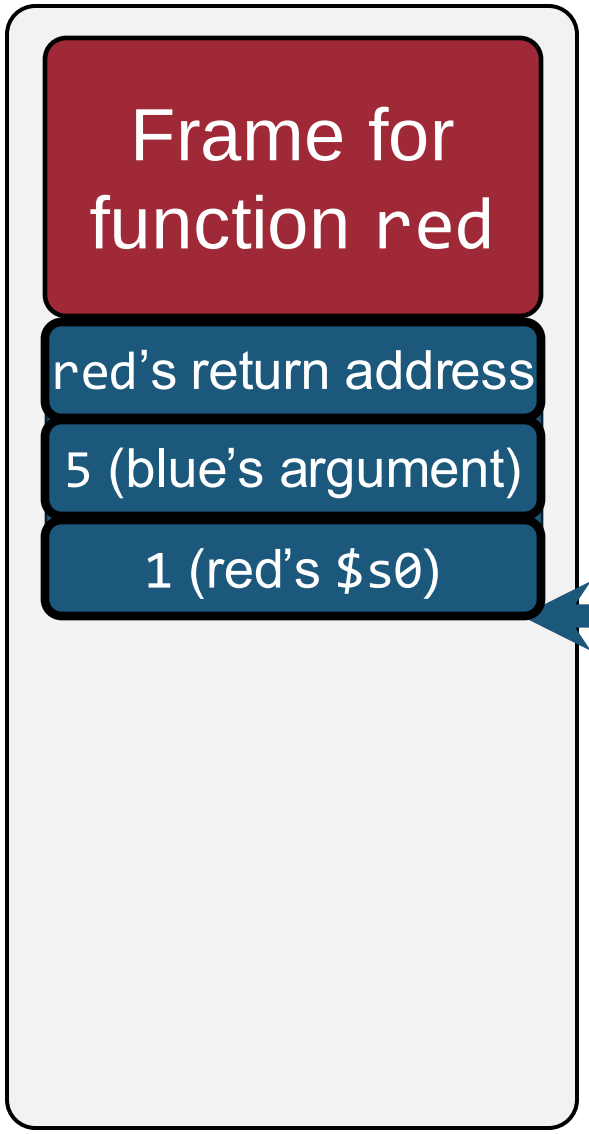
$a0: 0x4 (purple's arg)
pc: 0x48
$sp: 0xbffeffee
$s0: 0x3 (blue's $s0)

Execution context

```
blue:
30: addi $sp, $sp, -12
34: sw   $ra, 8($sp)
38: sw   $a0, 4($sp)
3c: sw   $s0, 0($sp)
40: addi $s0, $zero, 3
44: addi $a0, $zero, 4
48: addi $a1, $zero, 3
5c: lw   $ra, 8($sp)
60: add  $v0, $a0, $t0
64: addi $sp, $sp, 12
68: jr   $ra
```

Blue's argument is overwritten, but it is preserved from stack

Non-leaf Procedure Example



Main memory

```
int blue(int a) {
    int b = 3;
    int c = purple(4, 3)
    return a + b + c;
}
```

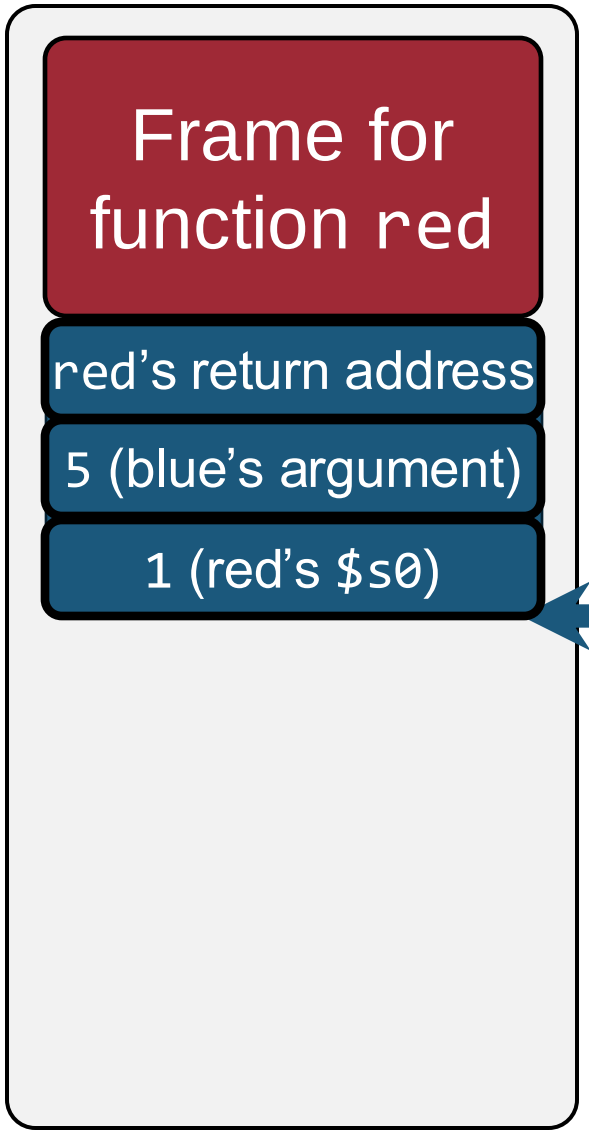
0xbffeffee

$a1: 0x3 (purple's arg)
$a0: 0x4 (purple's arg)
pc: 0x4c
$sp: 0xbffeffee
$s0: 0x3 (blue's $s0)

Execution context

```
blue:
30: addi $sp, $sp, -12
34: sw   $ra, 8($sp)
38: sw   $a0, 4($sp)
3c: sw   $s0, 0($sp)
40: addi $s0, $zero, 3
44: addi $a0, $zero, 4
48: addi $a1, $zero, 3
4c: jal  purple
50: add  $t0, $s0, $v0
54: lw   $s0, 0($sp)
58: lw   $a0, 4($sp)
5c: lw   $ra, 8($sp)
60: add  $v0, $a0, $t0
64: addi $sp, $sp, 12
68: jr   $ra
```

Non-leaf Procedure Example



Main memory

```
int blue(int a) {
    int b = 3;
    int c = purple(4, 3)
    return a + b + c;
}
```

