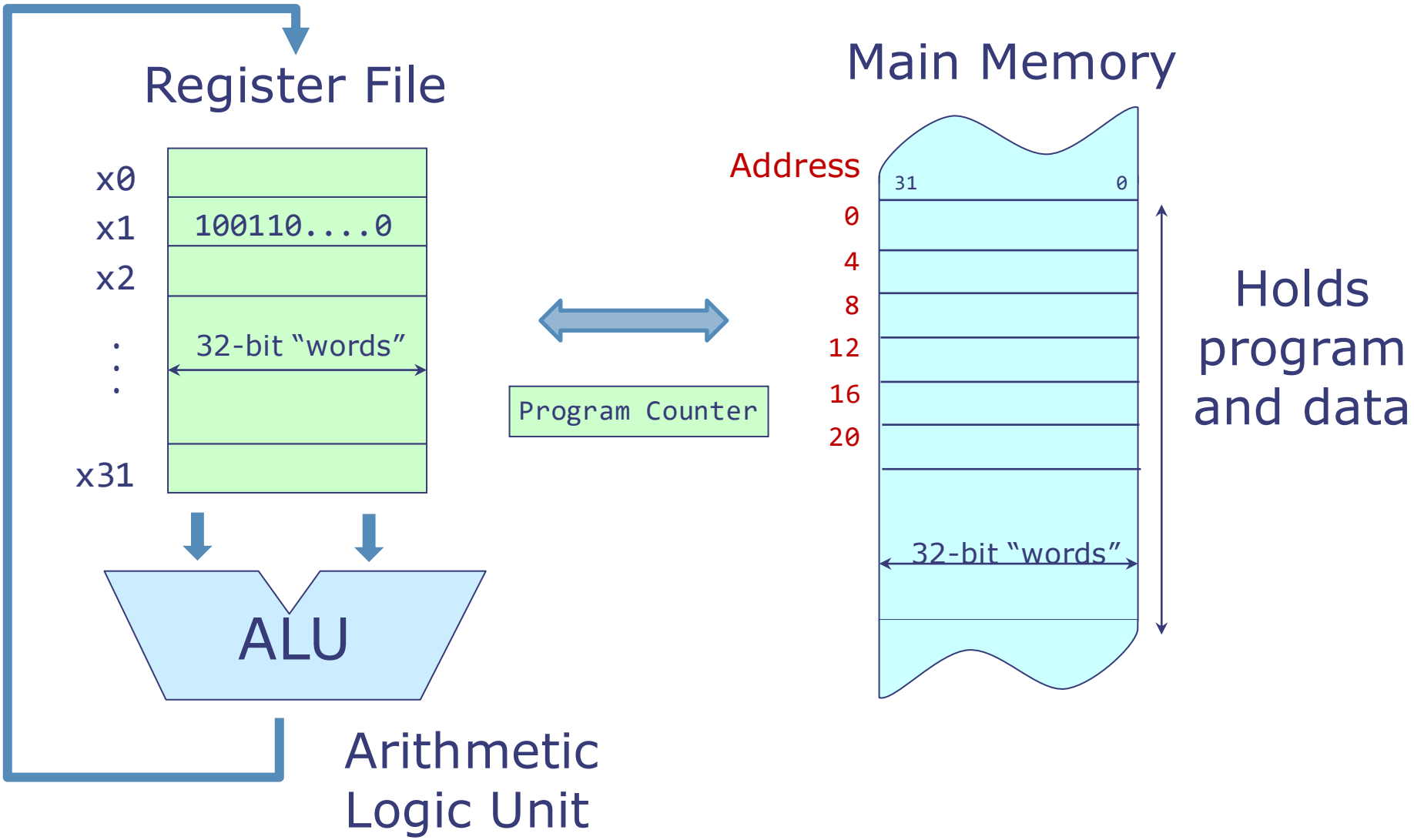
Compiling Code, and Implementing Procedures

Reminders:

- Next week - In lecture quiz 2 will cover lectures 4 and 5

Components of a RISC-V Processor



RISC-V Instruction Types

- Computational Instructions executed by ALU
  - Register-Register: `oper rd, rs1, rs2`
  - Register-Immediate: `oper rd, rs1, constant (12-bit)`
  - `lui rd, luiConstant (20-bit)`
- Loads and Stores
  - `lw rd, offset(rs1)`
  - `sw rs2, offset(rs1)`
  - Memory address = Reg[rs1] + sign_extend(offset)
- Control flow instructions
  - Conditional: `comp rs1, rs2, label`
  - Unconditional: `jal rd, label` and `jalr rd, offset(rs1)`
- Pseudoinstructions
  - Shorthand for other instructions

Pseudoinstructions

- Not "real", but they make our lives easier. Can use freely!
- They are converted into actual RISC-V instructions when being assembled into machine code.

