

CS 110
Computer Architecture
Lecture 6:
More RISC-V, RISC-V Functions

Instructor:
Sören Schwertfeger

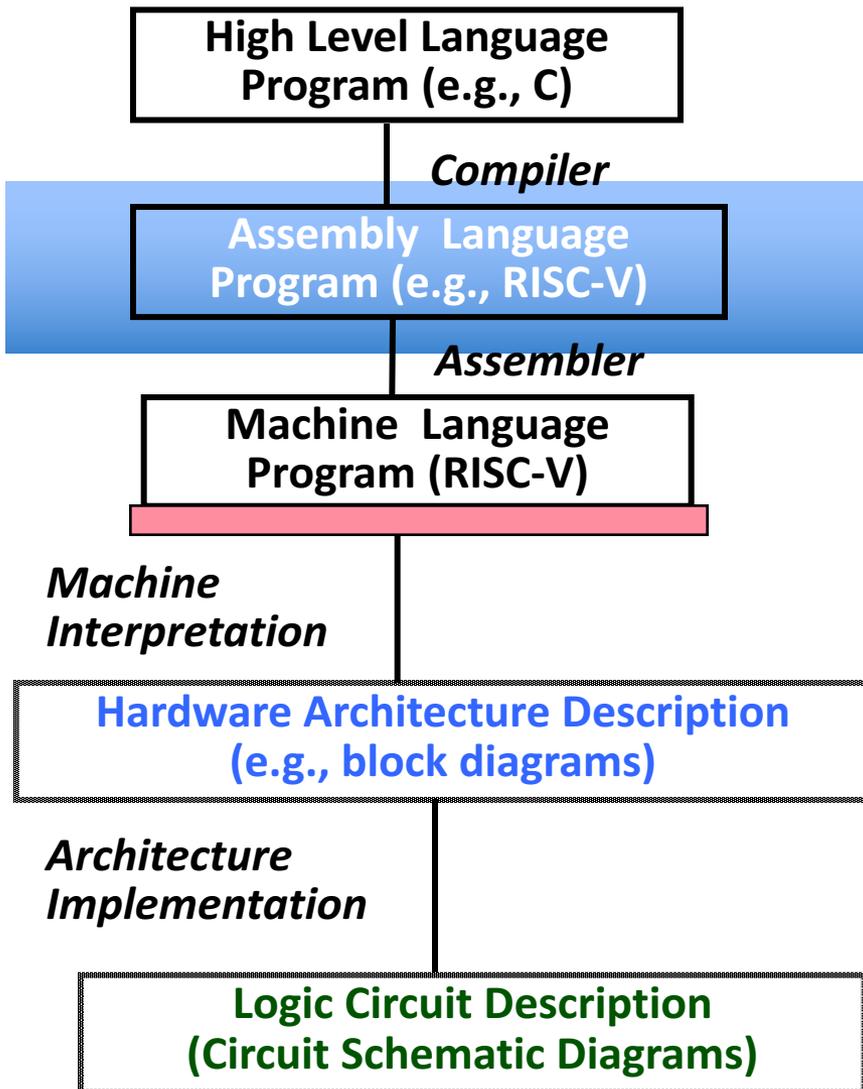
<http://shtech.org/courses/ca/>

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkley's CS61C

Levels of Representation/Interpretation

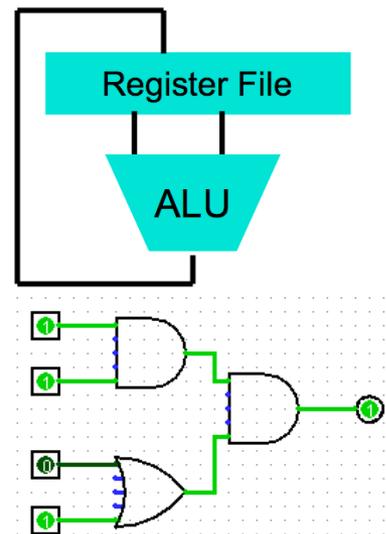


```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

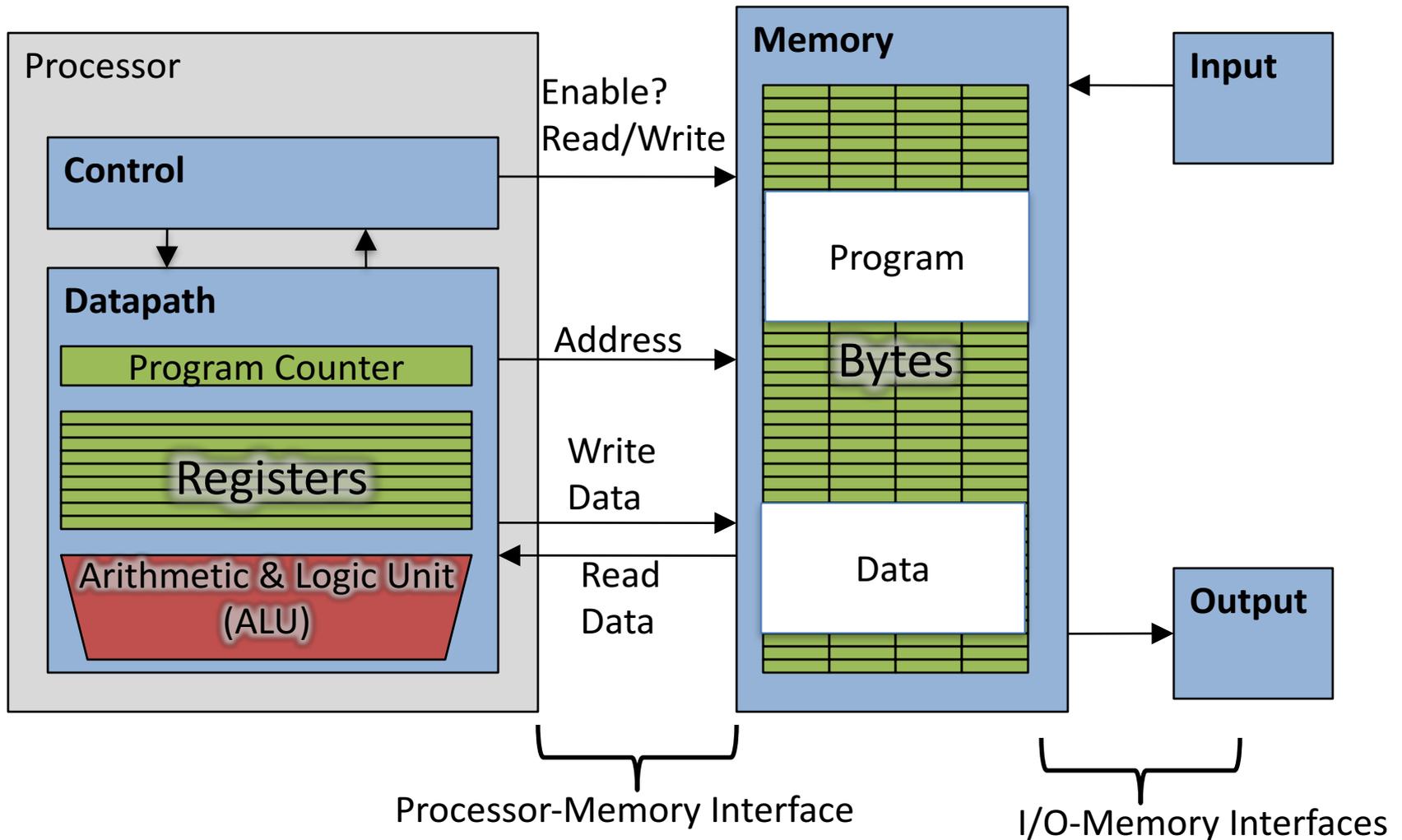
Anything can be represented as a *number*, i.e., data or instructions