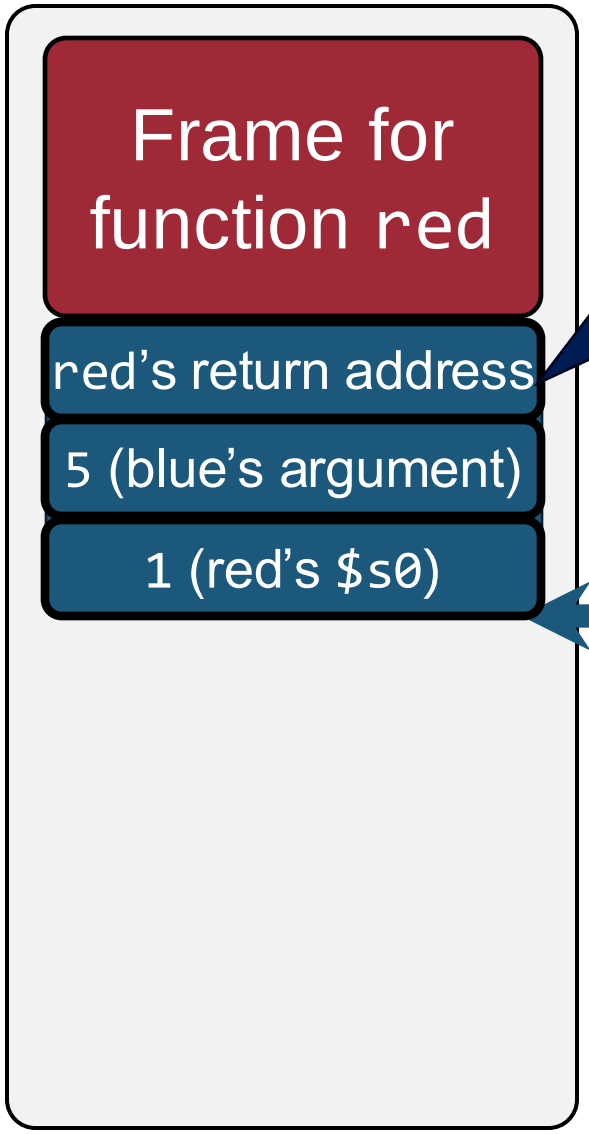
0xbffeffee

$a1:	0x3 (purple's arg)
$a0:	0x4 (purple's arg)
pc:	0x50
$sp:	0xbffeffee
$s0:	0x3 (blue's $s0)

Execution context

```
blue:
30: addi $sp, $sp, -12
34: sw   $ra, 8($sp)
38: sw   $a0, 4($sp)
3c: sw   $s0, 0($sp)
40: addi $s0, $zero, 3
44: addi $a0, $zero, 4
48: addi $a1, $zero, 3
4c: jal  purple
50: add  $t0, $s0, $v0
54: lw   $s0, 0($sp)
58: lw   $a0, 4($sp)
5c: lw   $ra, 8($sp)
60: add  $v0, $a0, $t0
64: addi $sp, $sp, 12
68: jr   $ra
```

Non-leaf Procedure Example



Main memory

red's return address is overwritten, but it is preserved from stack

$ra: 0x50 (blue's ret. addr.)
$a1: 0x3 (purple's arg)
$a0: 0x4 (purple's arg)
pc: 0x14
$sp: 0xbffeffee
$s0: 0x3 (blue's $s0)

Execution context

```
blue:
30: addi $sp, $sp, -12
34: sw   $ra, 8($sp)
   : sw   $a0, 4($sp)
   : sw   $s0, 0($sp)
   : addi $s0, $zero, 3
44: addi $a0, $zero, 4
48: addi $a1, $zero, 3
4c: jal  purple
50: add  $t0, $s0, $v0
54: l    $s0, 0($sp)
5
5
60: add  $v0, $a0, $t0
64: addi $sp, $sp, 12
68: jr   $ra
```

$ra = pc
pc = target address

Non-leaf Procedure Example

Frame for function red

red's return address

```
int blue(int a) {
    int b = 3;
    int c = purple(4, 3)
    return a + b + c;
}
```

```
blue:
30: addi $sp, $sp, -12
34: sw   $ra, 8($sp)
38: sw   $a0, 4($sp)
3c: sw   $s0, 0($sp)
40: addi $s0, $zero, 3
44: addi $a0, $zero, 4
```

Now, you can connect to the "Leaf Procedure Example" slides

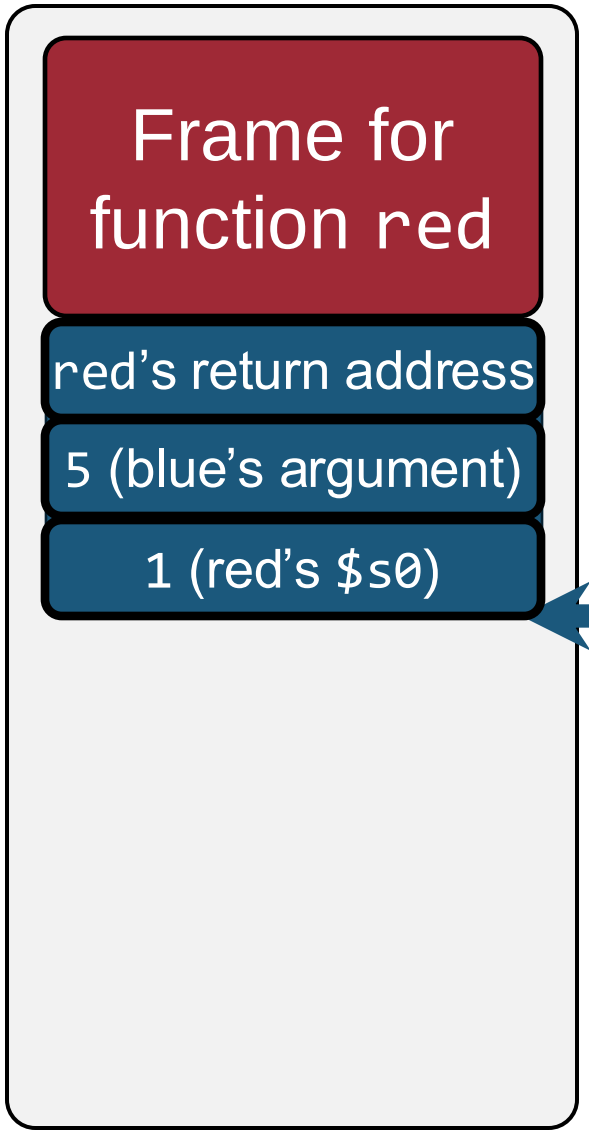
$a0: 0x4 (purple's arg)
pc: 0x50
$sp: 0xbffeffee
$s0: 0x3 (blue's $s0)

Main memory

Execution context

```
58: lw   $a0, 4($sp)
5c: lw   $ra, 8($sp)
60: add  $v0, $a0, $t0
64: addi $sp, $sp, 12
68: jr   $ra
```

Non-leaf Procedure Example



Main memory

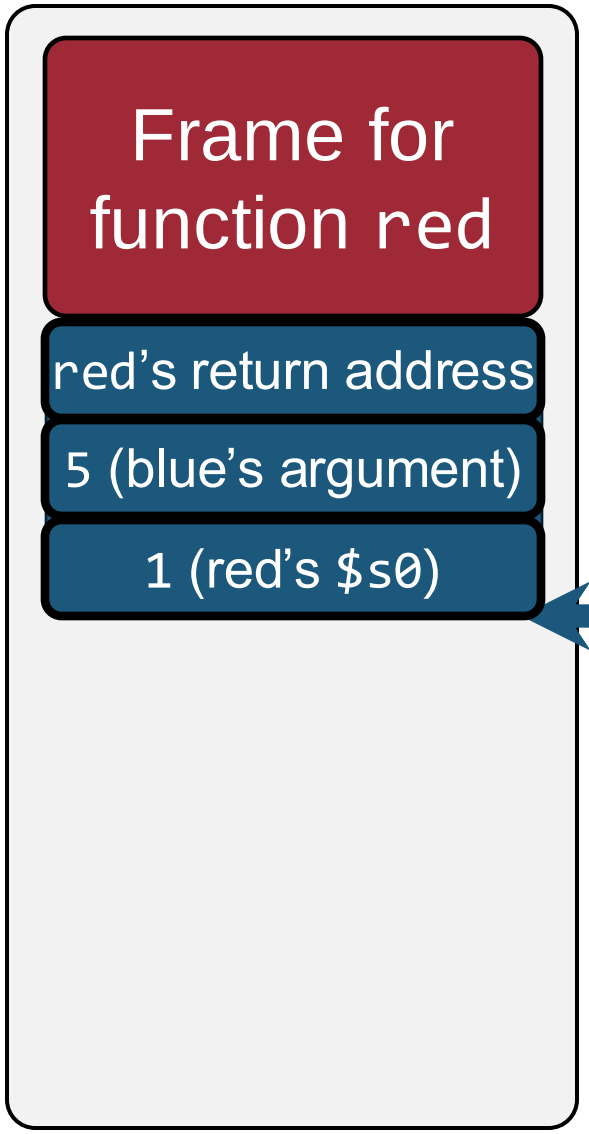
```
int blue(int a) {
    int b = 3;
    int c = purple(4, 3)
    return a + b + c;
}
```

