

CS 110
Computer Architecture
Lecture 5:
More RISC-V, RISC-V Functions

Instructors:

Sören Schwertfeger & Chundong Wang

<https://robotics.shanghaitech.edu.cn/courses/ca/20s/>

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkley's CS61C

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 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

Logic Shifting

- Shift Left: `slli 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

- All shift instructions: register and immediate variant!
- 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 srai 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?

```
addi    x10, x0, 0x7FF
slli    x12, x10, 0x10
srli    x12, x12, 0x08
and     x12, x12, x10
```



A:	0x0
B:	0x700
C:	0x7F0
D:	0xFF00
E:	0x7FF

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**

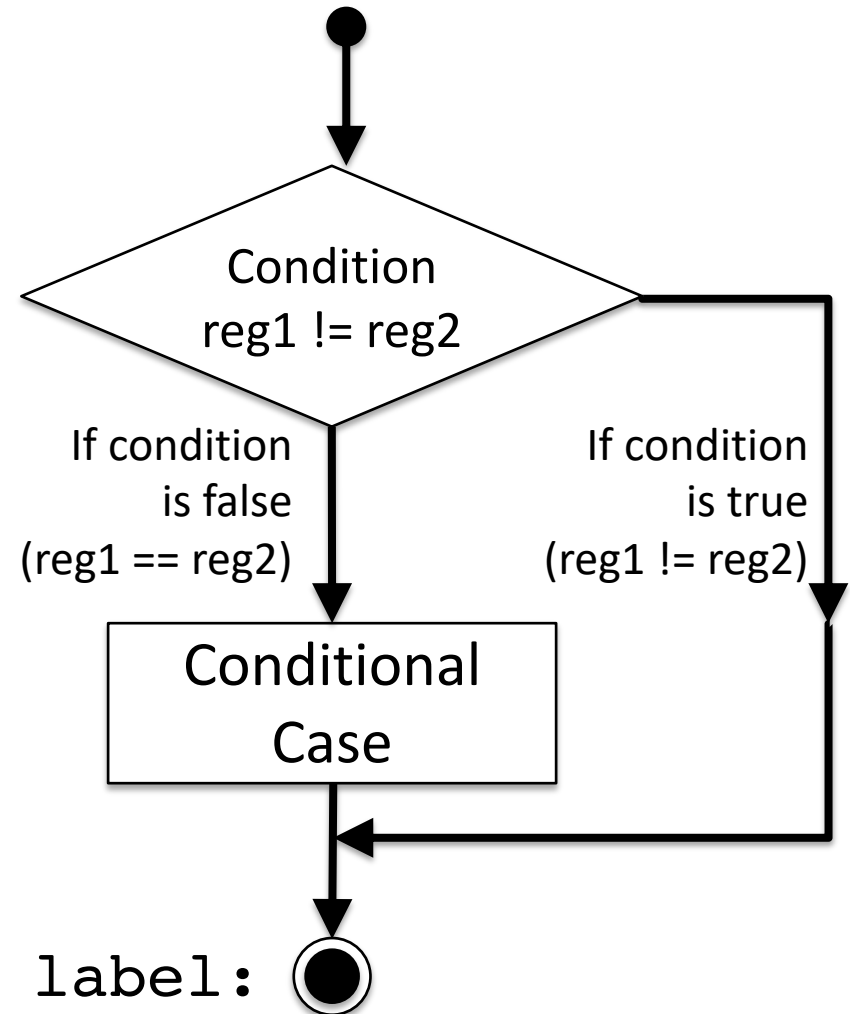
Computer Decision Making

- Based on computation, do something different
- Normal operation: execute instructions in sequence
- 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*

bne flowchart

bne

- Branch if not equal
- `bne reg1, reg2, label`
- Jump if condition is true
- Condition false:
 - continue with next instruction
- If label is after bne:
 - Conditional case will reach label (if no other jump)



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          # Assembler directive
2                  # static data
3
4  somedata:      # Label to some data "somedata"
5      .word      0xA # inicializa the word (32bit) with 10
6
7  .text          # code (instructions) follow here
8
9  main:          # label to first instruction of "main function"
10
11     la x6, somedata # address of "somedata" in x6
12     lw x5, 0(x6)    # (initial) value of "somedata" to x5
13
14  loop:          # label to the next instruction:
15                  # some jump goal in function (name "loop")
16
17     addi, x5, x5, 1 # x5 += 1 (label loop points here)
18     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];
```

```
# Assume x8 holds pointer to A
# Assign x10=sum
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:
```

Optimization

- The simple translation is suboptimal!
 - A more efficient way:
- Inner loop is now 4 instructions rather than 7
 - And only 1 branch/jump rather than two: Because first time through is always true so can move check to the end!
- The compiler will often do this automatically for optimization
 - See that i is only used as an index in a loop

```
# Assume x8 holds pointer to A
# Assign x10=sum
add  x10, x0, x0 # sum=0
add  x11, x8, x0 # ptr = A
addi x12, x11, 80 # end = A + 80
Loop:
    lw  x13, 0(x11) # x13 = *ptr
    add x10, x10, x13 # sum += x13
    addi x11, x11, 4 # ptr++
    blt x11, x12, Loop: # ptr < end
```


Premature Optimization...

- In general we want **correct** translations of C to RISC-V
- It is **not** necessary to optimize
 - Just translate each C statement on its own
- Why?
 - Correctness first, performance second
 - Getting the wrong answer fast is not what we want from you...
 - We're going to need to read your assembly to grade it!
 - Multiple ways to optimize, but the straightforward translation is mostly unique-ish.

Question

- What value does x12 have at the end?
- Answer:
x12 = 16



```
1  #include <stdio.h>
2
3  int main (){
4      int x10 = 7;
5      int x12 = 0;
6      do{
7          int x14 = x10 & 1;
8          if(x14)
9              x12 += x10;
10
11             x10--;
12     } while (x10 != 0);
13     printf("%d", x12);
14 }
```