```
lw  xt0, 0(x2)
lw  xt1, 4(x2)
sw  xt1, 0(x2)
sw  xt0, 4(x2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```



Review: Components of a Computer



Last lecture

- In RISC-V Assembly Language:
 - Registers replace C variables
 - One instruction (simple operation) per line
 - Simpler is Better, Smaller is Faster
- In RV32, words are 32bit
- Instructions:
`add, addi, sub, lw, sw, lb`
- Registers:
 - 32 registers, referred to as `x0 – x31`
 - Zero: `x0`

RISC-V: Little Endian

(E.g., $1025 = 0x401 = 0b\ 0100\ 0000\ 0001$)

ADDR3	ADDR2	ADDR1	ADDR0
BYTE3	BYTE2	BYTE1	BYTE0
00000000	00000000	00000100	00000001

Little Endian

Most significant byte in a word
(numbers are addresses) ↓

...
<u>12</u>	13	14	15
<u>8</u>	9	10	11
<u>4</u>	5	6	7
<u>0</u>	1	2	3

- Hexadecimal number:

0xFD34AB88 ($4,248,087,432_{\text{ten}}$) =>

- Byte **0**: **0x88** (136_{ten})
- Byte **1**: **0xAB** (171_{ten})
- Byte **2**: **0x34** (52_{ten})
- Byte **3**: **0xFD** (253_{ten})

Address:	64 address of word (e.g. int)			
Address:	64	65	66	67
Data:	0x88	0xAB	0x34	0xFD

- Little Endian: The "Endianess" is little:
 - It starts with the smallest (least significant) Byte
 - Swapped from how we write the number

Peer Instruction

We want to translate $*x = *y$ into RISC-V
 x, y ptrs stored in: $x3$ $x5$

```
1: add x3, x5, zero
2: add x5, x3, zero
3: lw  x3, 0(x5)
4: lw  x5, 0(x3)
5: lw  x8, 0(x5)
6: sw  x8, 0(x3)
7: lw  x5, 0(x8)
8: sw  x3, 0(x8)
```

A	1
B	2
C	3
D	4
E	5 → 6
F	6 → 5
G	7 → 8

RISC-V Logical Instructions

- Useful to operate on fields of bits within a word
 - e.g., characters within a word (8 bits)
- Operations to pack /unpack bits into words
- Called *logical operations*

Logical operations	C operators	Java operators	RISC-V instructions
Bit-by-bit AND	&	&	and
Bit-by-bit OR			or
Bit-by-bit XOR	^	^	xor
Bit-by-bit NOT	~	~	xori
Shift left	<<	<<	sll
Shift right	>>	>>	srl

RISC-V Logical Instructions

- Always two variants
 - Register: `and x5, x6, x7 # x5 = x6 & x7`
 - Immediate: `andi x5, x6, 3 # x5 = x6 & 3`
- Used for 'masks'
 - `andi` with `0000 00FFhex` isolates the least significant byte
 - `andi` with `FF00 0000hex` isolates the most significant byte
 - `andi` with `0000 0008hex` isolates the 4th bit (`0000 1000two`)

Your Turn. What is in x11?

```
xor    x11, x10, x10
ori    x11, x11, 0xFF
andi   x11, x11, 0xF0
```

A:

0x0

B:

0xF

C:

0xF0

D:

0xFF00

E:

0xFFFFFFFF

A:	0x0
B:	0xF
C:	0xF0
D:	0xFF00
E:	0xFFFFFFFF

Logic Shifting

- Shift Left: `sll x11, x12, 2` #`x11=x12<<2`
 - Store in x11 the value from x12 shifted 2 bits to the left (they fall off end), **inserting 0's** on right; `<<` in C.

Before: `0000 0002`_{hex}

`0000 0000 0000 0000 0000 0000 0000 0010`_{two}

After: `0000 0008`_{hex}

`0000 0000 0000 0000 0000 0000 0000 1000`_{two}

What arithmetic effect does shift left have?

multiply with 2^n

- Shift Right: `srl` is opposite shift; `>>`

Arithmetic Shifting

- Shift right arithmetic moves n bits to the right (insert high order sign bit into empty bits)
- For example, if register x10 contained
1111 1111 1111 1111 1111 1111 1110 0111_{two} = -25_{ten}
- If executed sra x10, x10, 4, result is:
1111 1111 1111 1111 1111 1111 1111 1110_{two} = -2_{ten}
- Unfortunately, this is NOT same as dividing by 2^n
 - Fails for odd negative numbers
 - C arithmetic semantics is that division should round towards 0

Your Turn. What is in x12?

x10 holds 0x34FF

slli x12, x10, 0x10

srli x12, x12, 0x08

and x12, x12, x10

A:

0x0

B:

0x3400

C:

0x4F0

D:

0xFF00

E:

0x34FF

A:	0x0
B:	0x3400
C:	0x4F0
D:	0xFF00
E:	0x34FF

Helpful RISC-V Assembler Features

- Symbolic register names
 - E.g., **a0–a7** for argument registers (**x10–x17**)
 - E.g., **zero** for **x0**
 - E.g., **t0–t6** (temporary) **s0–s11** (saved)
- Pseudo-instructions
 - Shorthand syntax for common assembly idioms
 - E.g., **mv rd, rs = addi rd, rs, 0**
 - E.g., **li rd, 13 = addi rd, x0, 13**

Computer Decision Making

- Based on computation, do something different
- In programming languages: *if*-statement
- RISC-V: *if*-statement instruction is
beq register1, register2, L1
means: go to statement labeled L1
if (value in register1) == (value in register2)
....otherwise, go to next statement
- **beq** stands for *branch if equal*
- Other instruction: **bne** for *branch if not equal*