$v0:	0x7
$ra:	0x50 (blue's ret. addr.)
$a1:	0x3 (purple's arg)
$a0:	0x4 (purple's arg)
pc:	0x50
$sp:	0xbffeffee
$s0:	0x3 (blue's $s0)

Execution context

```
blue:
30: addi $sp, $sp, -12
34: sw   $ra, 8($sp)
38: sw   $a0, 4($sp)
3c: sw   $s0, 0($sp)
40: addi $s0, $zero, 3
44: addi $a0, $zero, 4
48: addi $a1, $zero, 3
4c: jal  purple
50: add  $t0, $s0, $v0
54: lw   $s0, 0($sp)
58: lw   $a0, 4($sp)
5c: lw   $ra, 8($sp)
60: add  $v0, $a0, $t0
64: addi $sp, $sp, 12
68: jr   $ra
```

Non-leaf Procedure Example



Main memory

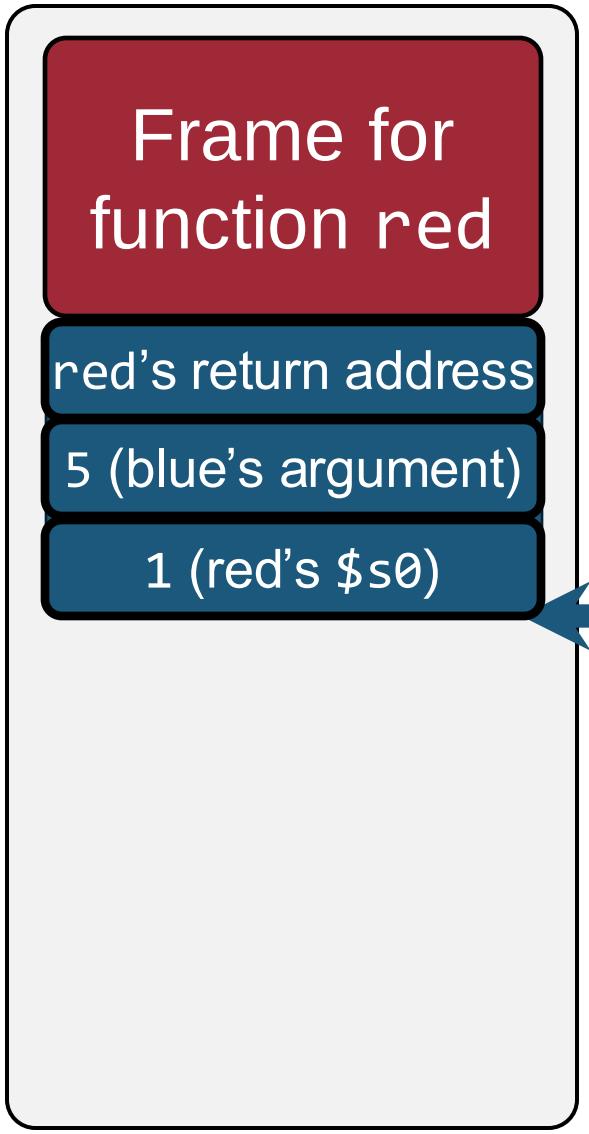
```
int blue(int a) {
    int b = 3;
    int c = purple(4, 3)
    return a + b + c;
}
```

$t0: 0xa
$v0: 0x7
$ra: 0x50 (blue's ret. addr.)
$a1: 0x3 (purple's arg)
$a0: 0x4 (purple's arg)
pc: 0x54
$sp: 0xbffeffee
$s0: 0x3 (blue's $s0)

Execution context

```
blue:
30: addi $sp, $sp, -12
34: sw   $ra, 8($sp)
38: sw   $a0, 4($sp)
3c: sw   $s0, 0($sp)
40: addi $s0, $zero, 3
44: addi $a0, $zero, 4
48: addi $a1, $zero, 3
4c: jal  purple
50: add  $t0, $s0, $v0
54: lw   $s0, 0($sp)
58: lw   $a0, 4($sp)
5c: lw   $ra, 8($sp)
60: add  $v0, $a0, $t0
64: addi $sp, $sp, 12
68: jr   $ra
```

Non-leaf Procedure Example



Main memory

```
int blue(int a) {
    int b = 3;
    int c = purple(4, 3)
    return a + b + c;
}
```

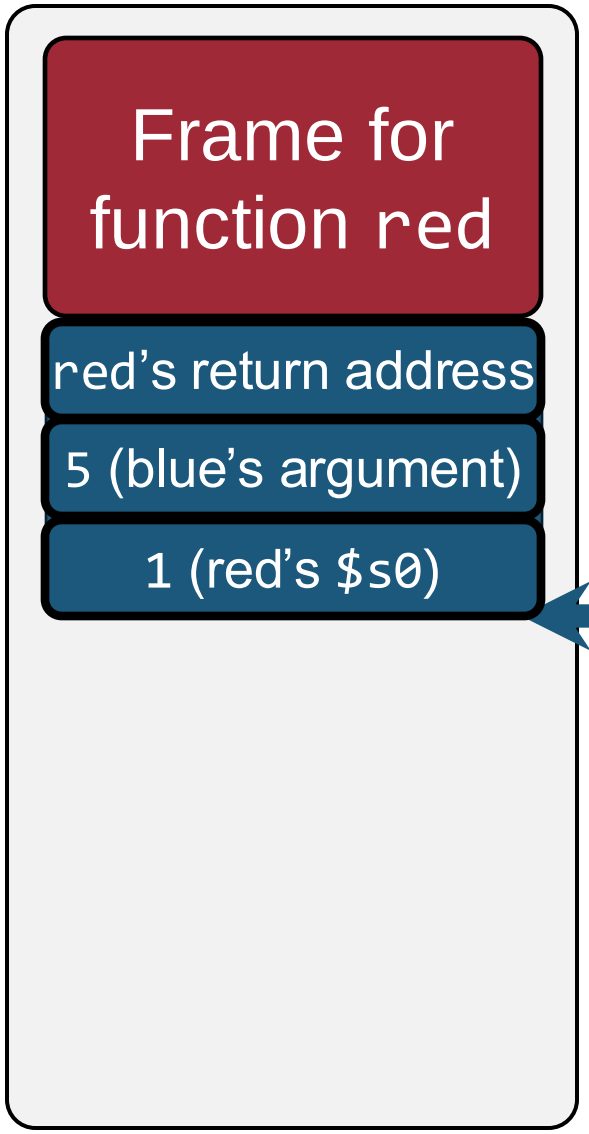
$t0: 0xa
$v0: 0x7
$ra: 0x50 (blue's ret. addr.)
$a1: 0x3 (purple's arg)
$a0: 0x4 (purple's arg)
pc: 0x58
$sp: 0xbffeffee
$s0: 0x3 (blue's $s0)