MIT 6.191 (6.004) ISA Reference Card: Pseudoinstructions

Pseudoinstruction	Description	Execution
`li rd, liConstant`	Load Immediate	`reg[rd] <= liConstant`
`mv rd, rs1`	Move	`reg[rd] <= reg[rs1] + 0`
`not rd, rs1`	Logical Not	`reg[rd] <= reg[rs1] ^ -1`
`neg rd, rs1`	Arithmetic Negation	`reg[rd] <= 0 - reg[rs1]`
`j label`	Jump	`pc <= label`
`jal label` `call label`	Jump and Link (with `ra`)	`reg[ra] <= pc + 4` `pc <= label`
`jr rs`	Jump Register	`pc <= reg[rs1] & ~1`
`jalr rs`	Jump and Link Register (with `ra`)	`reg[ra] <= pc + 4` `pc <= reg[rs1] & ~1`
`ret`	Return from Subroutine	`pc <= reg[ra]`
`bgt rs1, rs2, label`	Branch $>$ (Signed)	`pc <= (reg[rs1] >s reg[rs2]) ? label : pc + 4`
`ble rs1, rs2, label`	Branch $\leq$ (Signed)	`pc <= (reg[rs1] <=s reg[rs2]) ? label : pc + 4`
`bgtu rs1, rs2, label`	Branch $>$ (Unsigned)	`pc <= (reg[rs1] >s reg[rs2]) ? label : pc + 4`
`bleu rs1, rs2, label`	Branch $\leq$ (Unsigned)	`pc <= (reg[rs1] <=s reg[rs2]) ? label : pc + 4`
`beqz rs1, label`	Branch $= 0$	`pc <= (reg[rs1] == 0) ? label : pc + 4`
`bnez rs1, label`	Branch $\neq 0$	`pc <= (reg[rs1] != 0) ? label : pc + 4`
`bltz rs1, label`	Branch < 0 (Signed)	`pc <= (reg[rs1] <s 0) ? label : pc + 4`
`bgez rs1, label`	Branch ≥ 0 (Signed)	`pc <= (reg[rs1] >=s 0) ? label : pc + 4`
`bgtz rs1, label`	Branch > 0 (Signed)	`pc <= (reg[rs1] >s 0) ? label : pc + 4`
`blez rs1, label`	Branch ≤ 0 (Signed)	`pc <= (reg[rs1] <=s 0) ? label : pc + 4`

Pseudoinstructions

- Psuedoinstructions are converted to actual RISC-V instructions.

Pseudoinstruction:	Equivalent Assembly Instruction:
`mv x2, x1`	`addi x2, x1, 0`
`ble x1, x2, label`	`bge x2, x1, label`
`j label`	`jal x0, label`

Pseudoinstructions

- li pseudoinstruction translates into either one or two instructions depending on size of constant.

Pseudoinstruction:	Equivalent Assembly Instruction:
`li x2, 3`	`addi x2, x0, 3`
`li x3, 0x4321`	`lui x3, 0x4` `addi x3, x3, 0x321`

Data Memory vs. Instruction Memory

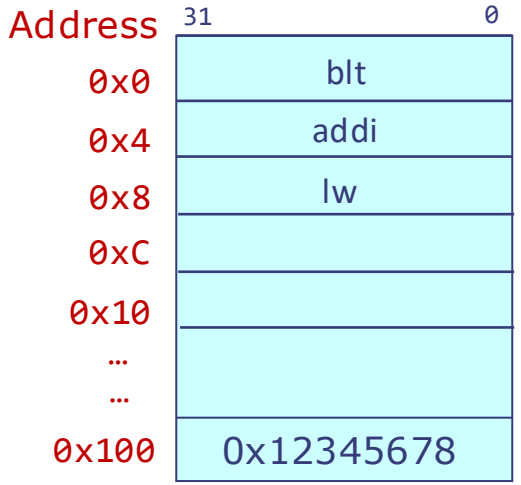
- Instructions (and data) are stored as 32-bit binary values in memory.
- Data will live in one part of the memory.
- The program (instructions) will live in another part of the memory
- Instruction memory is usually write-protected (to avoid changing program during run-time)
  - lw and sw operations access data memory

Review: Program Counter

- The program counter (pc) register keeps track of the address of your next instruction
  - Default (non control-flow instructions): pc = pc + 4
  - Control flow instructions might update pc by a different amount.

```
. = 0x0
   blt x1, x2, label        # if x1 < x2: pc = 0x8
   addi x1, x1, 1           # else: pc = pc + 4
label:
   lw x3, 0x100(x0)         # x3 = 0x12345678

. = 0x100
   .word 0x12345678
```



Reminder: Let's Sum an Array

Assume we have an array arr with 10 integers starting at memory address 0x700.

C Program

```
int i = 0;
int arr_sum = 0;
int arr_len = 10;
while (i < arr_len) {
  int val = *(arr + i);
  arr_sum += val;
  i++;
}
```

Assembly

```
  addi x5, x0, 0
  addi x6, x0, 0
  addi x7, x0, 40
loop:
  bge x5, x7, end
  lw x8, 0x700(x5)
  add x6, x6, x8
  addi x5, x5, 4
  jal x0, loop
end:
```

Let's calculate the GCD of two numbers!