Types of Branches

- **Branch** – change of control flow
- **Conditional Branch** – change control flow depending on outcome of comparison
 - branch *if* equal (**beq**) or branch *if not* equal (**bne**)
 - Also branch if less than (**blt**) and branch if greater than or equal (**bge**)
- **Unconditional Branch** – always branch
 - a RISC-V instruction for this: *jump* (**j**), as in **j label**

Label

- Holds the address of data or instructions
 - Think: "constant pointer"
 - Will be replaced by the actual address (number) during assembly (or linking)

- Also available in C for "goto":

- **NEVER** use goto !!!!
Very bad programming style!

```
1  static int somedata = 10;
2
3  main(){
4      int tmp = somedata;
5      loop: // label called "loop"
6          tmp = tmp + 1;
7          goto loop;
8  }
```

Label

```
1  .data          # Static data
2
3  somedata:     # label to some data "somedata"
4      .word     0xA # initialize the word (32bit) with 10
5
6  .text        # code (instructions) follow here
7
8  main:        # label to instruction "main function"
9
10     la x6, somedata # address of "somedata" in x6
11     lw x5, 0(x6)   # (initial) value of "somedata" to x5
12
13  loop:       # label to the next instruction:
14             # some jump goal in function (name "loop")
15
16     addi, x5, x5, 1 # x5 += 1 (label "loop" points here)
17     j loop         # jump to loop
```

```
1  static int somedata = 10;
2
3  main(){
4      int tmp = somedata;
5      loop: // label called "loop"
6      tmp = tmp + 1;
7      goto loop;
8  }
```

Example *if* Statement

- Assuming translations below, compile *if* block

$f \rightarrow x10$ $g \rightarrow x11$ $h \rightarrow x12$

$i \rightarrow x13$ $j \rightarrow x14$

```
if (i == j)           bne x13,x14,Exit
    f = g + h;         add x10,x11,x12
                        Exit:
```

- May need to negate branch condition

Example *if-else* Statement

- Assuming translations below, compile

$f \rightarrow x10$ $g \rightarrow x11$ $h \rightarrow x12$

$i \rightarrow x13$ $j \rightarrow x14$

if (i == j)

f = g + h;

else

f = g - h;

bne x13,x14,Else

add x10,x11,x12

j Exit

Else: sub x10,x11,x12

Exit:

Magnitude Compares in RISC-V

- Until now, we've only tested equalities (`==` and `!=` in C); General programs need to test `<` and `>` as well.

- RISC-V magnitude-compare branches:

“Branch on Less Than”

Syntax: **`blt reg1,reg2, label`**

Meaning: `if (reg1 < reg2) // treat registers as signed integers
goto label;`

- “Branch on Less Than Unsigned”

Syntax: **`bltu reg1,reg2, label`**

Meaning: `if (reg1 < reg2) // treat registers as unsigned integers
goto label;`

C Loop Mapped to RISC-V Assembly

```
int A[20];
int sum = 0;
for (int i=0; i < 20; i++)
    sum += A[i];
```

```
add x9, x8, x0 # x9=&A[0]
add x10, x0, x0 # sum=0
add x11, x0, x0 # i=0
addi x13, x0, 20 # x13=20
```

Loop:

```
bge x11, x13, Done
lw x12, 0(x9) # x12=A[i]
add x10, x10, x12 # sum+=
addi x9, x9, 4 # &A[i+1]
addi x11, x11, 1 # i++
j Loop
```

Done:

Administrivia

- HW2 Autolab is online – due in one week! March 14
 - START latest today!
 - Go to OH if you have problems – don't ask your fellow students
 - Use piazza frequently.
- HW3 and Project 1.1 will be published this weekend!
- Midterm I is in one month during lecture hours ... (April 3rd)

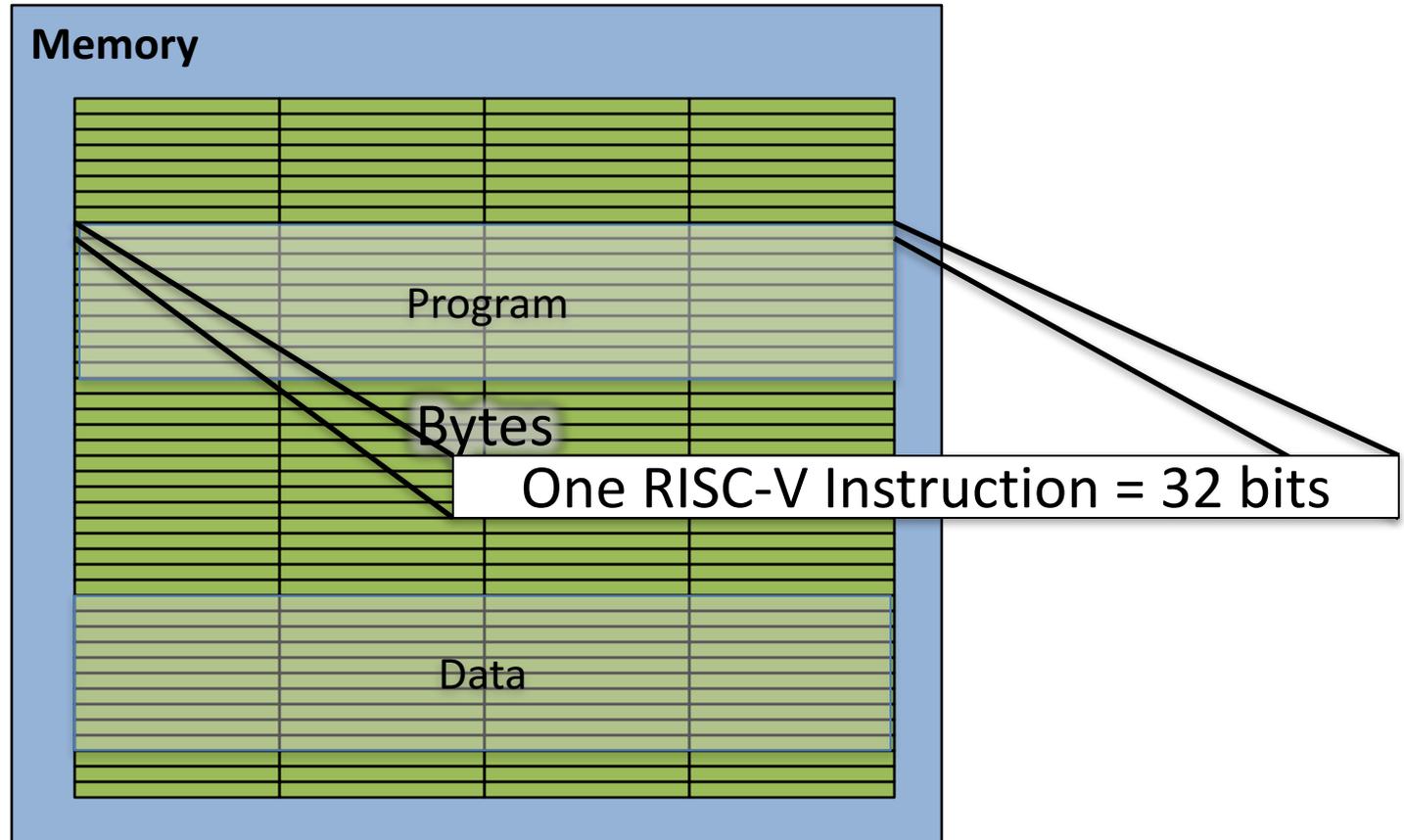
Schedule

2 weeks for HW and Projects; start early!