Execution context

Restore the values from stack!

```
blue:
30: addi $sp, $sp, -12
34: sw   $ra, 8($sp)
38: sw   $a0, 4($sp)
3c: sw   $s0, 0($sp)
40: addi $s0, $zero, 3
     , $zero, 4
     , $zero, 3
     ple
50: add  $t0, $s0, $v0
54: lw   $s0, 0($sp)
58: lw   $a0, 4($sp)
5c: lw   $ra, 8($sp)
60: add  $v0, $a0, $t0
64: addi $sp, $sp, 12
68: jr   $ra
```

Non-leaf Procedure Example



Main memory

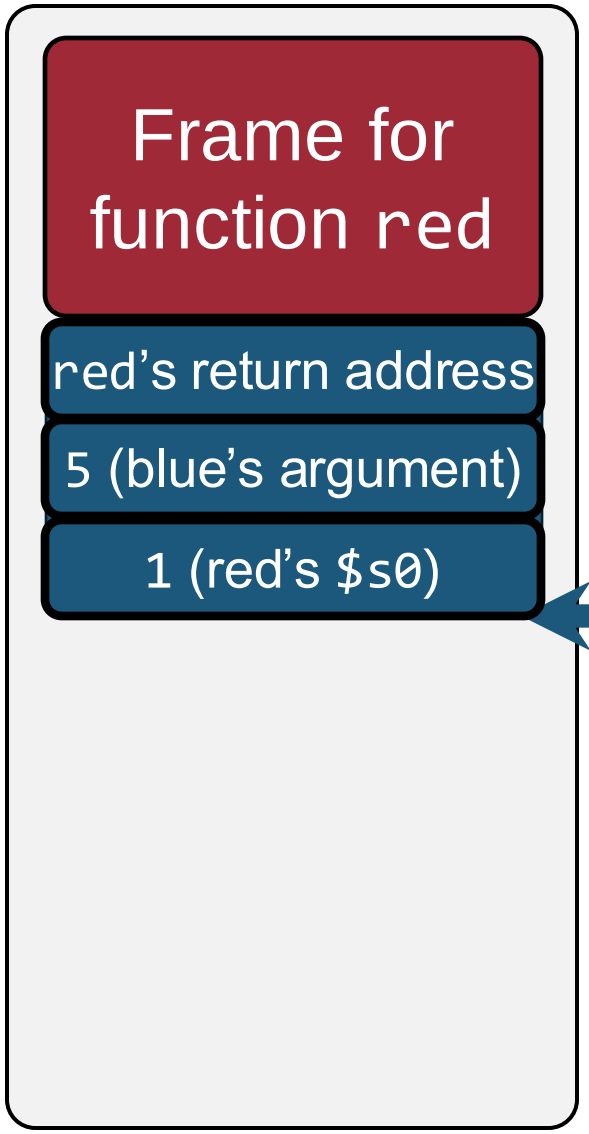
```
int blue(int a) {
    int b = 3;
    int c = purple(4, 3)
    return a + b + c;
}
```

$t0: 0xa
$v0: 0x7
$ra: 0x50 (blue's ret. addr.)
$a1: 0x3 (purple's arg)
$a0: 0x4 (purple's arg)
pc: 0x58
$sp: 0xbffeffee
$s0: 0x1 (red's $s0)

Execution context

```
blue:
30: addi $sp, $sp, -12
34: sw   $ra, 8($sp)
38: sw   $a0, 4($sp)
3c: sw   $s0, 0($sp)
40: addi $s0, $zero, 3
44: addi $a0, $zero, 4
48: addi $a1, $zero, 3
4c: jal  purple
50: add  $t0, $s0, $v0
54: lw   $s0, 0($sp)
58: lw   $a0, 4($sp)
5c: lw   $ra, 8($sp)
60: add  $v0, $a0, $t0
64: addi $sp, $sp, 12
68: jr   $ra
```

Non-leaf Procedure Example



Main memory

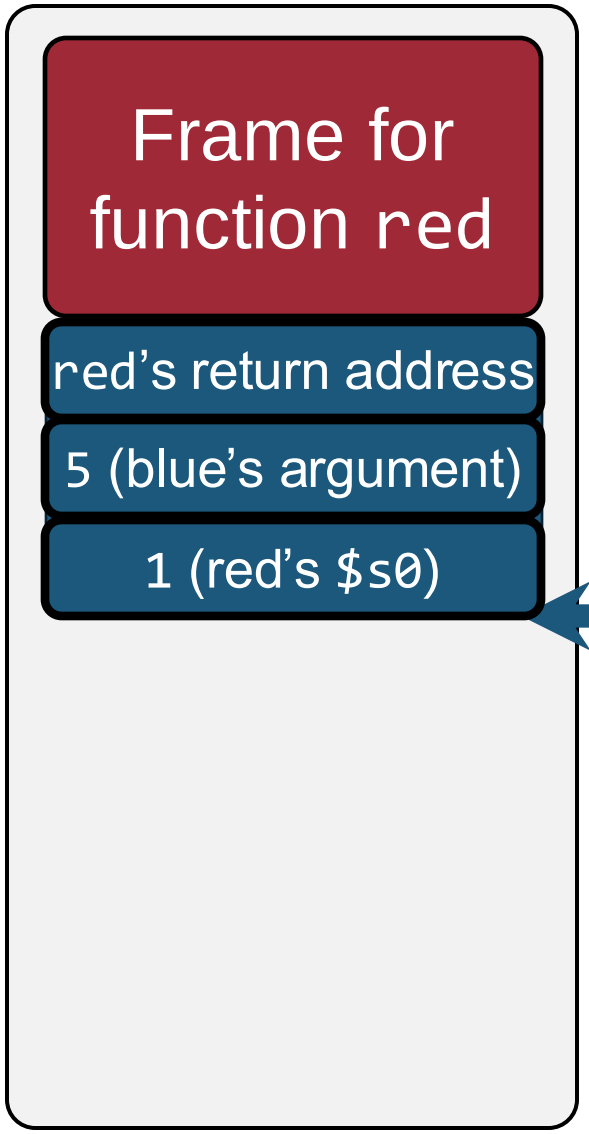
```
int blue(int a) {
    int b = 3;
    int c = purple(4, 3)
    return a + b + c;
}
```

$t0: 0xa
$v0: 0x7
$ra: 0x50 (blue's ret. addr.)
$a1: 0x3 (purple's arg)
$a0: 0x5 (blue's arg)
pc: 0x5c
$sp: 0xbffeffee
$s0: 0x1 (red's $s0)

Execution context

```
blue:
30: addi $sp, $sp, -12
34: sw   $ra, 8($sp)
38: sw   $a0, 4($sp)
3c: sw   $s0, 0($sp)
40: addi $s0, $zero, 3
44: addi $a0, $zero, 4
48: addi $a1, $zero, 3
4c: jal  purple
50: add  $t0, $s0, $v0
54: lw   $s0, 0($sp)
58: lw   $a0, 4($sp)
5c: lw   $ra, 8($sp)
60: add  $v0, $a0, $t0
64: addi $sp, $sp, 12
68: jr   $ra
```