GCD: Greatest common divisor

C code

```
while (x != y) {
    if (x > y) {
    x = x – y;
  } else {
    y = y – x;
  }
}
```

Compiling Conditional Statements

C code

```
if (expr) {
  if-code;
} else {
  else-code;
}
```

1. Compile *expr* into *xN*
2. Conditional branch instruction that will jump to *else* when *expr* is 0.
3. Compile *if-code* into assembly.
4. Add unconditional jump past else branch.
5. Compile *else-code* into assembly.

Ex:

```
if (x > y) {
  x = x – y;
} else {
  y = y – x;
}
```

We'll use x10 for x and x11 for y

```
  slt x12, x11, x10
  beq x12, x0, else
  sub x10, x10, x11
  j endif
else:
  sub x11, x11, x10
endif:
```

Compiling Conditional Statements… Another Way

C code

```
if (expr) {
  if-code;
} else {
  else-code;
}
```

1. Conditional branch instruction that will jump when the *opposite* of *expr* is true.
2. Compile *if-code* into assembly.
3. Add unconditional jump past else branch.
4. Compile *else-code* into assembly.

Ex:

```
if (x > y) {
  x = x – y;
} else {
  y = y – x;
}
```

```
  ble x10, x11, else
  sub x10, x10, x11
  j endif
else:
  sub x11, x11, x10
endif:
```

Compiling Loops

C code

```
while (expr) {
  loop-code
}
```

1. Conditional branch instruction that will jump past loop body when *opposite* of *expr* is true.
2. Compile *loop-code* into assembly.
3. Add unconditional jump to top of loop at end of *loop-code*.

```
while (x != y) {
    if (x > y) {
    x = x – y;
  } else {
    y = y – x;
  }
}
```

```
loop:
  beq x10, x11, endloop
  ble x10, x11, else
  sub x10, x10, x11
  j endif
else:
  sub x11, x11, x10
endif:
  j loop
endloop:
```

Compiling Loops… Another Way

C code

```
while (expr) {
  loop-code
}
```

1. Compile *loop-code* into assembly.
2. Put a conditional branch instruction after *loop-code* that jumps to top of loop when *expr* is true.
3. Before starting loop, jump to that conditional branch instruction.

```
while (x != y) {
    if (x > y) {
    x = x – y;
  } else {
    y = y – x;
  }
}
```

```
j compare
loop:
  ble x10, x11, else
  sub x10, x10, x11
  j endif
else:
  sub x11, x11, x10
endif:
compare:
  bne x10, x11, loop
```

Procedures

- What if we wanted to write a gcd function that we could use throughout a program?
- Encapsulate code into a procedure

```
int gcd(int a, int b) {
  int x = a;
  int y = b;
  while (x != y) {
    if (x > y) {
    x = x – y;
   } else {
    y = y – x;
   }
  }
  return x;
}
```

Procedures

- Procedure (a.k.a. function or subroutine): Reusable code fragment that performs a specific task
  - Single named entry point
  - Zero or more formal arguments
  - Local storage
  - Returns to the caller when finished
- Using procedures enables abstraction and reuse
  - Compose large programs from collections of simple procedures

```
int gcd(int a, int b) {
  int x = a;
  int y = b;
  while (x != y) {
    if (x > y) {
      x = x - y;
    } else {
      y = y - x;
    }
  }
  return x;
}

bool coprimes(int a, int b) {
  return gcd(a, b) == 1;
}

coprimes(5, 10); # false
coprimes(9, 10); # true
```

Two Players: Caller and Callee

- When one procedure is called by another, a relationship is created between two procedures:
  - Caller: the procedure that called the other procedure.
  - Callee: the procedure that was called by the caller.

```
int coprimes(int a, int b) {
  return gcd(a, b) == 1;
}
```

- gcd(a,b): coprimes is the caller, and gcd is the callee.

```
void app_main() {
  int a = 5;
  int b = 10;
  int x = coprimes(a, b);
  ...
  ...
}
```

- coprimes(a,b): app_main is the caller, and coprimes is the callee.

A procedure can be both a caller and a callee! Depends on which procedure call we're talking about.

Procedures introduce many questions

- How to transfer control to callee and back to caller?
- How to communicate arguments and return values?
- How to let procedures use more storage than can fit in registers?
- What if the caller and the callee need to use the same register?

Procedures introduce many questions

- How to transfer control to callee and back to caller?
- How to communicate arguments and return values?
- How to let procedures use more storage than can fit in registers?
- What if the caller and the callee need to use the same register?

How to transfer control to callee and back to caller?

- Caller could use an unconditional jump instruction to get to the callee procedure…
  - But the callee could be called from multiple places in a program.
  - So how would the callee know where to return to?

```
proc:
        …
[0x100] j sum
        …
[0x678] j sum
        …
```

```
sum:
  …
  j ??
```

Should sum return to the instruction at address 0x104 or 0x67C?