Week ↕	Date ↕	Topic ↕	Reading ↕	Discussion ↕	Homework ↕	Lab ↕	Project ↕
1	02-20	Introduction (pdf)	P&H: 2.4	No Discussion	HW1: On Autolab Due Tue., Mar 5, 23:59	No Lab	
	02-22	Introduction to C I (pdf)	K&R Ch. 1-5				
2	02-27	Introduction to C II (pdf)	K&R Ch. 6-7	Discussion 1	HW2 Due Thu., Mar 14, 23:59	Lab 1	
	03-01	Introduction to C III (pdf)	K&R Ch. 8, App. A & B				
3	03-06	RISC-V Intro (pdf)	P&H: 2.1 - 2.3	Discussion 2		Lab 2	Project 1.1 Due Mar. 31, 23:59
	03-08	RISC-V Decisions	P&H: 2.6, 2.7, 2.9, 2.10				
4	03-13	RISC-V Instruction Formats	P&H: 2.5, 2.10		HW3 Due Thu., Mar 25, 23:59	Lab 3	
	03-15	Compiler, Assembler, Linker, Loader (CALL)	P&H: 2.12				
5	03-20	Intro to Synchronous Digital Systems (SDS), Logic	P&H: A.2, A.3				Project 1.2 Due April. 14, 23:59
	03-22	Functional Units, FSM	P&H: A.3 - A.6				
6	03-27	RISC-V Datapath, Single-Cycle Control Intro	P&H: 4.1, 4.3, 4.4				
	03-29	RISC-V Single-Cycle Control	P&H: 4.5 - 4.8				
7	04-03	Mid-Term I					