Non-leaf Procedure Example



Main memory

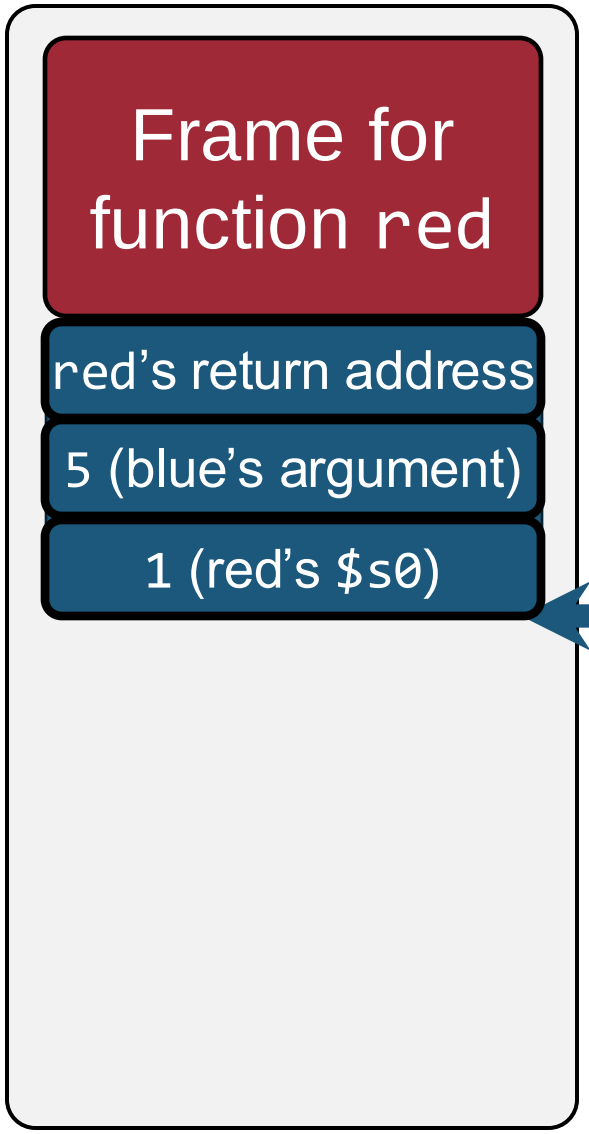
```
int blue(int a) {
    int b = 3;
    int c = purple(4, 3)
    return a + b + c;
}
```

$t0: 0xa
$v0: 0x7
$ra: red's ret. addr.
$a1: 0x3 (purple's arg)
$a0: 0x5 (blue's arg)
pc: 0x60
$sp: 0xbffeffee
$s0: 0x1 (red's $s0)

Execution context

```
blue:
30: addi $sp, $sp, -12
34: sw   $ra, 8($sp)
38: sw   $a0, 4($sp)
3c: sw   $s0, 0($sp)
40: addi $s0, $zero, 3
44: addi $a0, $zero, 4
48: addi $a1, $zero, 3
4c: jal  purple
50: add  $t0, $s0, $v0
54: lw   $s0, 0($sp)
58: lw   $a0, 4($sp)
5c: lw   $ra, 8($sp)
60: add  $v0, $a0, $t0
64: addi $sp, $sp, 12
68: jr   $ra
```

Non-leaf Procedure Example



Main memory

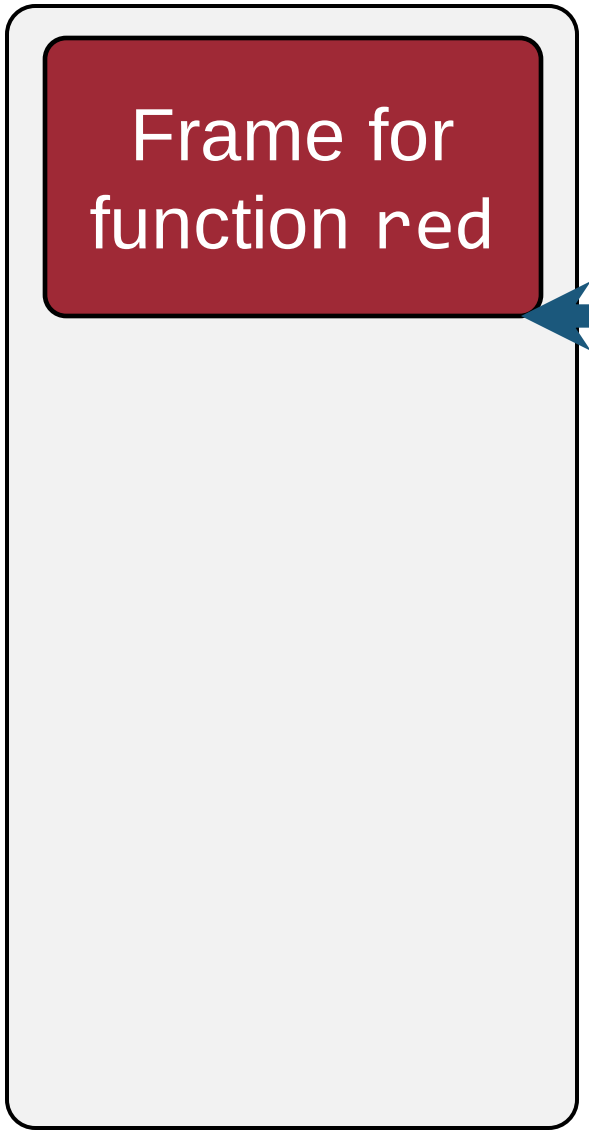
```
int blue(int a) {
    int b = 3;
    int c = purple(4, 3)
    return a + b + c;
}
```

$t0: 0xa
$v0: 0xf
$ra: red's ret. addr.
$a1: 0x3 (purple's arg)
$a0: 0x5 (blue's arg)
pc: 0x64
$sp: 0xbffeffee
$s0: 0x1 (red's $s0)

Execution context

```
blue:
30: addi $sp, $sp, -12
34: sw   $ra, 8($sp)
38: sw   $a0, 4($sp)
3c: sw   $s0, 0($sp)
40: addi $s0, $zero, 3
44: addi $a0, $zero, 4
48: addi $a1, $zero, 3
4c: jal  purple
50: add  $t0, $s0, $v0
54: lw   $s0, 0($sp)
58: lw   $a0, 4($sp)
5c: lw   $ra, 8($sp)
60: add  $v0, $a0, $t0
64: addi $sp, $sp, 12
68: jr   $ra
```

Non-leaf Procedure Example



Main memory

0xbfff0000

```
$t0:  0xa
$v0:  0xf
$ra:  red's ret. addr.
$a1:  0x3 (purple's arg)
$a0:  0x5 (blue's arg)
pc:  0x68
$sp:  0xbfff0000
$s0:  0x1 (red's $s0)
```

Execution context

```
blue:
 30: addi $sp, $sp, -12
 34: sw   $ra, 8($sp)
 38: sw   $a0, 4($sp)
 3c: sw   $s0, 0($sp)
 40: addi $s0, $zero, 3
 44: addi $a0, $zero, 4
 48: addi $a1, $zero, 3
 4c: jal  purple
 50: add  $t0, $s0, $v0
 54: lw   $s0, 0($sp)
 58: lw   $a0, 4($sp)
 5c: lw   $ra, 8($sp)
 60: add  $v0, $a0, $t0
 64: addi $sp, $sp, 12
 68: jr   $ra
```