Return address (ra)

- Register x1 (symbolic name: ra) is used to hold a procedure's *return address*

Symbolic name	Registers	Description
ra	x1	Return address

- Once finished, the callee will use ra to jump back to the correct instruction within the caller procedure.

What does this look like in assembly?

Jump and **Link**

jal: Unconditional jump and link

Format: jal rd, label

- Jump to instruction at label
  - program counter (pc) updates to be address of that instruction
- Link: stored in register rd ← ***NEW THIS WEEK***
  - The *link* is the address of the instruction ***after*** the jal instruction (rd = pc + 4)
  - This is how we remember where we came from!

```
jal x1, label
addi x7, x7, 1
```

1. Jump to instruction following label
2. x1 holds address of addi instruction

How to transfer control to callee?

- jal ra, sum
  - When calling a procedure, we store the link in ra

```
proc:
        …
[0x100] jal ra, sum # ra = 0x104
        …
[0x678] jal ra, sum # ra = 0x67C
        …
```

```
sum:
   …
   j ??
```

Equivalent pseudoinstructions:

- jal sum
- call sum

How can the callee use ra to return back to the caller?

Jump and **Link** Register

jalr: Unconditional jump via register and link

Format: jalr rd, offset(rs1)

- Jump to instruction stored at address Reg[rs1] + offset
  - program counter (pc) updates to be address of that instruction
- Link: stored in register rd ← ***NEW THIS WEEK***
  - The *link* is the address of the instruction ***after*** the jalr instruction (rd = pc + 4)
  - This is how we remember where we came from!

```
jalr x3, 0(x1)
addi x7, x7, 1
```

1. Jump to instruction at memory address in x1
2. x3 holds address of addi instruction

How to transfer control back to caller?

- jalr x0, 0(ra)
  - ra indicates where the program should return to
    - As long as sum was called using jal ra, sum or equivalent, and *ra hasn't been overwritten since.*
  - Store link in x0
    - Effectively discarding it… we have no need to remember that we came from the end of a procedure.

```
proc:
        …
[0x100] jal ra, sum # ra = 0x104
        …
[0x678] jal ra, sum # ra = 0x67C
        …
```

```
sum:
  …
  jalr x0, 0(ra)
```

Equivalent pseudoinstructions:

- jr ra
- ret

Summary: Transferring control between caller and callee

- To call a procedure: jal ra, label
  - or jal label
  - or call label
- To return from a procedure: jalr x0, 0(ra)
  - or jr ra
  - or ret

```
proc:
        …
[0x100] jal ra, sum # ra = 0x104
        …
[0x678] jal ra, sum # ra = 0x67C
        …
```

```
sum:
  …
  jalr x0, 0(ra)
```

Procedures introduce many questions

- How to transfer control to callee and back to caller?
- How to communicate arguments and return values?
- How to let procedures use more storage than can fit in registers?
- What if the caller and the callee need to use the same register?

How to communicate arguments and return values?

- The caller needs to pass arguments to the called procedure (callee), and
- the callee also needs to send results back to the caller.

There are 32 possible registers:

- How will the callee know where to find a specific argument?
- How will the caller know where to find the return value?

We need some rules.

How to communicate arguments and return values?

- A calling convention specifies rules for register usage across procedures
  - RISC-V calling convention gives symbolic names to registers x0-x31 to denote their role
- For example, certain registers are specified to hold function arguments and return values.

Symbolic name	Registers	Description
a0 to a7	x10 to x17	Function arguments
a0 and a1	x10 and x11	Function return values

RISC-V Function Arguments

- Registers a0 – a7 (x10 – x17) are used as function arguments.
  - The first argument goes in a0, second in a1, and so on…
  - If a function has more than 8 arguments, the extra ones will be passed in via memory... more on that in a bit.
- The caller must put function arguments in the necessary registers before calling a procedure.
  - That way the callee can find them!

RISC-V Function Arguments

- The caller is responsible for putting function arguments in the necessary registers before calling a procedure.

```
int coprimes(int a, int b) {
  return gcd(a, b) == 1;
}
```

coprimes responsible for making sure a0 = a, a1 = b before calling gcd...

```
int gcd(int x, int y){
   ...
}
```

... so that gcd can assume a0 = x and a1 = y

RISC-V Return Values

- Registers a0 – a1 (x10 – x11) are used as function return values.
  - The first return value goes in a0, second in a1
  - If a function has more than 2 return values, the extra ones will be returned via specific slots in memory... Stay tuned.
- The callee is responsible for putting return values in the necessary registers before returning.

RISC-V Return Values

- The callee is responsible for putting return values in the necessary registers before returning.

```
int gcd(int x, int y){
  ...
  return val;
}
```

gcd responsible for putting val in a0 before returning…

```
int coprimes(int a, int b) {
  return gcd(a, b) == 1;
}
```

…so coprimes can assume gcd(a,b) will be in a0, ready to use

Responsibility enables assumptions

- The caller doesn't know the implementation details of the callee, and vice versa.