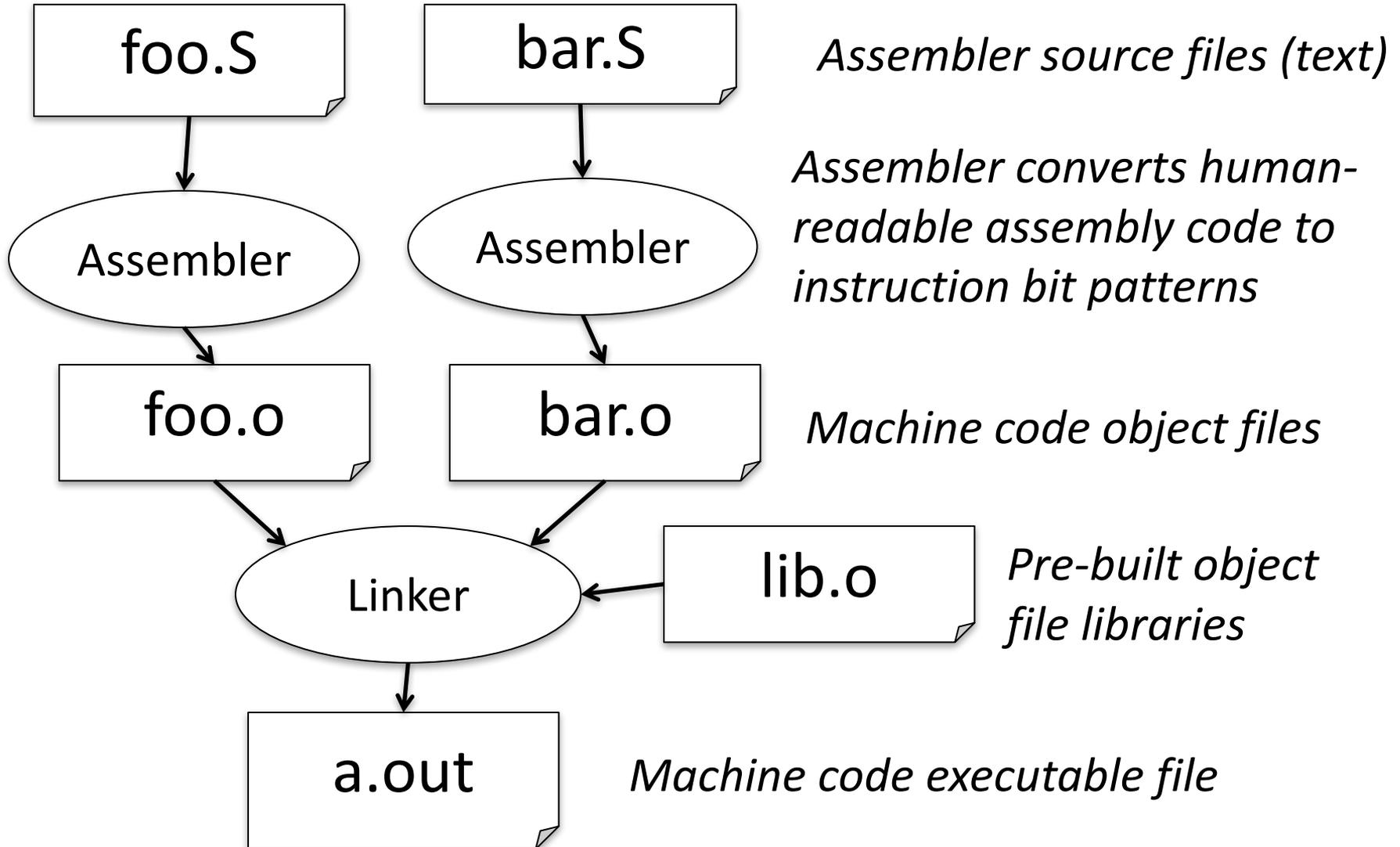
Procedures in RISC-V

How Program is Stored

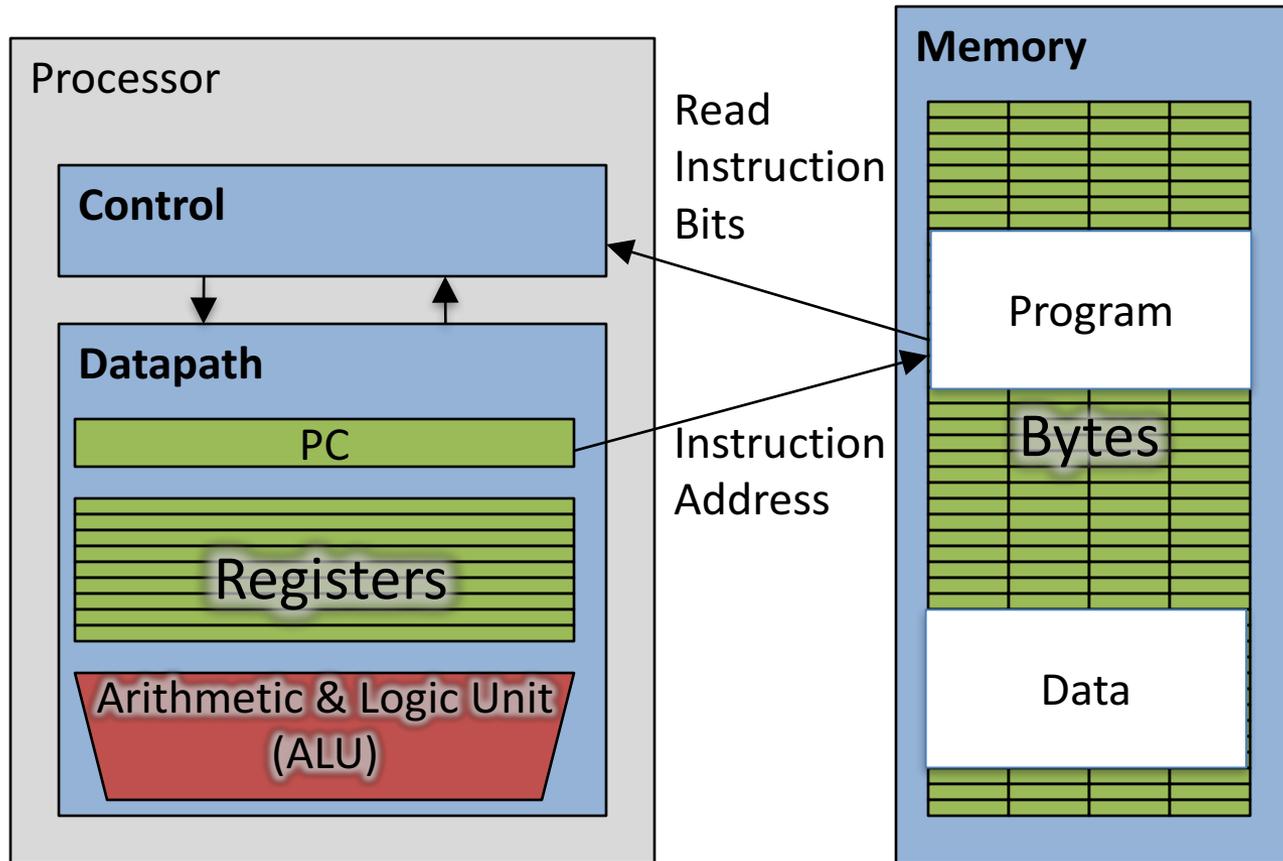


Assembler to Machine Code

(more later in course)



Executing a Program



- The **PC** (program counter) is internal register inside processor holding byte address of next instruction to be executed.
- Instruction is fetched from memory, then control unit executes instruction using datapath and memory system, and updates program counter (default is add +4 bytes to PC, to move to next sequential instruction)

C Functions

```
main() {  
    int i,j,k,m;  
    ...  
    i = mult(j,k); ...  
    m = mult(i,i); ...  
}
```

What information must
compiler/programmer
keep track of?

```
/* really dumb mult function */  
int mult (int mcand, int mlier){  
    int product = 0;  
    while (mlier > 0) {  
        product = product + mcand;  
        mlier = mlier -1; }  
    return product;  
}
```

What instructions can
accomplish this?

Six Fundamental Steps in Calling a Function

1. Put parameters in a place where function can access them
2. Transfer control to function
3. Acquire (local) storage resources needed for function
4. Perform desired task of the function
5. Put result value in a place where calling code can access it and restore any registers you used
6. Return control to point of origin, since a function can be called from several points in a program

RISC-V Function Call Conventions

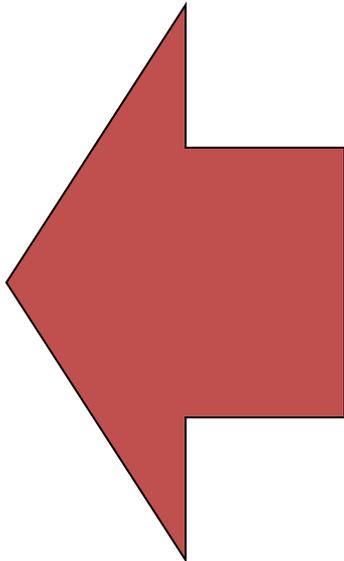
- Registers faster than memory, so use them
- $a0-a7$ ($x10-x17$): eight *argument* registers to pass parameters and return values ($a0-a1$)
- ra : one *return address* register to return to the point of origin ($x1$)
- Also $s0-s1$ ($x8-x9$) and $s2-s11$ ($x18-x27$): saved registers (more about those later)

Instruction Support for Functions (1/4)

```
... sum(a,b);... /* a, b: s0, s1 */  
}  
C int sum(int x, int y) {  
    return x+y;  
}
```

address (shown in decimal)

1000
1004
1008
1012
1016
...
2000
2004



In RISC-V, all instructions are 4 bytes, and stored in memory just like data. So here we show the addresses of where the programs are stored.

Instruction Support for Functions (2/4)

C

```
... sum(a,b);... /* a, b: s0, s1 */  
}  
int sum(int x, int y) {  
    return x+y;  
}
```

address (shown in decimal)

RISC-V

```
1000 add    a0, s0, x0           # x = a  
1004 mv     a1, s1             # y = b  
1008 addi   ra, zero, 1016     # ra=1016  
1012 j      sum                # jump to sum  
1016 ..  
..  
2000 sum:  add  a0, a0, a1  
2004 jr    ra      # new instr. "jump register"
```

Instruction Support for Functions (3/4)

C

```
... sum(a,b);... /* a,b:$s0,$s1 */  
}  
int sum(int x, int y) {  
    return x+y;  
}
```

- Question: Why use **jr** here? Why not use **j**?
- Answer: **sum** might be called by many places, so we can't return to a fixed place. The calling proc to **sum** must be able to say "return here" somehow.

RISC-V

```
2000 sum: add a0, a0, a1  
2004 jr ra # new instr. "jump register"
```

Instruction Support for Functions (4/4)

- Single instruction to jump and save return address:
jump and link (**jal**)
- Before:

```
1008 addi ra, zero, 1016 # $ra=1016
1012 j sum # goto sum
```
- After:

```
1008 jal sum # ra=1012, goto sum
```
- Why have a **jal**?
 - Make the common case fast: function calls very common.
 - Reduce program size
 - Don't have to know where code is in memory with **jal**!