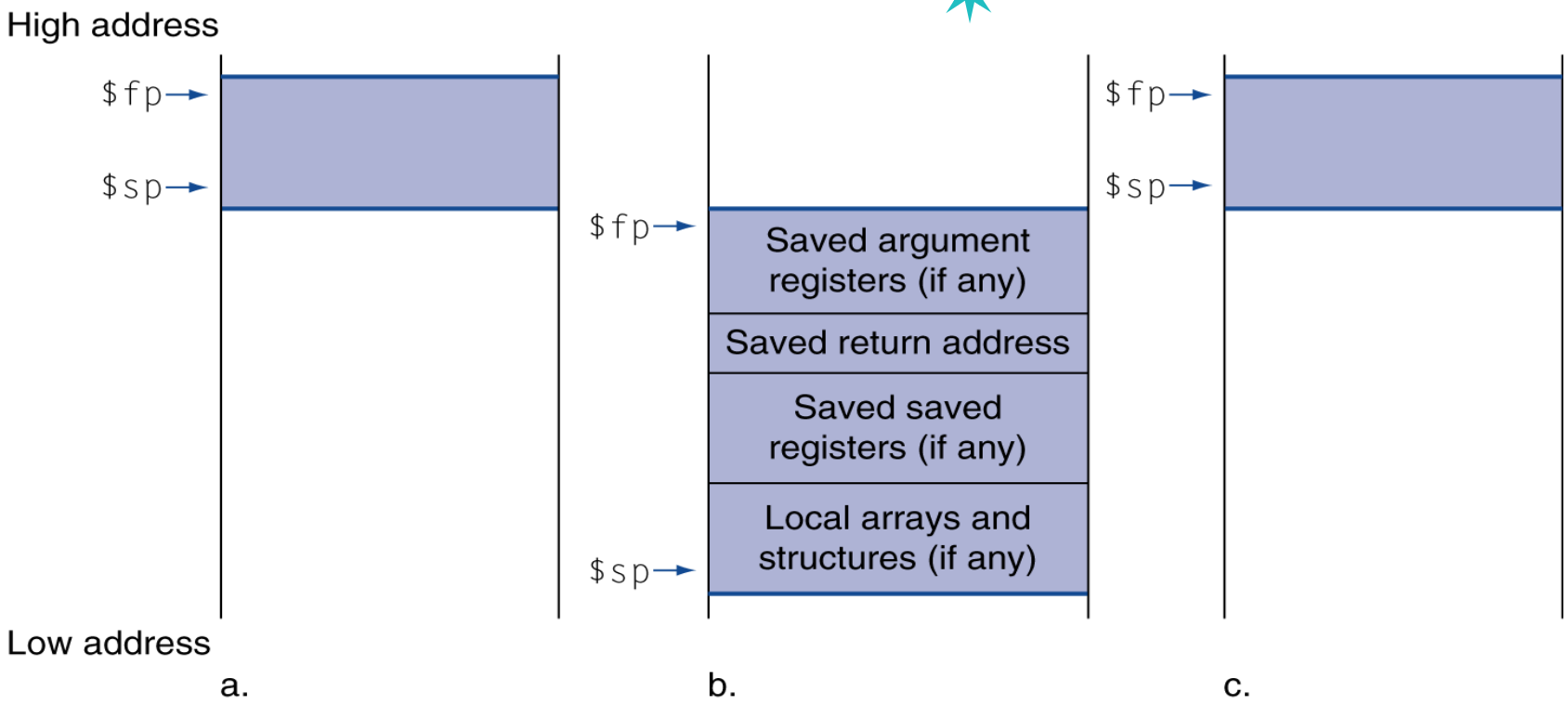
Summary: Procedures Example

```c
int purple(int g, h) {
    int f;
    f = g + h;
    return f;
}

int blue(int a) {
    int b = 3;
    int c = purple(4, 3)
    return a + b + c;
}

int red(int a) {
    return blue(5);
}
```

```
purple:
 14: addi $sp, $sp, -4
 18: sw   $s0, 0($sp)
 1c: add  $s0, $a0, $a1
 20: add  $v0, $s0, $zero
 24: lw   $s0, 0($sp)
 28: addi $sp, $sp, 4
 2c: jr   $ra
```

```
blue:
 30: addi $sp, $sp, -12
 34: sw   $ra, 8($sp)
 38: sw   $a0, 4($sp)
 3c: sw   $s0, 0($sp)
 40: addi $s0, $zero, 3
 44: addi $a0, $zero, 4
 48: addi $a1, $zero, 3
 4c: jal  purple
 50: add  $t0, $s0, $v0
 54: lw   $s0, 0($sp)
 58: lw   $a0, 4($sp)
 5c: lw   $ra, 8($sp)
 60: add  $v0, $a0, $t0
 64: addi $sp, $sp, 12
 68: jr   $ra
```

