CS 110
Computer Architecture
Lecture 6:
RISC-V Instruction Formats

Instructors:
Sören Schwertfeger & Chundong Wang

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkley's CS61C

Instruction Support for Functions (2/4)

C

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

RISC-V

```
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 (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: jalr

- 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

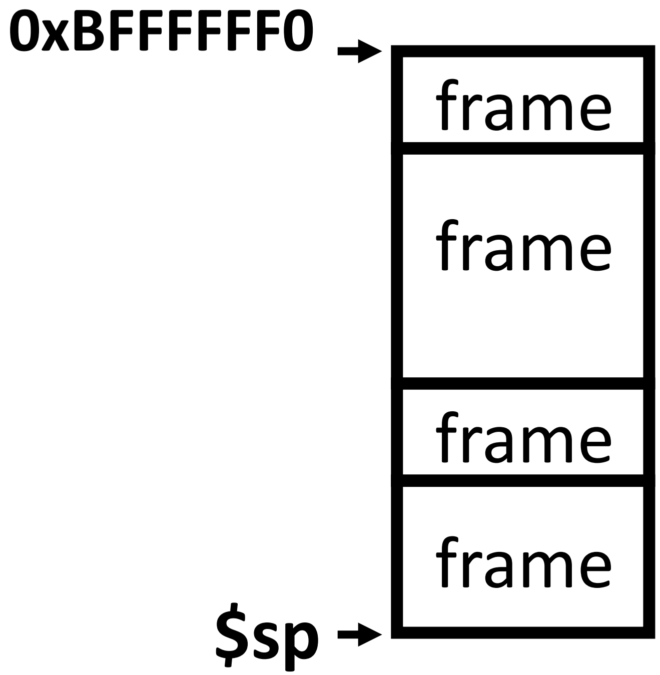
- Calling program (*caller*) puts parameters into registers `a0-a7` and uses `jal` X to invoke (*callee*) at address labeled X
- Must have register in computer with address of currently executing instruction
  - Instead of *Instruction Address Register* (better name), historically called *Program Counter* (*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

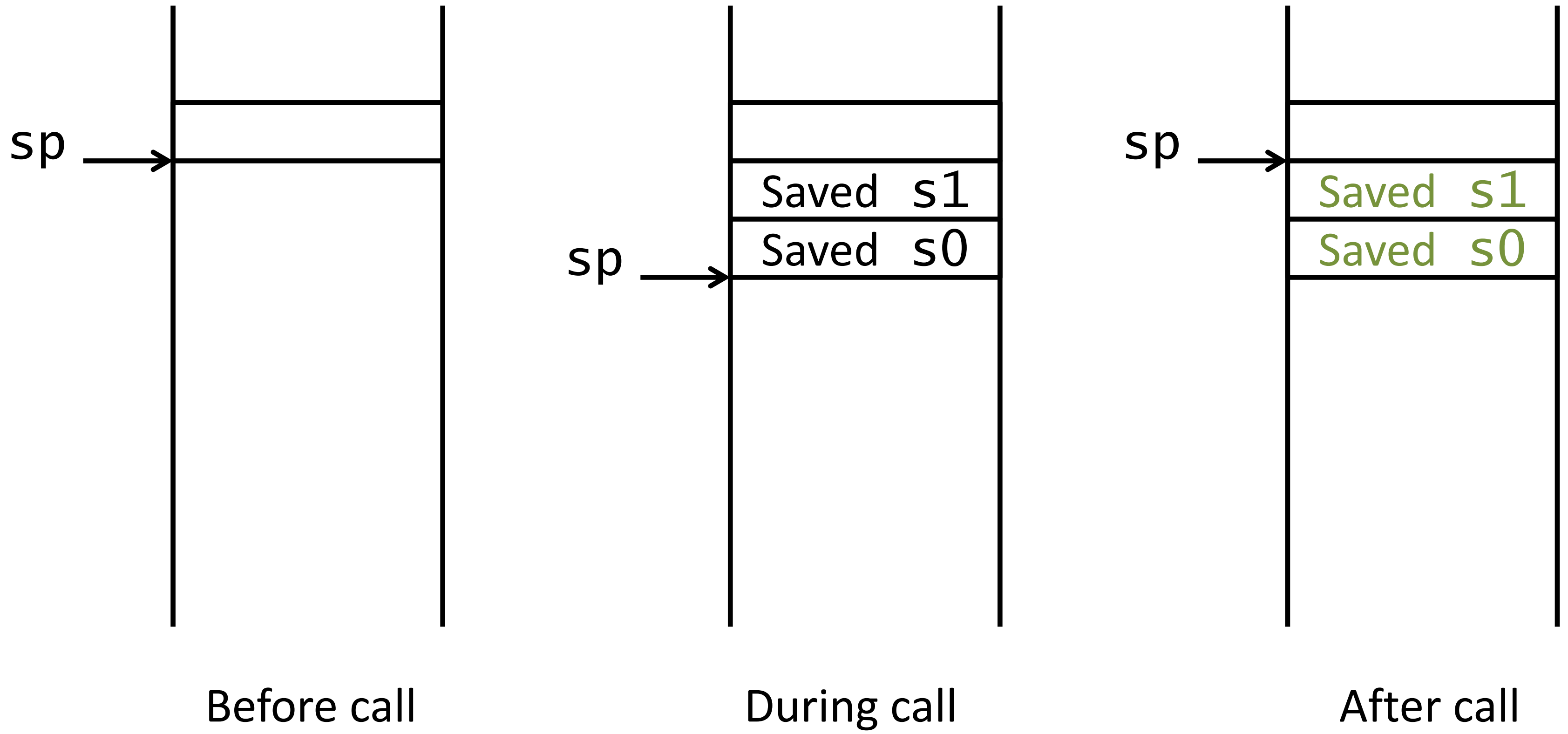
Leaf Function Example