```
    addi    x10, x0 , 0x07
    add     x12, x0 , x0
label_a:
    andi    x14, x10, 1
    beq     x14, x0 , label_b
    add     x12, x10, x12
label_b:
    addi    x10, x10, -1
    bne     x10, x0 , label_a
```



TA Discussion

Cheng Yu



Q & A



Quiz



Quiz

- **DOWNLOAD** to disk!
- Then edit with proper PDF reader!
- [https://robotics.shanghaitech.edu.cn/courses/ca/20s/notes/CA Lecture 4 Quiz.pdf](https://robotics.shanghaitech.edu.cn/courses/ca/20s/notes/CA%20Lecture%204%20Quiz.pdf)

- Submit to gradescope:
- <https://www.gradescope.com/courses/77872>
- Only if you have problems with gradescope, send the PDF to:
Head TA Yanjie Song <songyj at shanghaitech.edu.cn>

CS 110
Computer Architecture
Lecture 5:
More RISC-V, RISC-V Functions
Video 2: Procedures in RISC-V

Instructors:

Sören Schwertfeger & Chundong Wang

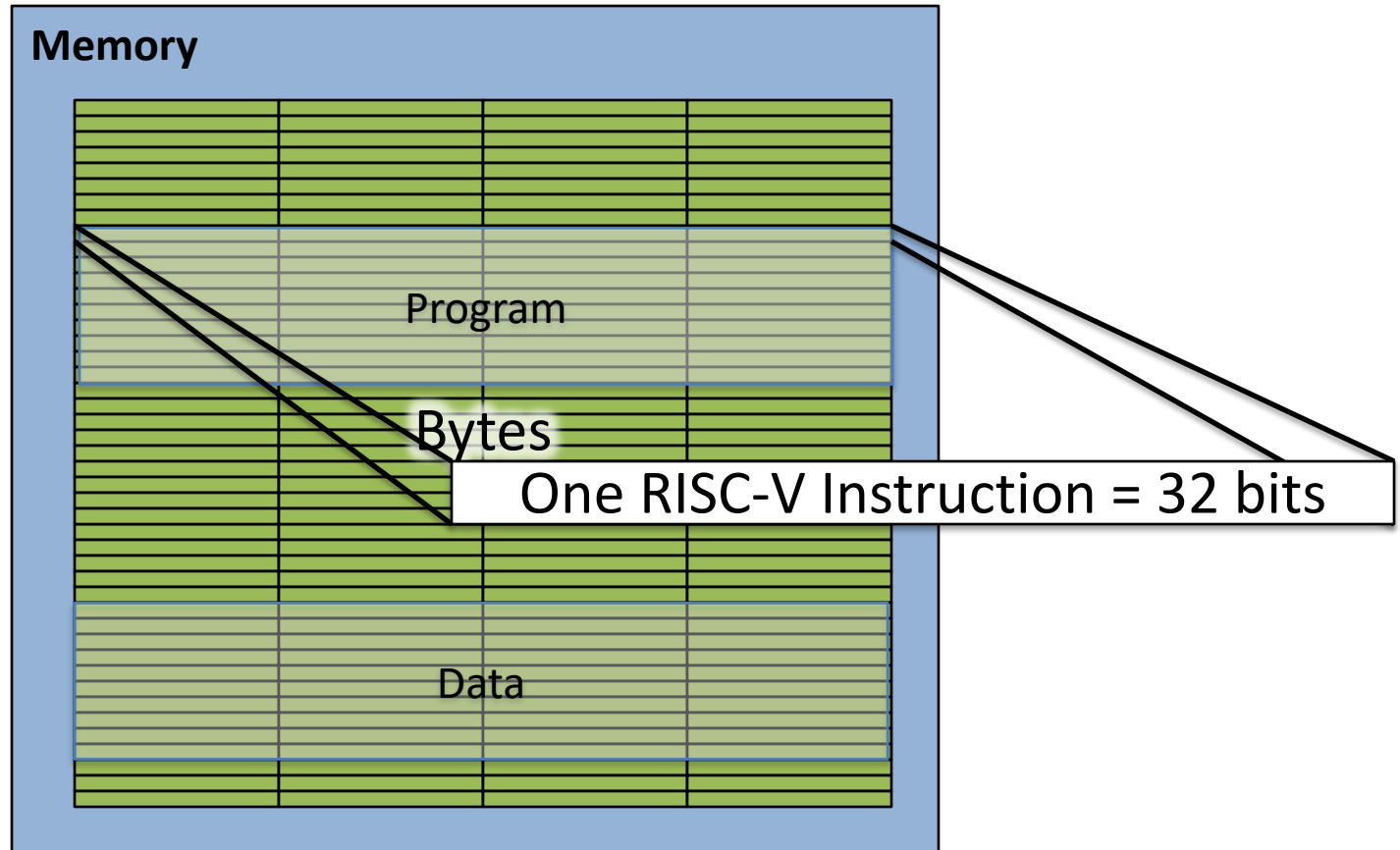
<https://robotics.shanghaitech.edu.cn/courses/ca/20s/>