- In order for them to be able to work together, they must each follow the shared set of rules defined by the *calling convention*.

Procedures introduce many questions

- How to transfer control to callee and back to caller?
- How to communicate arguments and return values?
- How to let procedures use more storage than can fit in registers?
- What if the caller and the callee need to use the same register?

How to let procedures use more storage than can fit in registers?

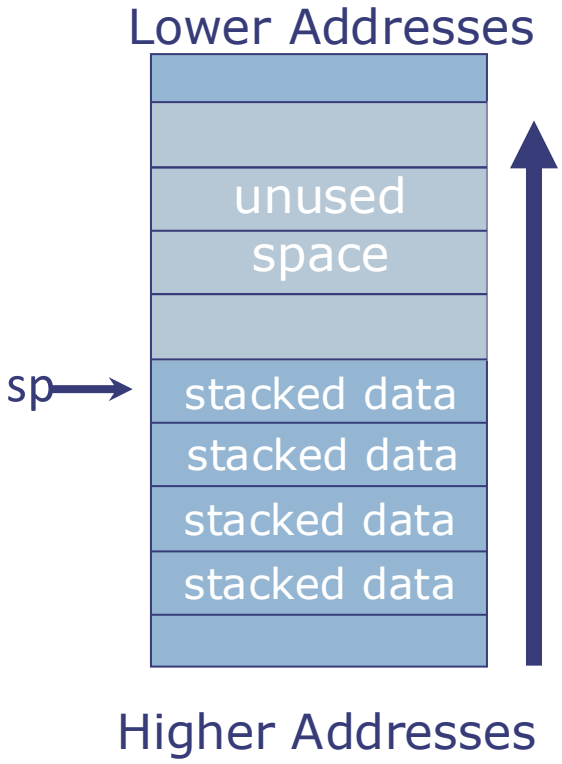
- There are only 32 registers available.
  - All procedures need to be able to use the same 32 registers – will need to "free them up" in order to share them.
  - But what if we run out of registers?

- Also, there are only so many registers dedicated to certain things.
  - Ex: 8 registers are dedicated to holding function arguments… but what happens if a procedure has 9 arguments?

Short answer: use a special section of memory.

RISC-V Stack

- Section of memory
  - Grows *down* from higher to lower addresses
  - sp register points to the top of the stack



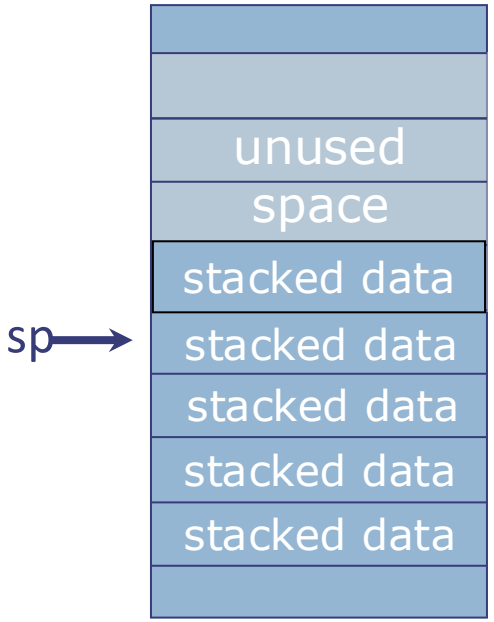
Using the stack

- Two operations: push and pop
  - Push: put something on top of stack

```
addi sp, sp, -4 # allocate space
sw a1, 0(sp)    # put element on
```

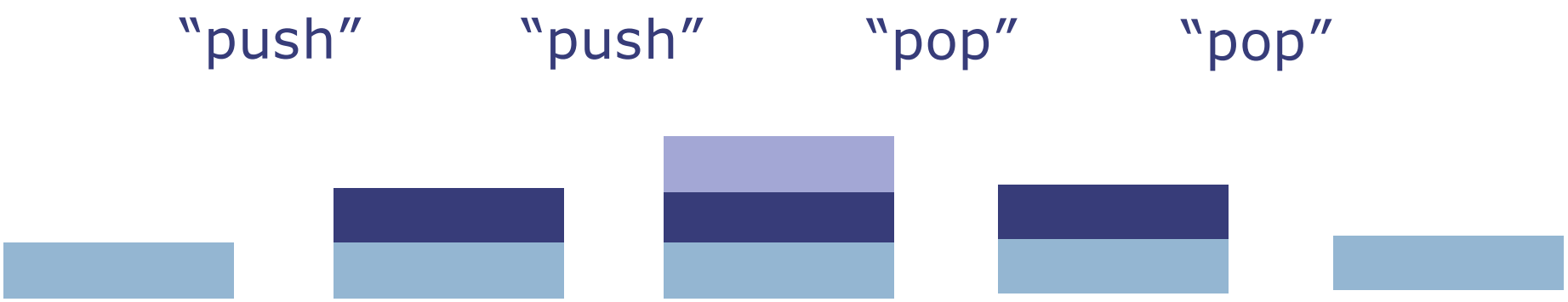
  - Pop: take something off top of stack

```
lw a1, 0(sp)    # take element off
addi sp, sp, 4  # deallocate space
```