```
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`



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

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.

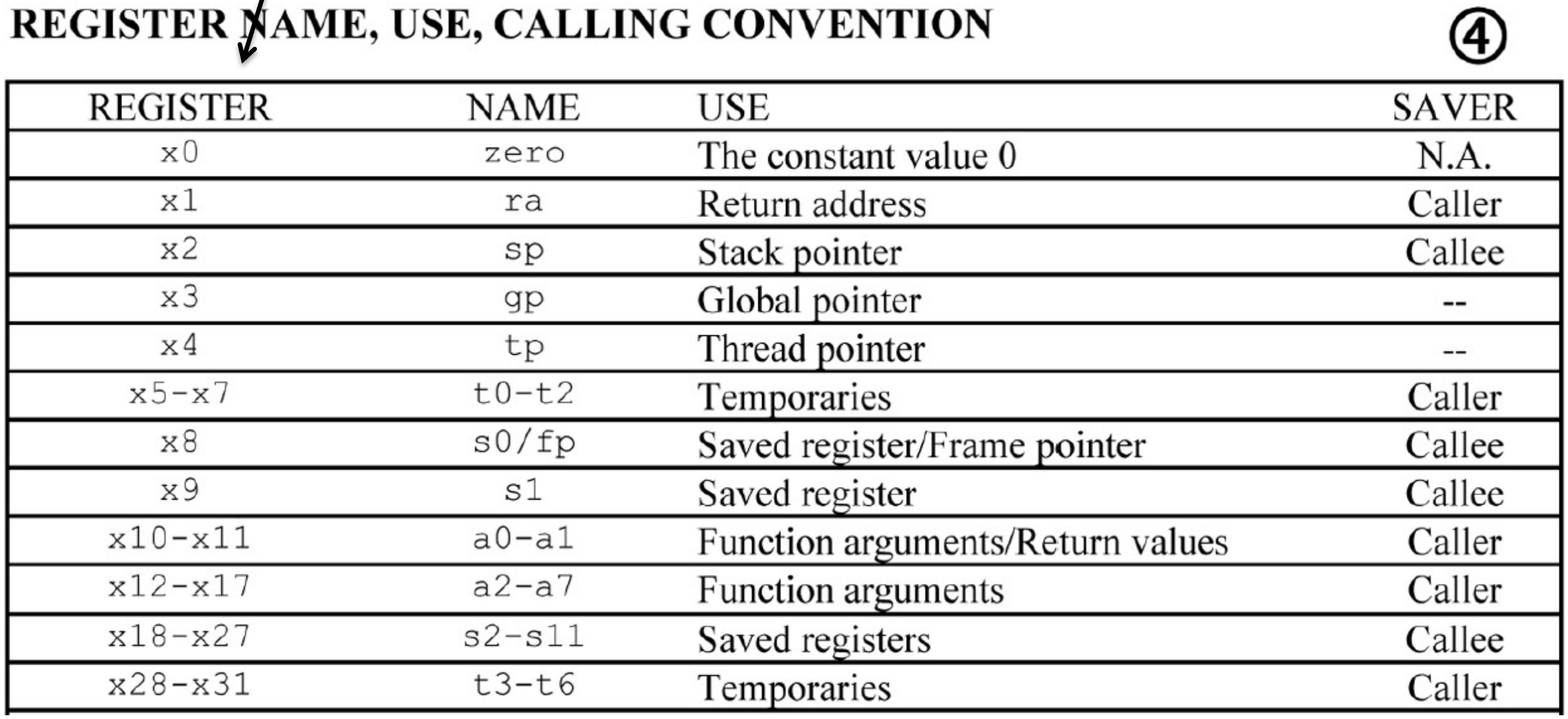
Register Conventions (2/2)