School of Information Science and Technology SIST

ShanghaiTech University

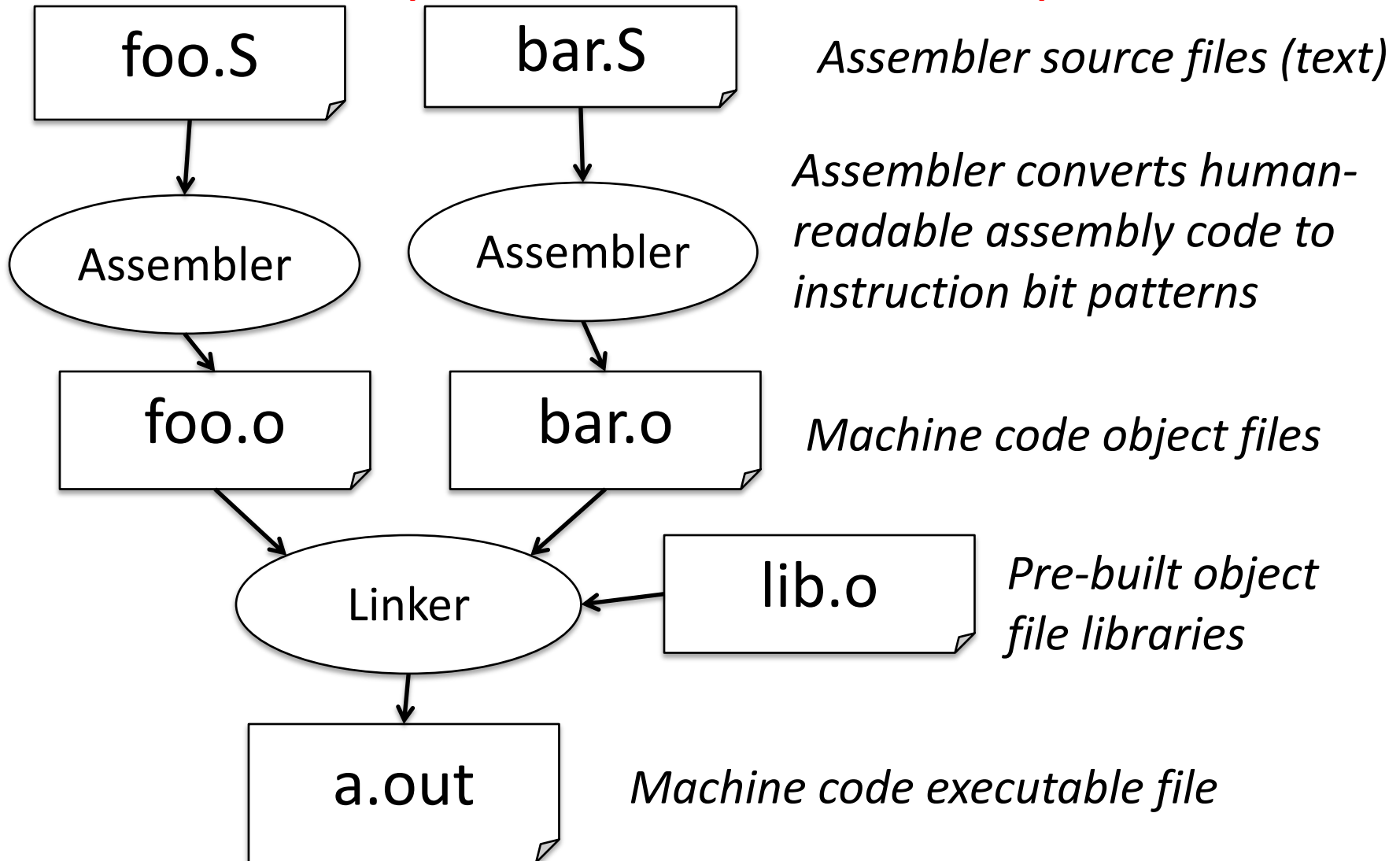
Slides based on UC Berkley's CS61C

How Program is Stored

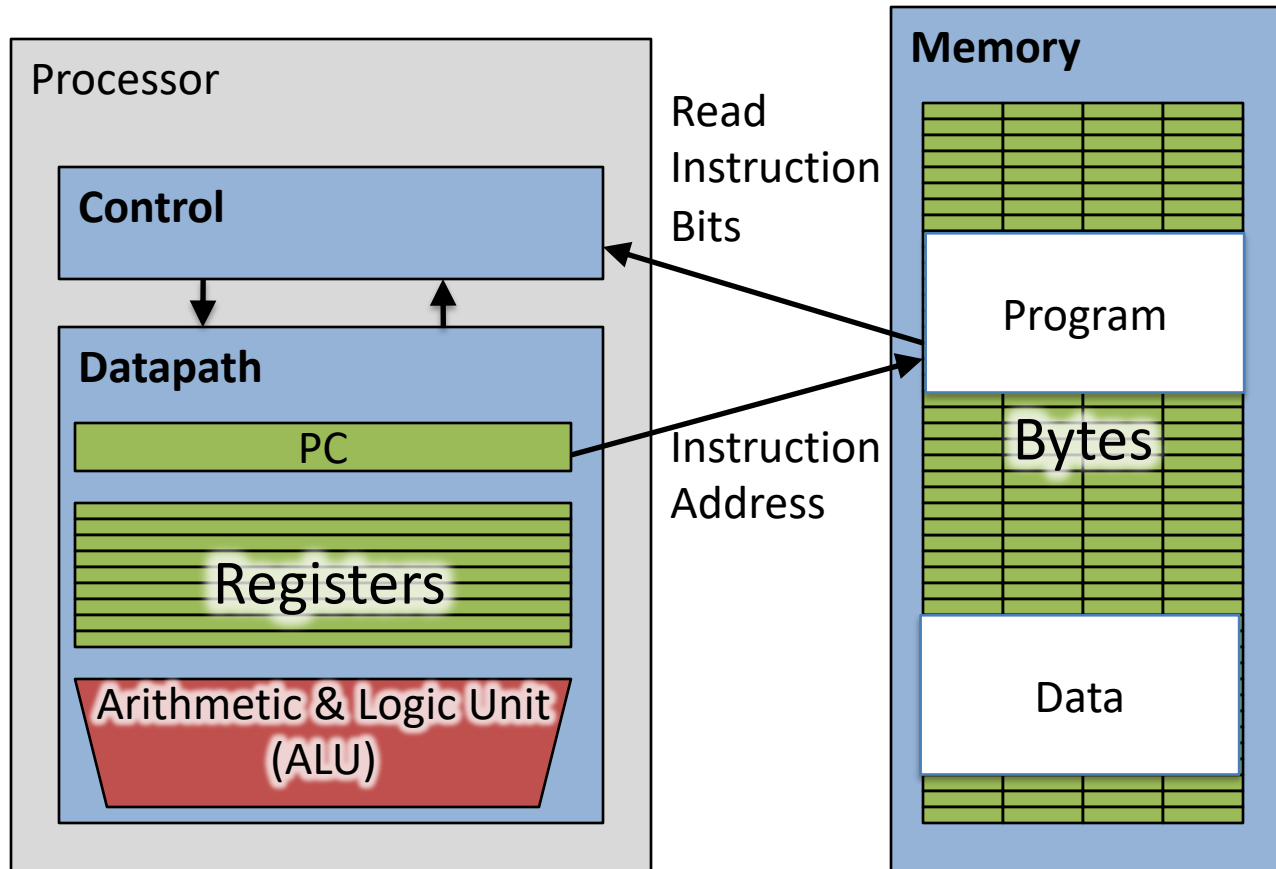


Assembler to Machine Code

(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

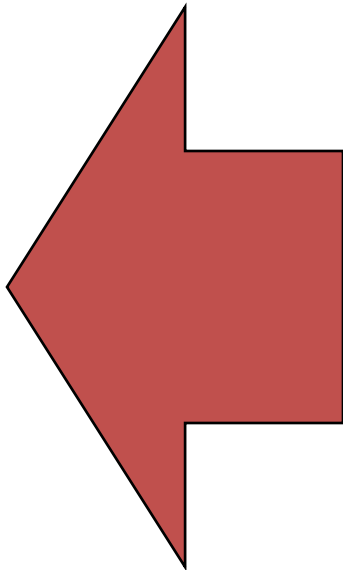
- Registers faster than memory, so use them
- Give names to registers, conventions on how to use them
- $a0-a7$ ($x10-x17$): eight *argument* registers to pass parameters and return values ($a0-a1$)
- ra : one *return address* register to return to the point of origin ($x1$)
- Also $s0-s1$ ($x8-x9$) 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 RV32, 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)

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

address (shown in decimal)


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

Instruction Support for Functions (3/4)