RISC-V Function Call Instructions

- Invoke function: *jump and link* instruction (`jal`)
(really should be `laj` “*link and jump*”)
 - “link” means form an *address* or *link* that points to calling site to allow function to return to proper address
 - Jumps to address and simultaneously saves the address of the following instruction in register `ra` (`x1`)
`jal FunctionLabel`
- Return from function: *jump register* instruction (`jr`)
 - Unconditional jump to address specified in register
`jr ra`
 - Assembler shorthand: `ret = jr ra`

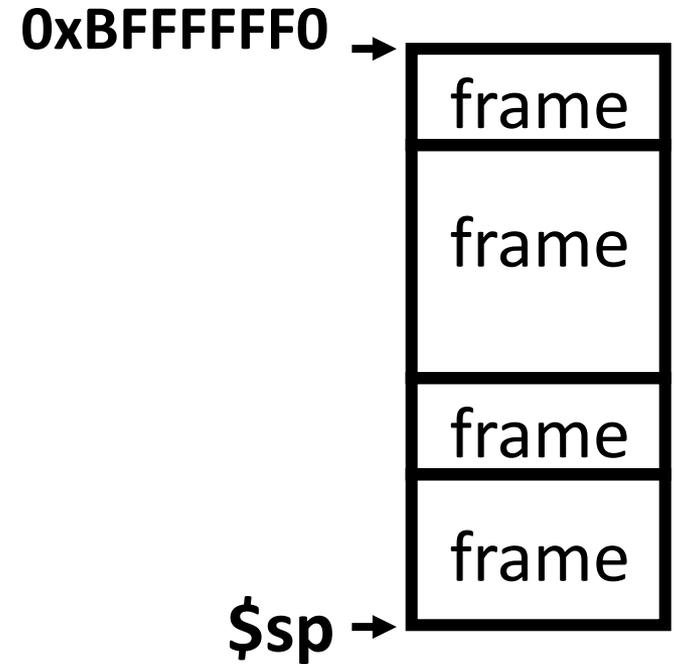
Notes on Functions

- Calling program (*caller*) puts parameters into registers `a0–a7` and uses `jal X` to invoke (*callee*) at address labeled `X`
- Must have register in computer with address of currently executing instruction
 - Instead of *Instruction Address Register* (better name), historically called *Program Counter* (*PC*)
 - It's a program's counter; it doesn't count programs!
- What value does `jal X` place into `ra`? **????**
- `jr ra` puts address inside `ra` back into PC

Where Are Old Register Values Saved to Restore Them After Function Call?

- Need a place to save old values before call function, restore them when return, and delete
- Ideal is *stack*: last-in-first-out queue (e.g., stack of plates)
 - Push: placing data onto stack
 - Pop: removing data from stack
- Stack in memory, so need register to point to it
- `sp` is the *stack pointer* in RISC-V (x2)
- Convention is grow from high to low addresses
 - *Push* decrements `sp`, *Pop* increments `sp`

Stack



- Stack frame includes:
 - Return “instruction” address
 - Parameters
 - Space for other local variables
- Stack frames contiguous blocks of memory; stack pointer tells where bottom of stack frame is
- When procedure ends, stack frame is tossed off the stack; frees memory for future stack frames

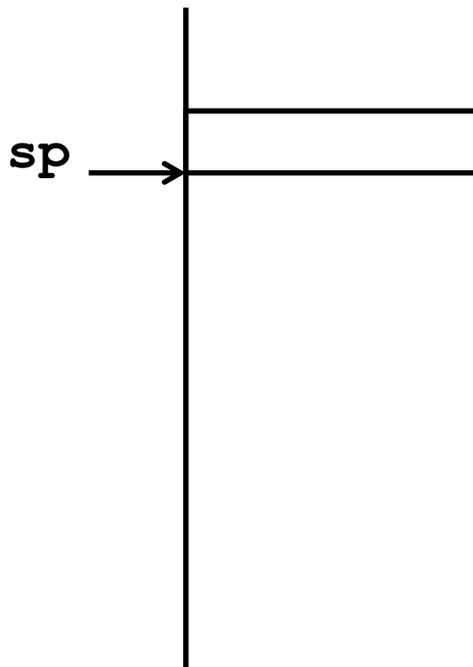
Example

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

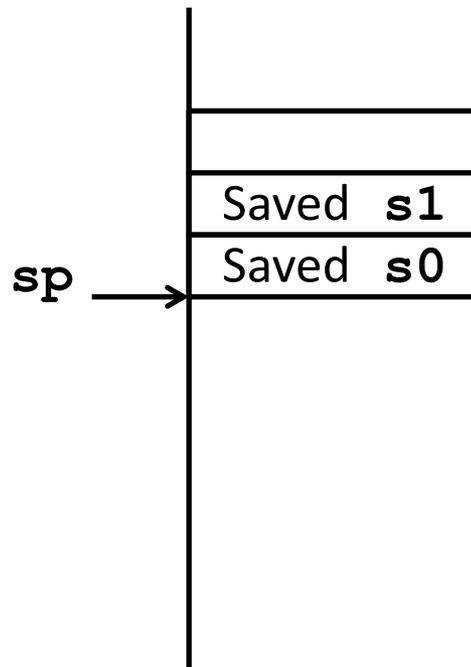
- Parameter variables g , h , i , and j in argument registers $a0$, $a1$, $a2$, and $a3$, and f in $s0$
- Assume need one temporary register $s1$

Stack Before, During, After Function

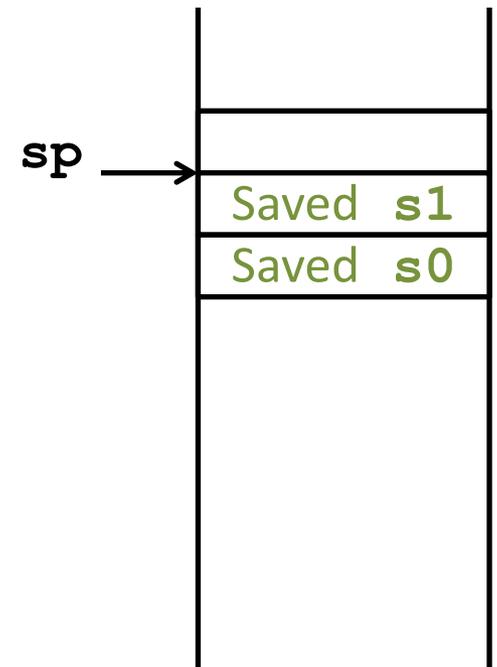
- Need to save old values of `s0` and `s1`