Using the Stack

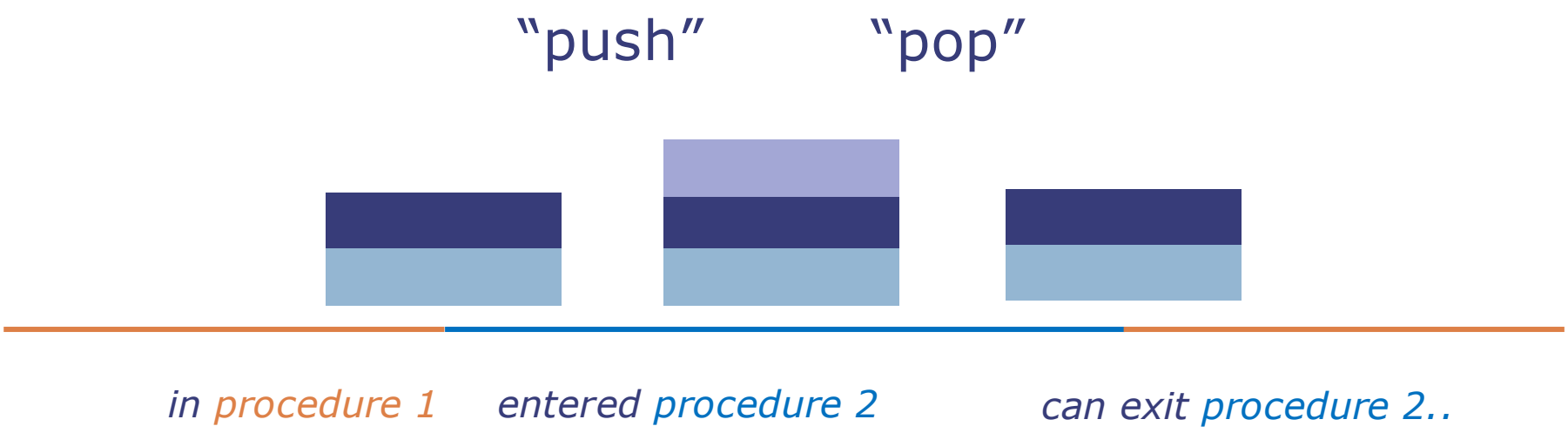
- A program must use the stack in a Last-In-First-Out (LIFO) manner
  - The most recent thing added to the stack will be the next thing taken off.
  - Kind of like a physical stack of things (e.g., books)



Stack Discipline

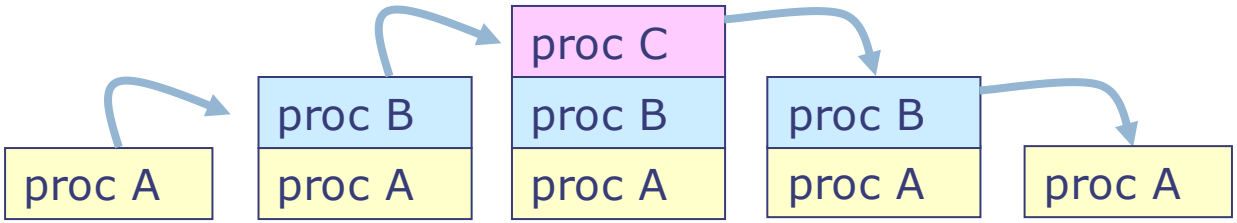


- Any procedure may use the stack, but it must leave the stack as it found it!
  - Anything that goes on the stack must get taken off by the time the procedure returns.
  - sp must be reset to the value that it was at the beginning of the procedure.



How to let a procedure use more memory than can fit in registers?

- An *activation record* holds all storage needs of procedure that do not fit in registers
  - A new activation record is put on the stack when a procedure is called
  - That activation record is taken off the stack when that procedure returns.
- Activation records go on the stack (Last-In-First-Out)



- The current procedure's activation record (a.k.a. stack frame) is always at the top of the stack

Procedures introduce many questions

- How to transfer control to callee and back to caller?
- How to communicate arguments and return values?
- How to let procedures use more storage than can fit in registers?
- What if the caller and the callee need to use the same register?

What if the caller and callee need to use the same register(s)?

- The caller procedure and the callee procedure use the same 32 registers.
  - The caller doesn't know how the callee procedure uses registers. And vice versa.
  - Both the caller and the callee should be able to use all 32 registers.

RISC-V calling convention specifies rules for sharing registers among different procedures.