```
... sum(a,b);... /* a,b:$s0,$s1 */
}
C 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**!

Unconditional Branches

- Only two actual instructions
 - **jal rd offset**
 - **jalr rd rs offset**
- Jump And Link
 - Add the immediate value to the current address in the program (the “Program Counter”), go to that location
 - The offset is 20 bits, sign extended and left-shifted **one (not two)**
 - At the same time, store into **rd** the value of PC+4
 - So we know where it came from (need to return to)
 - **jal offset == jal x1 offset** (pseudo-instruction; x1 = ra = return address)
 - **j offset == jal x0 offset** (yes, jump is a pseudo-instruction in RISC-V)
- Two uses:
 - Unconditional jumps in loops and the like
 - Calling other functions

Jump and Link Register

- The same except the destination
 - Instead of PC + immediate it is **rs** + immediate
 - Same immediate format as I-type: 12 bits, sign extended
- Again, if you don't want to record where you jump to...
 - **jr rs == jalr x0 rs**
- Two main uses
 - Returning from functions (which were called using Jump and Link)
 - Calling pointers to function
 - We will see how soon!

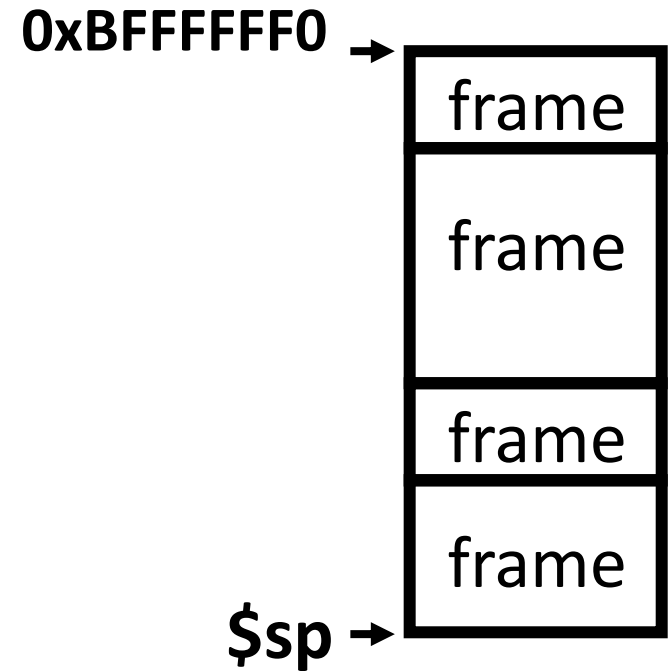
Notes on Functions

- 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

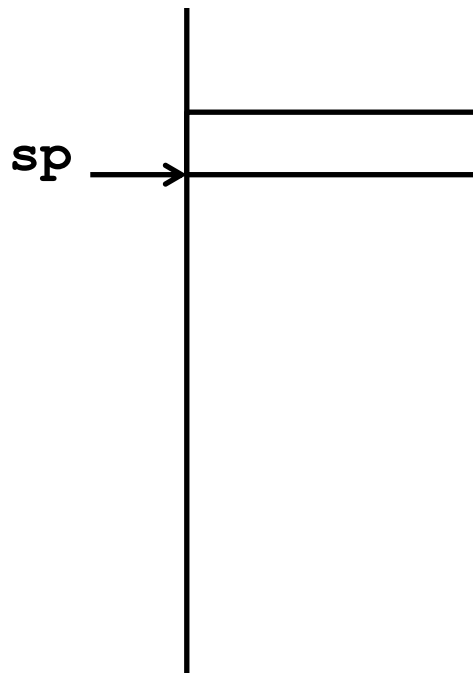
Example

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

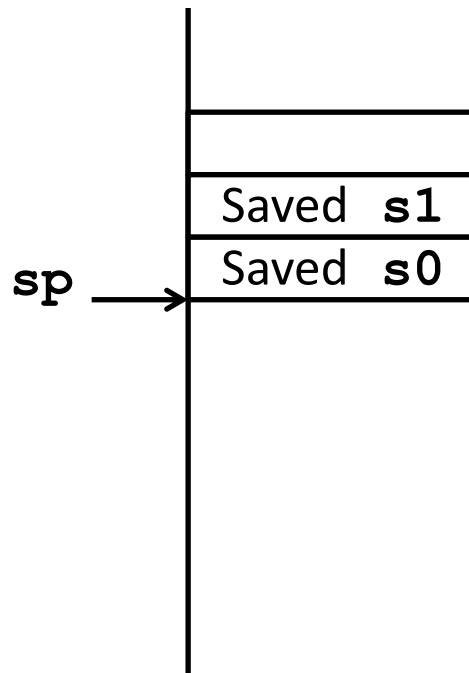
- Parameter variables g , h , i , and j in argument registers $a0$, $a1$, $a2$, and $a3$, and f in $s0$
- Assume need one temporary register $s1$

Stack Before, During, After Function

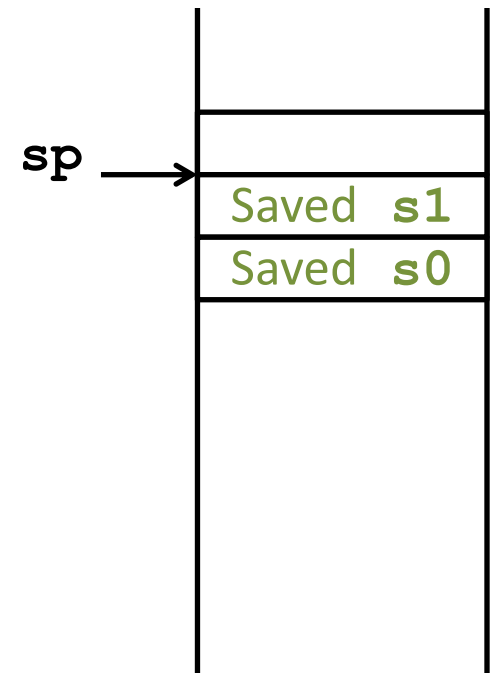
- Need to save old values of `s0` and `s1`



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
```

Question:

Piazza: "Lecture 5 Memory poll"

We want to translate C: $*x = *(y+1)$ into RISC-V
 x, y are int ptrs stored in: $x3$ $x5$