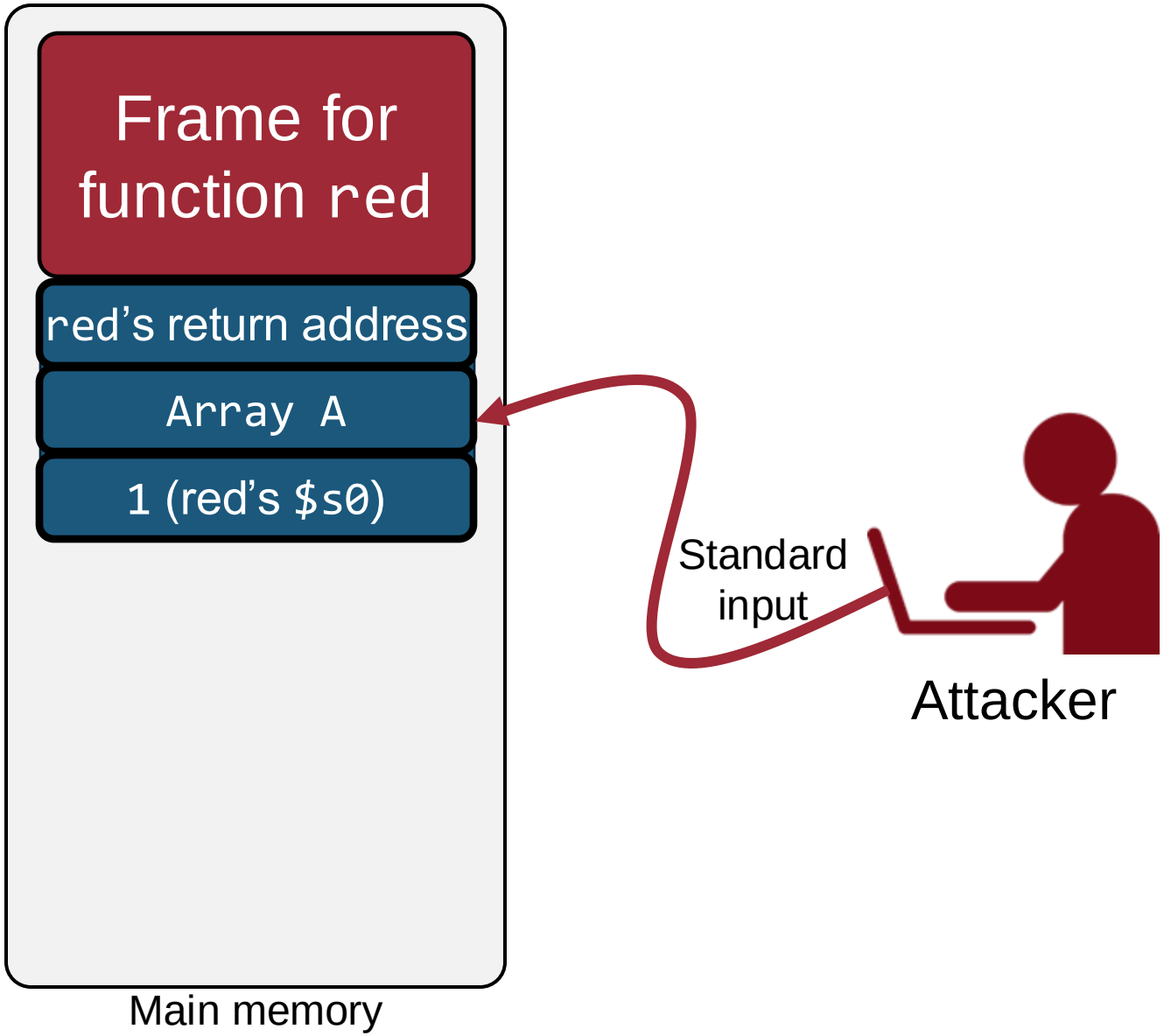
Summary: Data on the Stack



- To hold values passed to a procedure as arguments
- To save registers that a procedure may modify, but which the procedure's caller does not want changed
- To provide space for variables (arrays, structures)

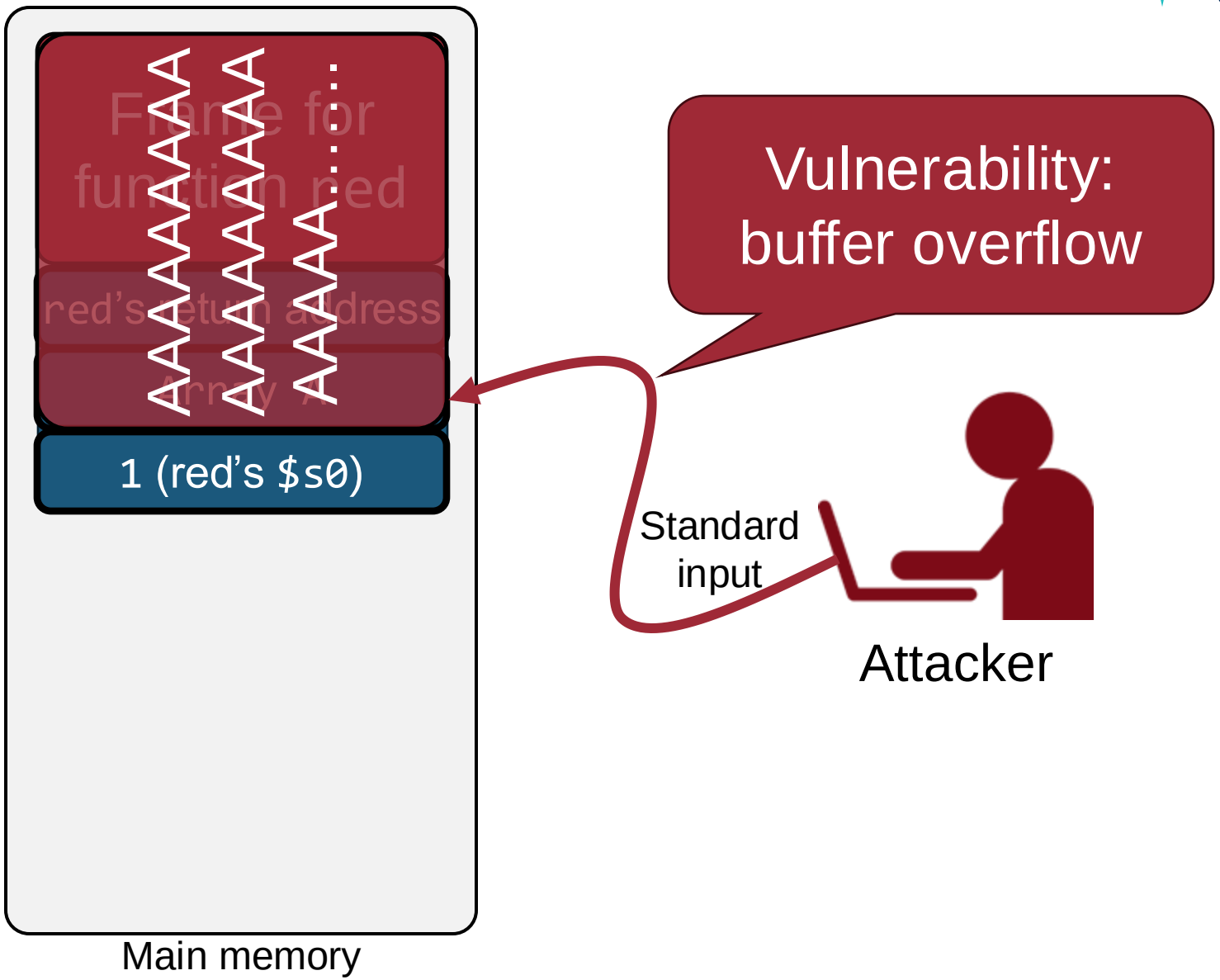
Fun: Control Flow Hijack in Computer Security

(Out-of-scope of the class)



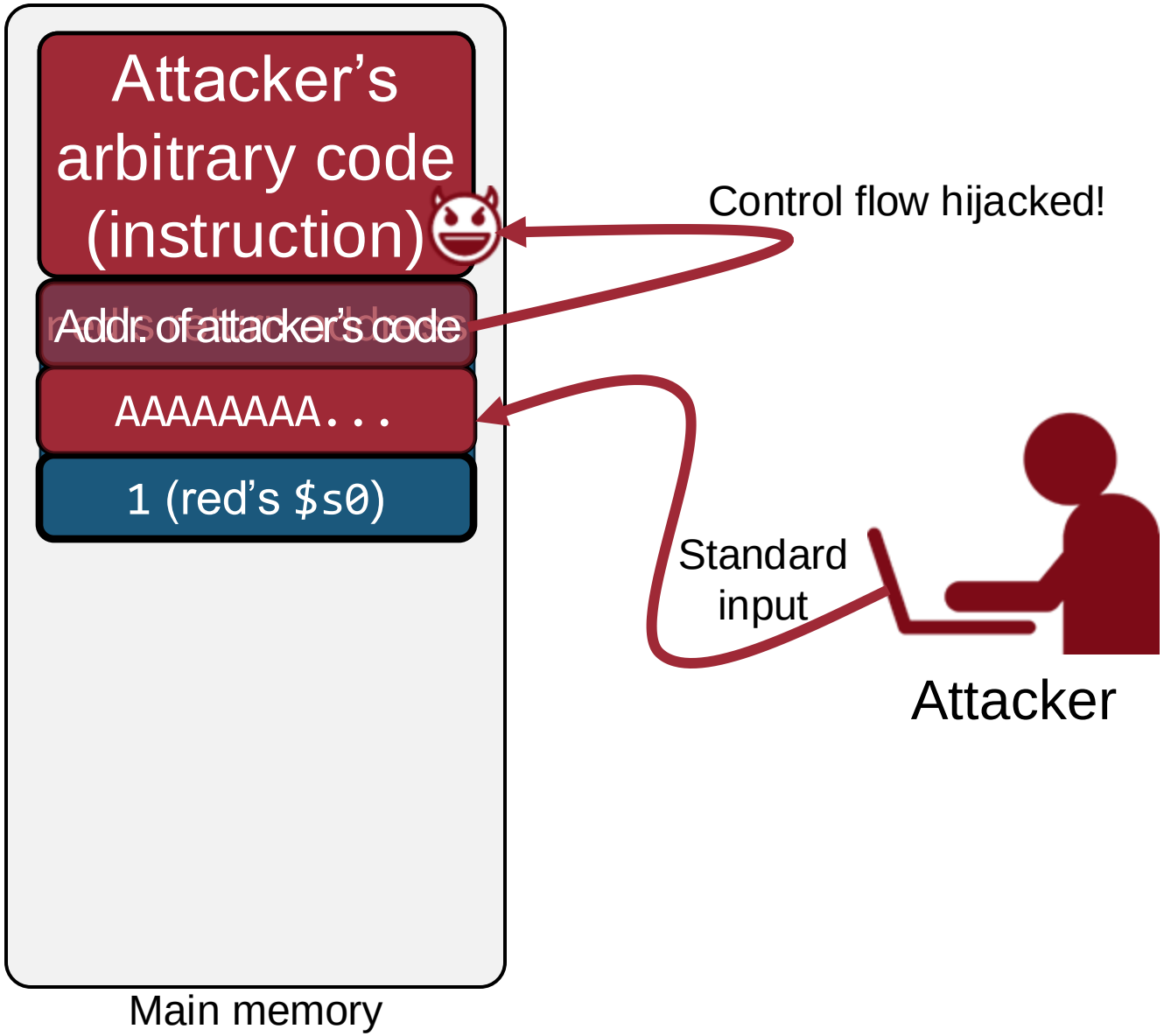
Fun: Control Flow Hijack in Computer Security