Calling Convention

- If a register will (potentially) be overwritten across a function call, its value must be saved on the stack.
  - The caller procedure is responsible for saving some registers…
  - …and the callee procedure is responsible for saving others.
- RISC-V ISA tells us which ones are which!

Registers	Symbolic names	Description	Saver
x0	zero	Hardwired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5-x7	t0-t2	Temporary registers	Caller
x8-x9	s0-s1	Saved registers	Callee
x10-x11	a0-a1	Function arguments and return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporary registers	Caller

Calling Convention

- RISC-V calling convention gives symbolic names to registers x0-x31 to denote their role:

Symbolic name	Registers	Description	Saver
a0 to a7	x10 to x17	Function arguments	Caller
a0 and a1	x10 and x11	Function return values	Caller
ra	x1	Return address	Caller
t0 to t6	x5-7, x28-31	Temporaries	Caller
s0 to s11	x8-9, x18-27	Saved registers	Callee
sp	x2	Stack pointer	Callee
gp	x3	Global pointer	---
tp	x4	Thread pointer	---
zero	x0	Hardwired zero	---

Caller-saved Registers (aN, tN, ra)

Registers not preserved across a procedure call.

- The callee can use them freely
- The caller doesn't know what registers the callee will use – must assume it'll use them all.

If the caller needs a caller-saved register's value after the procedure call, it must:

- *Before calling the procedure:* save the register's value on the stack.
- *After returning from the procedure:* restore the old value from the stack before next using the register.

Any callee can use these registers freely because it can safely assume the caller will save any register values it will need.

Callee-saved Registers (sN, sp*)

Preserved across a procedure call!

- The caller can assume that the register's value will be the same before and after the procedure call.

If the callee wants to use a callee-saved register, it must:

- *Before using:* save the register's value on the stack.
- *After using:* restore the old value from the stack before returning to the caller.

*The stack pointer (sp) doesn't go on the stack, just needs to be reset to it's original value.

Any caller can safely assume that these register values will be "untouched" because the callee will make sure to restore them to their original state before returning.

Let's put this all together

Caller:

- Saves any aN, tN, or ra registers ***whose values need to be maintained past the procedure call*** on the stack prior to the procedure call
- Restores them ***before using them again*** after returning from the callee.

Callee:

- Saves original values of sN registers on the stack ***before using them*** in a procedure.
- Must restore sN registers and stack pointer when done using them, ***before exiting procedure***.

Example: Using callee-saved registers

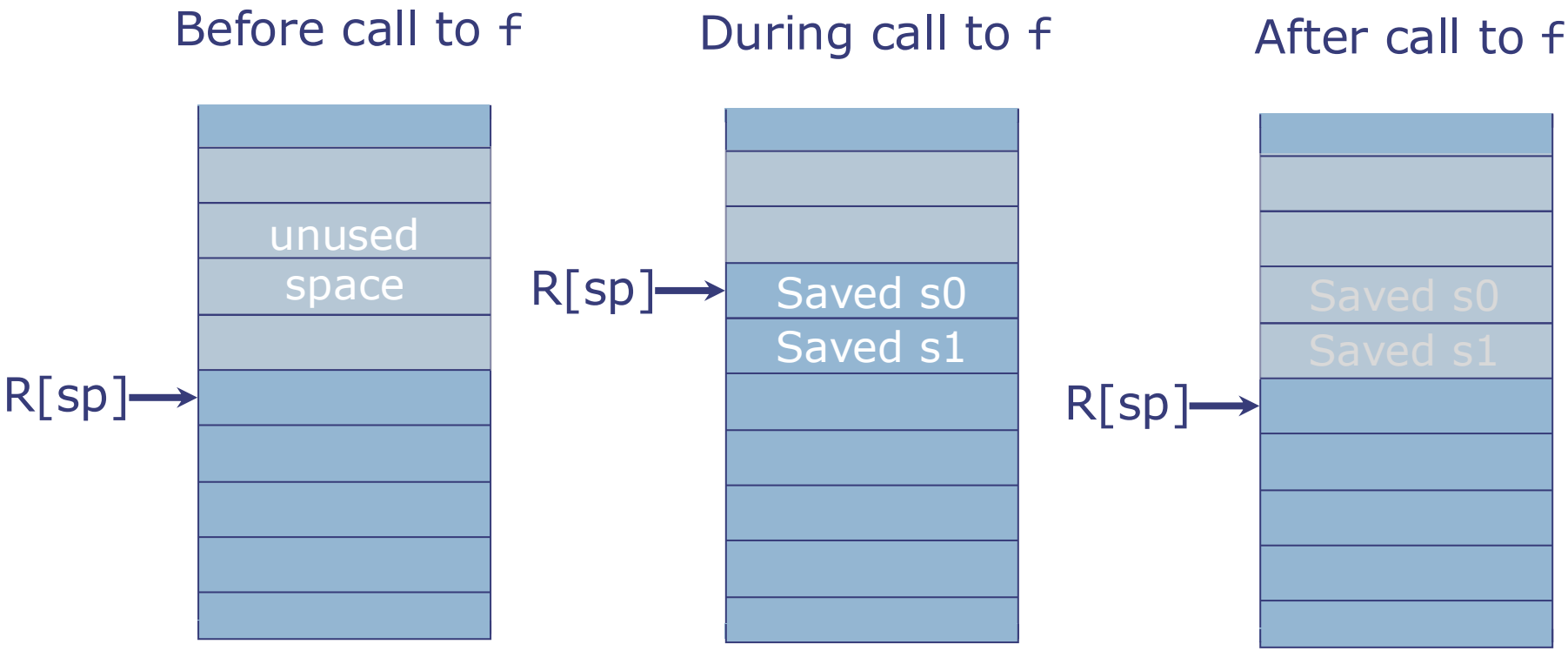
- Implement f using s0 and s1 to store intermediate values

```
int f(int x, int y) {
    return (x + 3)|(y + 0x123456);
}
```

```
f:
  addi sp, sp, -8    # allocate 2 words (8 bytes) on stack
  sw s0, 0(sp)       # save s0
  sw s1, 4(sp)       # save s1
  addi s0, a0, 3
  li s1, 0x123456
  add s1, a1, s1
  or a0, s0, s1
  lw s0, 0(sp)       # restore s0
  lw s1, 4(sp)       # restore s1
  addi sp, sp, 8     # deallocate 2 words from stack
                     # (restore sp)
  ret
```