```
1:  addi x3, x5, 1
2:  addi x5, x3, 1
3:  sw   x3, 0(x5)
4:  sw   x5, 1(x3)
5:  sw   x3, 1(x5)
6:  sw   x3, 4(x5)
7:  sw   x5, 4(x3)
8:  sw   x8, 0(x3)
9:  sw   x3, 0(x8)
10: lw   x3, 1(x5)
11: lw   x8, 1(x5)
12: lw   x5, 1(x8)
13: lw   x3, 4(x5)
14: lw   x8, 4(x5)
15: lw   x5, 4(x8)
```

A	1
B	2
C	3
D	4
E	5
F	6
G	10
H	13
I	10→7
J	11→8
K	11→9
L	12→3
M	13→3
N	14→8
O	14→9
P	15→9

CS 110
Computer Architecture
Lecture 5:
More RISC-V, RISC-V Functions
Video 3: Nested Functions

Instructors:

Sören Schwertfeger & Chundong Wang

<https://robotics.shanghaitech.edu.cn/courses/ca/20s/>

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkley's CS61C

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

The "ABI" Conventions & Mnemonic Registers

- The "Application Binary Interface" defines our 'calling convention'
 - How to call other functions
- A critical portion is "what do registers mean by convention"
 - We have 32 registers, but how are they used
- Who is responsible for saving registers?
 - ABI defines a contract: When you call another function, that function promises **not** to overwrite certain registers
- We also have more convenient names based on this
 - So going forward, no more x3, x6... type notation

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 NAME, USE, CALLING CONVENTION

④

REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller

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

RISC-V Green Card

PSEUDO INSTRUCTIONS

MNEMONIC	NAME	DESCRIPTION	USES
beqz	Branch = zero	$\text{if } R[\text{rs1}] = 0 \text{ PC} = \text{PC} + \{\text{imm}, 1\text{b}0\}$	beq
bnez	Branch \neq zero	$\text{if } R[\text{rs1}] \neq 0 \text{ PC} = \text{PC} + \{\text{imm}, 1\text{b}0\}$	bne
fabs.s, fabs.d	Absolute Value	$F[\text{rd}] = (F[\text{rs1}] < 0) ? -F[\text{rs1}] : F[\text{rs1}]$	fsgnx
fmv.s, fmv.d	FP Move	$F[\text{rd}] = F[\text{rs1}]$	fsgnj
fneg.s, fneg.d	FP negate	$F[\text{rd}] = -F[\text{rs1}]$	fsgnjn
j	Jump	$\text{PC} = \{\text{imm}, 1\text{b}0\}$	jal
jr	Jump register	$\text{PC} = R[\text{rs1}]$	jalr
la	Load address	$R[\text{rd}] = \text{address}$	auipc
li	Load imm	$R[\text{rd}] = \text{imm}$	addi
mv	Move	$R[\text{rd}] = R[\text{rs1}]$	addi
neg	Negate	$R[\text{rd}] = -R[\text{rs1}]$	sub
nop	No operation	$R[0] = R[0]$	addi
not	Not	$R[\text{rd}] = \neg R[\text{rs1}]$	xori
ret	Return	$\text{PC} = R[1]$	jalr
segez	Set = zero	$R[\text{rd}] = (R[\text{rs1}] = 0) ? 1 : 0$	altiu
snez	Set \neq zero	$R[\text{rd}] = (R[\text{rs1}] \neq 0) ? 1 : 0$	altiu

ARITHMETIC CORE INSTRUCTION SET

RV64M Multiply Extension

MNEMONIC	FMT NAME	DESCRIPTION (in Verilog)	NOTE
mul, mulw	R MULtiple (Word)	$R[\text{rd}] = (R[\text{rs1}] * R[\text{rs2}]) \{63:0\}$	1)
mulh	R MULtiple High	$R[\text{rd}] = (R[\text{rs1}] * R[\text{rs2}]) \{127:64\}$	
mulhu	R MULtiple High Unsigned	$R[\text{rd}] = (R[\text{rs1}] * R[\text{rs2}]) \{127:64\}$	2)
mulhsu	R MULtiple upper Half Sign/Uns	$R[\text{rd}] = (R[\text{rs1}] * R[\text{rs2}]) \{127:64\}$	6)
div, divw	R DIVide (Word)	$R[\text{rd}] = (R[\text{rs1}] / R[\text{rs2}])$	1)
divu	R DIVide Unsigned	$R[\text{rd}] = (R[\text{rs1}] / R[\text{rs2}])$	2)
rem, remw	R REMAinder (Word)	$R[\text{rd}] = (R[\text{rs1}] \% R[\text{rs2}])$	1)
remu, remuw	R REMAinder Unsigned (Word)	$R[\text{rd}] = (R[\text{rs1}] \% R[\text{rs2}])$	1,2)

RV64A Atomic Extension