Before call



During call



After call

RISC-V Code for Leaf()

Leaf:

```
addi  sp, sp, -8  # adjust stack for 2 items
sw    s1, 4(sp)   # save s1 for use afterwards
sw    s0, 0(sp)   # save s0 for use afterwards

add   s0, a0, a1  # f = g + h
add   s1, a2, a3  # s1 = i + j
sub   a0, s0, s1  # return value (g + h) - (i + j)

lw    s0, 0(sp)   # restore register s0 for caller
lw    s1, 4(sp)   # restore register s1 for caller
addi  sp, sp, 8   # adjust stack to delete 2 items
jr    ra         # jump back to calling routine
```

Nested Procedures (1/2)

```
int sumSquare(int x, int y) {  
    return mult(x,x)+ y;  
}
```

- Something called **sumSquare**, now **sumSquare** is calling **mult**
- So there's a value in **ra** that **sumSquare** wants to jump back to, but this will be overwritten by the call to **mult**

Need to save **sumSquare** return address before call to **mult**

Nested Procedures (2/2)

- In general, may need to save some other info in addition to `ra`.
- When a C program is run, there are 3 important memory areas allocated:
 - **Static**: Variables declared once per program, cease to exist only after execution completes - e.g., C globals
 - **Heap**: Variables declared dynamically via **malloc**
 - **Stack**: Space to be used by procedure during execution; this is where we can save register values

Register Conventions (1/2)

- CalleR: the calling function
- CalleE: the function being called
- When callee returns from executing, the caller needs to know which registers may have changed and which are guaranteed to be unchanged.
- **Register Conventions**: A set of generally accepted rules as to which registers will be unchanged after a procedure call (**jal**) and which may be changed.

Register Conventions (2/2)

To reduce expensive loads and stores from spilling and restoring registers, RISC-V function-calling convention divides registers into two categories:

1. Preserved across function call
 - Caller can rely on values being unchanged
 - **sp, gp, tp**, “saved registers” **s0- s11** (**s0** is also **fp**)
2. Not preserved across function call
 - Caller *cannot* rely on values being unchanged
 - Argument/return registers **a0-a7**, **ra**, “temporary registers” **t0-t6**

RISC-V Symbolic Register Names

Numbers: hardware understands

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

Human-friendly **symbolic names** in assembly code

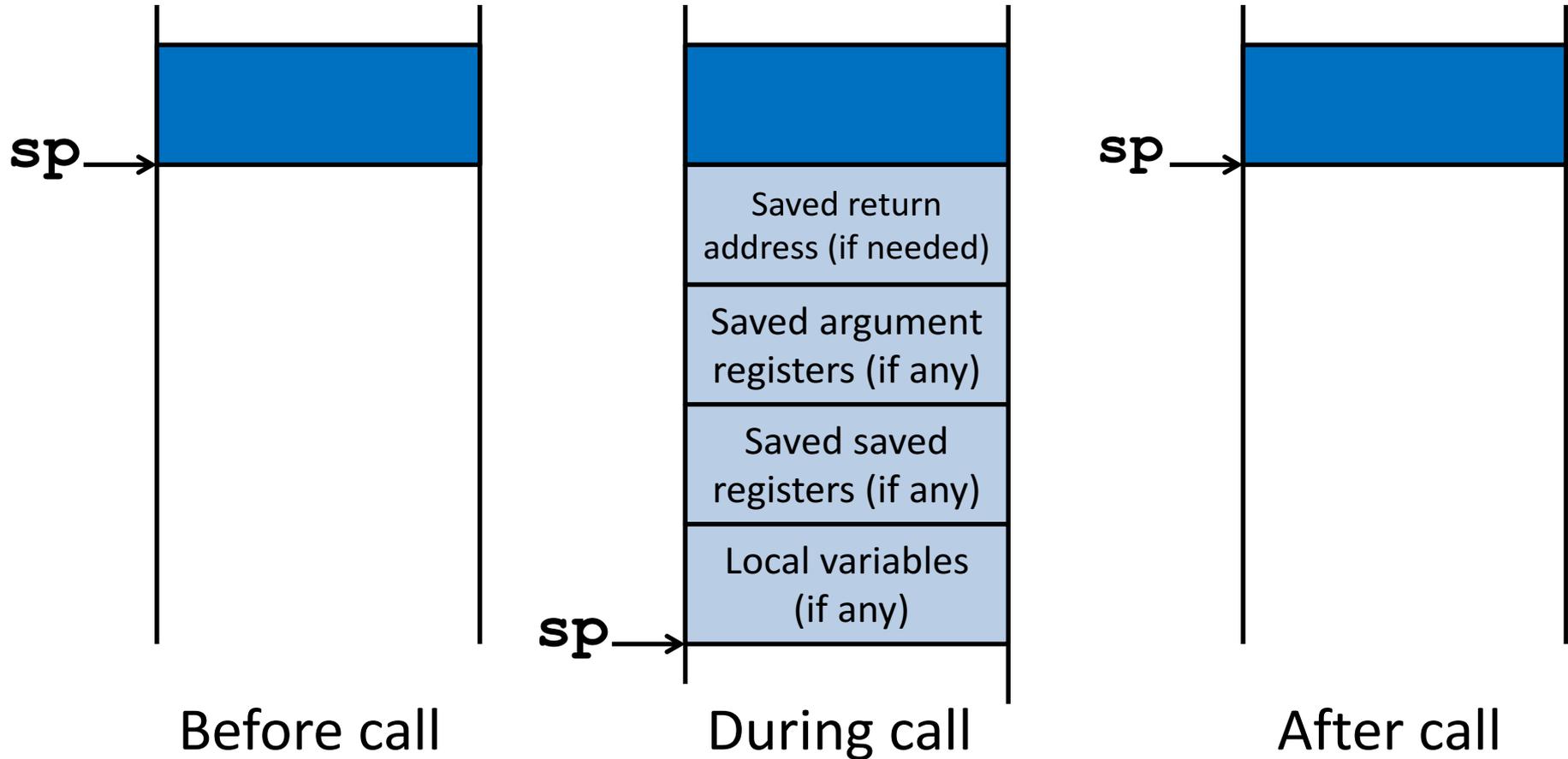
Question

- Which statement is FALSE?
 - A: RISC-V uses `jal` to invoke a function and `jr` to return from a function
 - B: `jal` saves PC+1 in `ra`
 - C: The callee can use temporary registers (`ti`) without saving and restoring them
 - D: The caller can rely on save registers (`si`) without fear of callee changing them

Allocating Space on Stack

- C has two storage classes: automatic and static
 - *Automatic* variables are local to function and discarded when function exits
 - *Static* variables exist across exits from and entries to procedures
- Use stack for automatic (local) variables that don't fit in registers
- *Procedure frame* or *activation record*: segment of stack with saved registers and local variables