Example: Using callee-saved registers

- Stack contents:



Example: Using caller-saved registers

Caller

```
int x = 1;
int y = 2;
int z = sum(x, y);
int w = sum(z, y);
```

```
li a0, 1
li a1, 2
addi sp, sp, -8
sw ra, 0(sp)
sw a1, 4(sp) # save y
jal ra, sum
# a0 = sum(x, y) = z
lw a1, 4(sp) # restore y
jal ra, sum
# a0 = sum(z, y) = w
lw ra, 0(sp)
addi sp, sp, 8
```

Callee

```
int sum(int a, int b) {
    return a + b;
}
```

```
sum:
  add a0, a0, a1
  ret
```

Why did we save a1?

Callee may have modified a1 (caller doesn't see implementation of sum!)

Why didn't we save a0?

We don't need the original value after either procedure call.

Nested Procedures

- If a procedure calls another procedure, it needs to save its own return address
  - Remember that `ra` is caller-saved
- Example:

```
int coprimes(int a, int b) {
  return gcd(a, b) == 1;
}
```

```
coprimes:
  addi sp, sp, -4
  sw ra, 0(sp)
  call gcd                    # overwrites ra
  addi a0, a0, -1
  sltiu a0, a0, 1
  lw ra, 0(sp)
  addi sp, sp, 4
  ret                         # where will the program
                              return to?
```

Computing with large data structures

- Suppose we want to write a procedure that finds the maximum value in an array a[].
  - The array is too large to be stored in registers

- How do we pass the array a[] as an argument?
  - Pass the base address (pointer to array) and the size of the array as arguments

Just like we've been doing in C!

Passing Complex Data Structures as Arguments

```
# Finds maximum element in an
# array with size elements
int maximum(int *a, int size) {
   int max = 0;
   for (int i = 0; i < size;
    i++) {
      if (a[i] > max) {
         max = a[i];
      }
   }
   return max;
}

int main() {
   int ages[5] =
      {23, 4, 6, 81, 16};
   int max = maximum(ages, 5);
}
```

Passing Complex Data Structures as Arguments

```
main:   li a0, ages
        li a1, 5
        call maximum
        # max returned in a0
```

```
ages:
        .word 0x17
        .word 0x4
        .word 0x6
        .word 0x51
        .word 0x10
```

```
int main() {
   int ages[5] =
      {23, 4, 6, 81, 16};
   int max = maximum(ages, 5);
}
```

Passing Complex Data Structures as Arguments

```
# Finds maximum element in an
# array with size elements
int maximum(int *a, int size) {
   int max = 0;
   for (int i = 0; i < size;
    i++) {
      if (a[i] > max) {
         max = a[i];
      }
   }
   return max;
}

int main() {
   int ages[5] =
      {23, 4, 6, 81, 16};
   int max = maximum(ages, 5);
}
```

```
maximum:
    mv t0, zero    # t0: i
    mv t1, zero    # t1: max
    j compare
loop:
    slli t2, t0, 2 # t2: i*4
    add t3, a0, t2
    # t3: addr of a[i]=base+i*4
    lw t4, 0(t3)   # t4: a[i]
    ble t4, t1, endif
    mv t1, t4      # max = a[i]
endif:
    addi t0, t0, 1 # i++
compare:
    blt t0, a1, loop

    mv a0, t1      # a0 = max
    ret
```

Thank you!

Next lecture: Dynamic Memory Allocation