amoadd.w, amoadd.d	R ADD	$R[\text{rd}] = M[R[\text{rs1}]]$ $M[R[\text{rs1}]] = M[R[\text{rs1}]] + R[\text{rs2}]$	9)
amoand.w, amoand.d	R AND	$R[\text{rd}] = M[R[\text{rs1}]]$ $M[R[\text{rs1}]] = M[R[\text{rs1}]] \& R[\text{rs2}]$	9)
amomax.w, amomax.d	R MAXimum	$R[\text{rd}] = M[R[\text{rs1}]]$ if $(R[\text{rs2}] > M[R[\text{rs1}]]) M[R[\text{rs1}]] = R[\text{rs2}]$	9)
amomaxu.w, amomaxu.d	R MAXimum Unsigned	$R[\text{rd}] = M[R[\text{rs1}]]$ if $(R[\text{rs2}] > M[R[\text{rs1}]]) M[R[\text{rs1}]] = R[\text{rs2}]$	2,9)
aminin.w, aminin.d	R MINimum	$R[\text{rd}] = M[R[\text{rs1}]]$ if $(R[\text{rs2}] < M[R[\text{rs1}]]) M[R[\text{rs1}]] = R[\text{rs2}]$	9)
amininu.w, amininu.d	R MINimum Unsigned	$R[\text{rd}] = M[R[\text{rs1}]]$ if $(R[\text{rs2}] < M[R[\text{rs1}]]) M[R[\text{rs1}]] = R[\text{rs2}]$	2,9)
amoor.w, amoor.d	R OR	$R[\text{rd}] = M[R[\text{rs1}]]$ $M[R[\text{rs1}]] = M[R[\text{rs1}]] R[\text{rs2}]$	9)
amoswap.w, amoswap.d	R SWAP	$R[\text{rd}] = M[R[\text{rs1}]]$, $M[R[\text{rs1}]] = R[\text{rs2}]$	9)
amoxor.w, amoxor.d	R XOR	$R[\text{rd}] = M[R[\text{rs1}]]$ $M[R[\text{rs1}]] = M[R[\text{rs1}]] \wedge R[\text{rs2}]$	9)
lr.w, lr.d	R Load Reserved	$R[\text{rd}] = M[R[\text{rs1}]]$, reservation on $M[R[\text{rs1}]]$	
sc.w, sc.d	R Store Conditional	if reserved, $M[R[\text{rs1}]] = R[\text{rs2}]$, $R[\text{rd}] = 0$; else $R[\text{rd}] = 1$	

CORE INSTRUCTION FORMATS

	31	27	26	25	24	20	19	15	14	12	11	7	6	0		
R	funct7							rs2	rs1	funct3			rd	Opcode		
I	imm[11:0]							rs1		funct3			rd	Opcode		
S	imm[11:5]					rs2	rs1	funct3		imm[4:0]			opcode			
SB	imm[12 0:5]					rs2	rs1	funct3		imm[4 1 11]			opcode			
U	imm[31:12]														rd	opcode
UJ	imm[20 10:1 11 19:12]														rd	opcode

REGISTER NAME, USE, CALLING CONVENTION

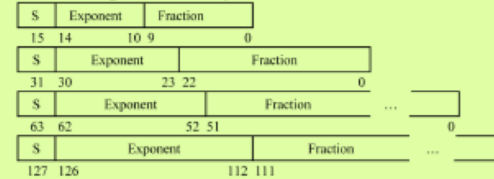
REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Caller
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/sf	Saved register/Frame pointer	Caller
x9	s1	Saved register	Caller
x10-x11	a0-a1	Function arguments/Return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Caller
x28-x31	t3-t6	Temporaries	Caller
t0-t7	ft0-ft7	FP Temporaries	Caller
f8-f9	f80-f81	FP Saved registers	Caller
f10-f11	f80-f83	FP Function arguments/Return values	Caller
f12-f17	f82-f83	FP Function arguments	Caller
f18-f27	f82-f83	FP Saved registers	Caller
f28-f31	ft8-ft11	$R[\text{rd}] = R[\text{rs1}] + R[\text{rs2}]$	Caller

IEEE 754 FLOATING-POINT STANDARD

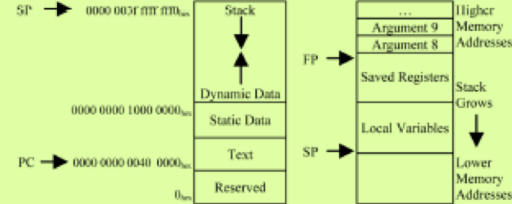
$$(-1)^s \times (1 + \text{Fraction}) \times 2^{(13-\text{exponent} - \text{bias})}$$

where Half-Precision Bias = 15, Single-Precision Bias = 127, Double-Precision Bias = 1023, Quad-Precision Bias = 16383

IEEE Half-, Single-, Double-, and Quad-Precision Formats:



MEMORY ALLOCATION



SIZE PREFIXES AND SYMBOLS

SIZE	PREFIX	SYMBOL	SIZE	PREFIX	SYMBOL
10 ³	Kilo-	K	2 ¹⁰	Kibi-	Ki
10 ⁶	Mega-	M	2 ²⁰	Mebi-	Mi
10 ⁹	Giga-	G	2 ³⁰	Gibi-	Gi
10 ¹²	Tera-	T	2 ⁴⁰	Tebi-	Ti
10 ¹⁵	Peta-	P	2 ⁵⁰	Pebi-	Pi
10 ¹⁸	Exa-	E	2 ⁶⁰	Exbi-	Ei
10 ²¹	Zetta-	Z	2 ⁷⁰	Zebi-	Zi
10 ²⁴	Yotta-	Y	2 ⁸⁰	Yobi-	Yi
10 ⁻³	milli-	m	10 ⁻¹⁵	femto-	f
10 ⁻⁶	micro-	μ	10 ⁻¹⁸	atto-	a
10 ⁻⁹	nano-	n	10 ⁻²¹	zepto-	z
10 ⁻¹²	pico-	p	10 ⁻²⁴	yocto-	y

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

Leaf() from last video:

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
```