Stack Before, During, After Function



Using the Stack (1/2)

- We have a register **sp** which always points to the last used space in the stack.
- To use stack, we decrement this pointer by the amount of space we need and then fill it with info.
- So, how do we compile this?

```
int sumSquare(int x, int y) {  
    return mult(x,x)+ y;  
}
```

Using the Stack (2/2)

```
int sumSquare(int x, int y) {  
    return mult(x,x)+ y; }  
}
```

sumSquare:

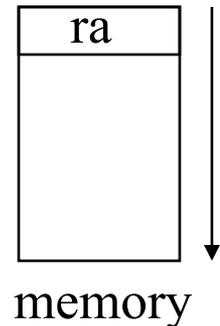
```
“push”  addi    sp, sp, -8    # space on stack  
        sw     ra, 4(sp)   # save ret addr  
        sw     a1, 0(sp)   # save y  
        mv     a1, a0      # mult(x,x)  
        jal    mult       # call mult  
        lw     a1, 0(sp)   # restore y  
“pop”   add     a0, a0, a1   # mult()+y  
        lw     ra, 4(sp)   # get ret addr  
        addi   sp, sp, 8   # restore stack  
        jr    ra  
mult:   ...
```

Basic Structure of a Function

Prologue

```
entry_label:  
addi sp,sp, -framesize  
sw    ra, framesize-4(sp)  # save ra  
save other regs if need be
```

Body ... (call other functions...)



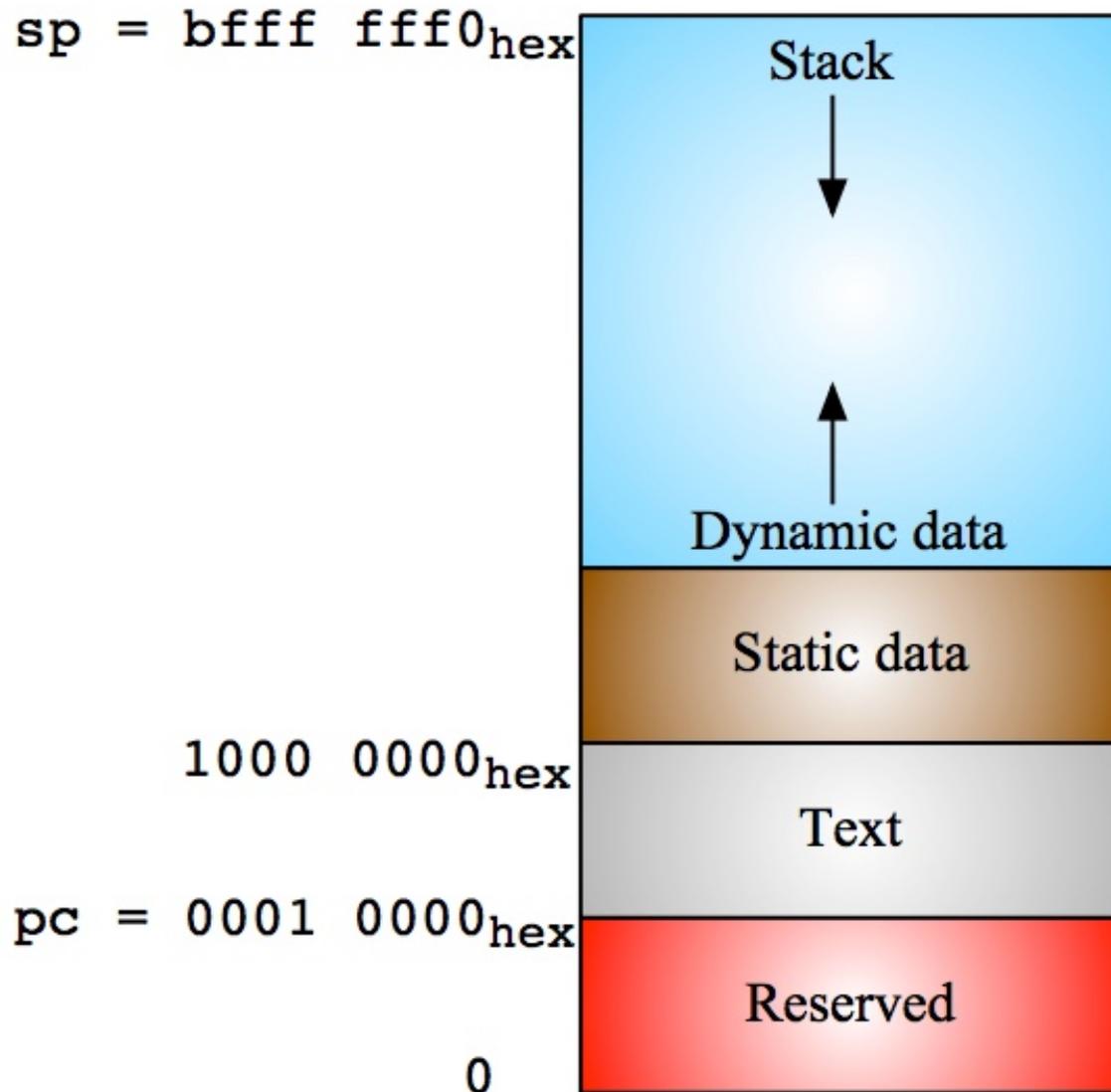
Epilogue

```
restore other regs if need be  
lw    ra, framesize-4(sp)  # restore $ra  
addi sp, sp, framesize  
jr ra
```

Where is the Stack in Memory?

- RV32 convention (RV64 and RV128 have different memory layouts)
- Stack starts in high memory and grows down
 - Hexadecimal: `bfff_fff0hex`
 - Stack must be aligned on 16-byte boundary (not true in examples above)
- RV32 programs (*text segment*) in low end
 - `0001_0000hex`
- *static data segment* (constants and other static variables) above text for static variables
 - RISC-V convention *global pointer* (**gp**) points to static
 - RV32 **gp** = `1000_0000hex`
- *Heap* above static for data structures that grow and shrink ; grows up to high addresses

RV32 Memory Allocation



“And in Conclusion...”

- Registers we know so far (All of them!)
 - a0-a7 for function arguments, a0-a1 for return values
 - sp, stack pointer, ra return address
 - s0-s11 saved registers
 - t0-t6 temporaries
 - zero
- Instructions we know:
 - Arithmetic: add, addi, sub
 - Logical: sll, srl, sla, slli, srli, slai, and, or, xor, andi, ori, xori
 - Decision: beq, bne, blt, bge
 - Unconditional branches (jumps): j, jr
 - Functions called with `jal`, return with `jr ra`.
- The stack is your friend: Use it to save anything you need. Just leave it the way you found it!