(Out-of-scope of the class)



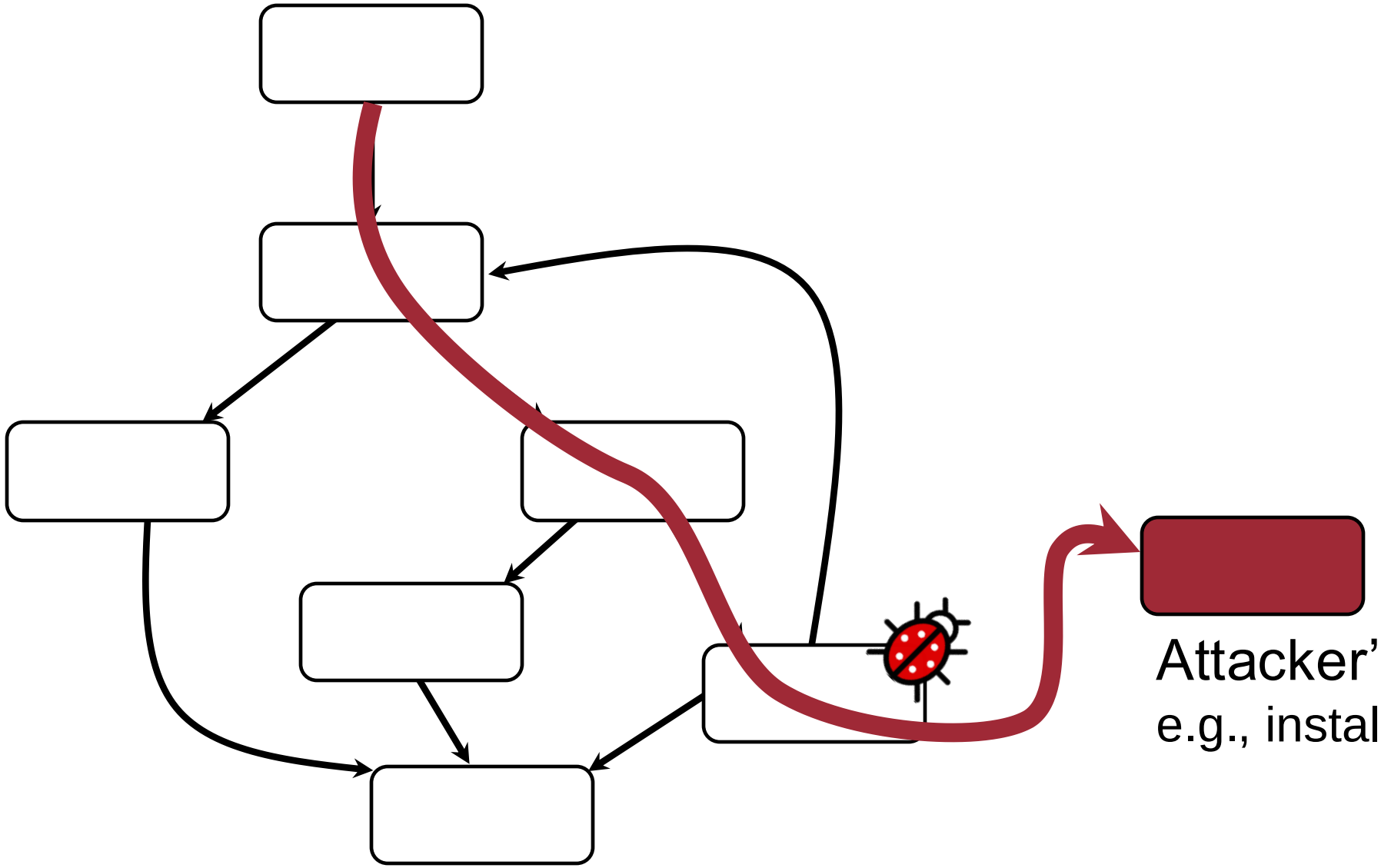
Fun: Control Flow Hijack in Computer Security

(Out-of-scope of the class)



Fun: Control Flow Hijack in Computer Security

(Out-of-scope of the class)



Attacker's own code/logic

e.g., install malicious software

Recursion Example

- C code:

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

  - Argument `n` in `$a0`
  - Result in `$v0`

Recursion Example

Observe yourselves how the stack grows

- MIPS code:

```
fact:
    addi $sp, $sp, -8      # adjust stack for 2 items
    sw   $ra, 4($sp)       # save return address
    sw   $a0, 0($sp)       # save argument
    slti $t0, $a0, 1       # test for n < 1
    beq  $t0, $zero, L1
    addi $v0, $zero, 1     # if so, result is 1
    addi $sp, $sp, 8       #   pop 2 items from stack
    jr   $ra               #   and return
L1: addi $a0, $a0, -1      # else decrement n
    jal  fact              # recursive call
    lw   $a0, 0($sp)       # restore original n
    lw   $ra, 4($sp)       #   and return address
    addi $sp, $sp, 8       # pop 2 items from stack
    mul  $v0, $a0, $v0     # multiply to get result
    jr   $ra               # and return
```

Function Call Summary

- MIPS ISA features supporting procedure call
  - `$a0-$a3`: argument
  - `$v0-$v1`: return value
  - `$ra`: return address
  - Call instruction: `jal`
  - Return instruction: `jr`

- Stacks are used for preserving registers and return address
  - `$sp`: stack pointer
  - `$fp`: frame pointer → `$fp` is optional and in practice rarely used

Question?