We could have optimized...

- We could have just as easily used **t0** and t1 instead...

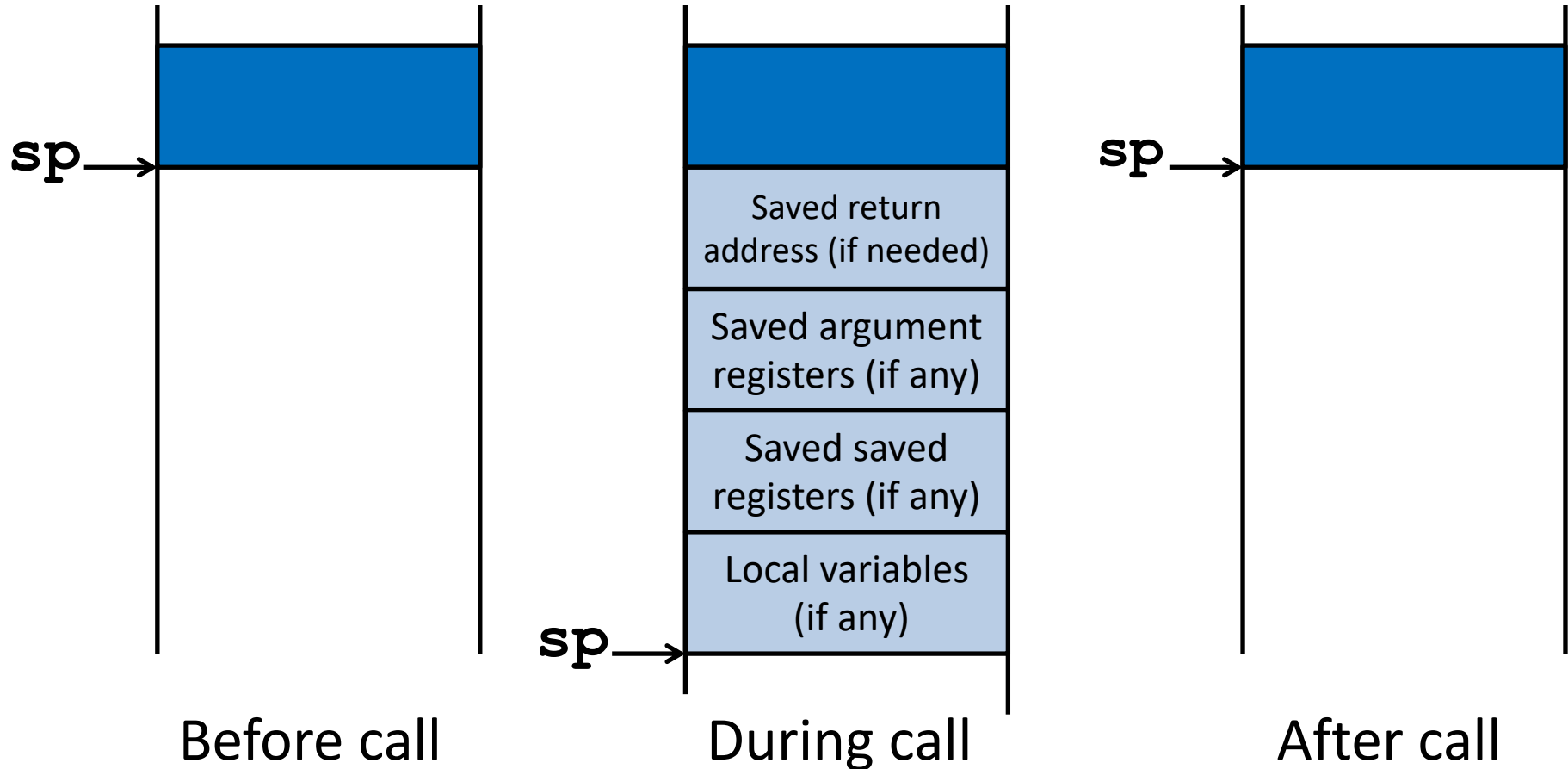
Leaf:

```
add    t0, a0, a1 # t0 = g + h
add    t1, a2, a3 # t1 = i + j
sub    a0, t0, t1 # return value (g + h) - (i + j)
ret                               # short for jalr x0 ra
```

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:

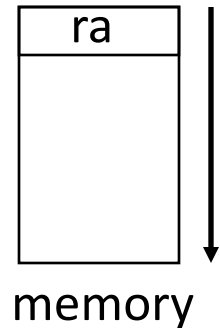
```
“push”  addi    sp, sp, -8    # space on stack  
        sw     ra, 4(sp)   # save ret addr  
        sw     a1, 0(sp)   # save y  
        mv    a1, a0       # mult(x,x)  
        jal   mult        # call mult  
        lw    a1, 0(sp)    # restore y  
“pop”   add    a0, a0, a1   # mult()+y  
 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

A Richer Translation Example

```
1  struct node {
2      unsigned char c, /* c will be at 0, */
3      struct node *next};
4  /* next will be at 4 because of alignment */
5  /* sizeof(struct node) == 8 */
6
7  struct node * foo(char c){
8      struct node *n
9      if(c < 0) return 0;
10     n = malloc(sizeof(struct node));
11     n->next = foo(c - 1);
12     n->c = c;
13     return n;
14 }
```

What is needed?

- We'll need to save **ra**
 - Because we are calling other function
- We'll need a local variable for **c**
 - Because we are calling other functions
 - Lets put this in **s0**
- We'll need a local variable for **n**
 - Lets put this in **s1**
- So lets form the “preamble” and “postamble”
 - What we always do on entering and leaving the function

```
1 foo:
2     addi sp sp -12      # Get stack space for 3 registers
3     sw s0 0(sp)        # Save s0
4     sw s1 4(sp)        # Save s1
5     sw ra 8(sp)        # Save ra
```

Body of function ...

```
21 foo_exit:              # Assume return value already in a0
22     lw s0 0(sp)         # Restore Registers
23     lw s1 4(sp)
24     lw ra 8(sp)
25     addi sp sp 12      # Restore stack pointer
26     ret                # aka.. jalr x0 ra
```

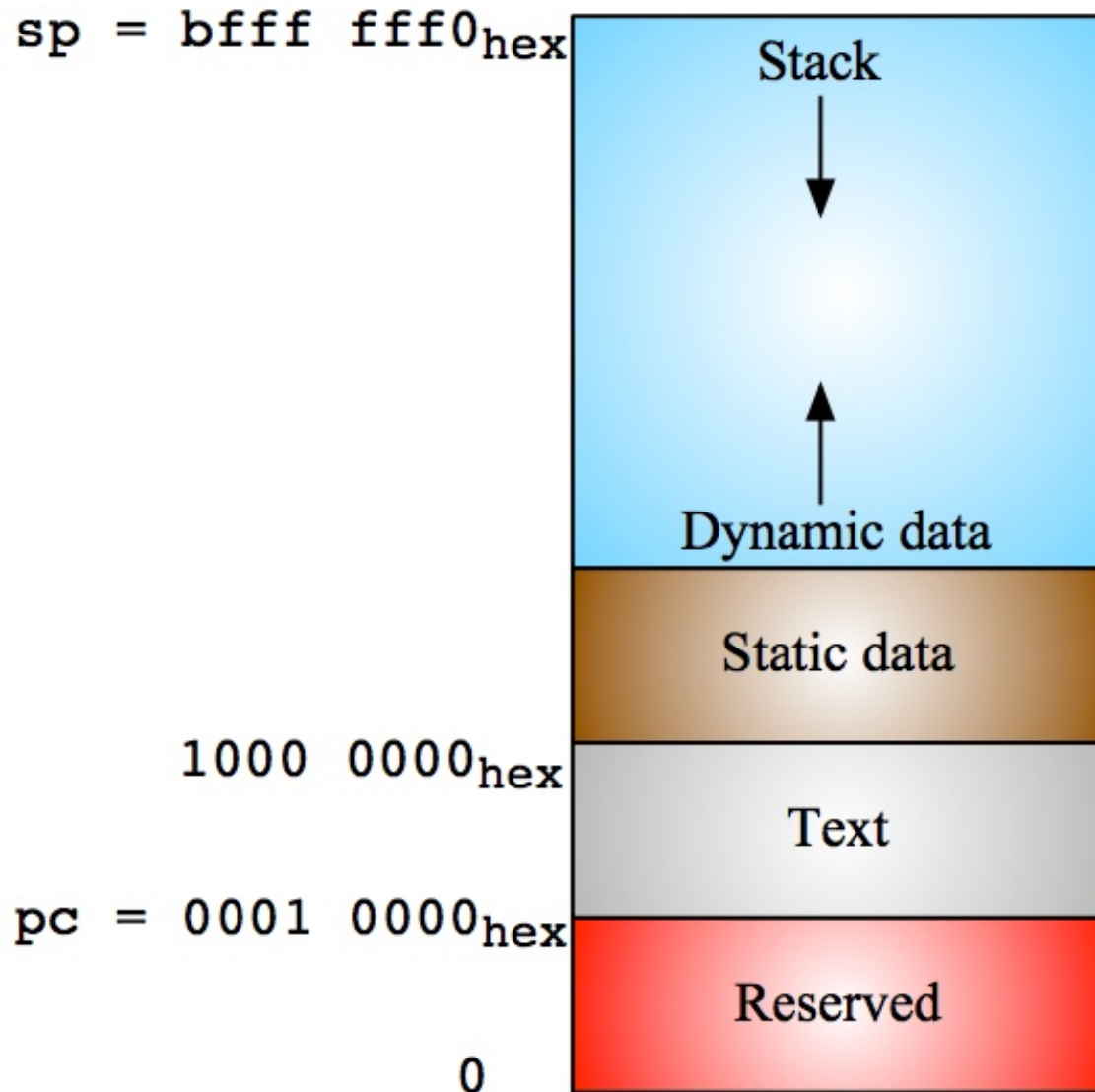
Again, we skipped a lot of optimization...

- On the leaf node ($c < 0$) we didn't need to save **ra** (or even **s0** & **s1** since we don't need to use them)
- We could get away with only one saved register..
 - Save c into **s0**
 - call **malloc**
 - save c into $n[0]$
 - calc $c-1$
 - save n in $s0$
 - recursive call
- But again, we don't needlessly optimize...

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 (Almost 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!