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

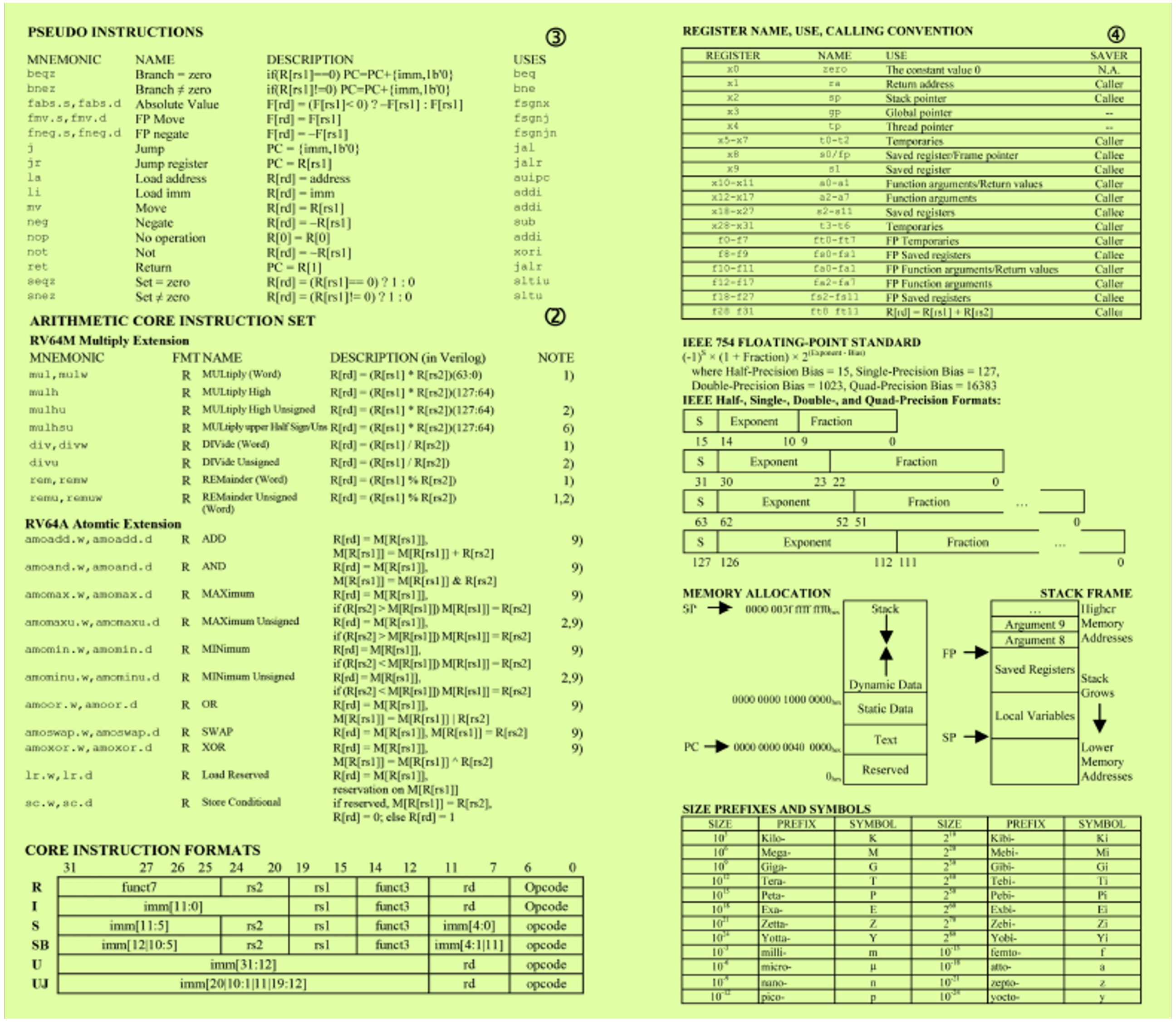


REGISTER NAME, USE, CALLING CONVENTION ④

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 ③

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

ARITHMETIC CORE INSTRUCTION SET ②

RV64M Multiply Extension

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

RV64A Atomtic Extension

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
`amoadd.w,amoadd.d`	R	ADD	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] + R[rs2]	9)
`amoand.w,amoand.d`	R	AND	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] & R[rs2]	9)
`amomax.w,amomax.d`	R	MAXimum	R[rd] = M[R[rs1]], if (R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
`amomaxu.w,amomaxu.d`	R	MAXimum Unsigned	R[rd] = M[R[rs1]], if (R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
`amomin.w,amomin.d`	R	MINimum	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
`amominu.w,amominu.d`	R	MINimum Unsigned	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
`amoor.w,amoor.d`	R	OR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] | R[rs2]	9)
`amoswap.w,amoswap.d`	R	SWAP	R[rd] = M[R[rs1]], M[R[rs1]] = R[rs2]	9)
`amoxor.w,amoxor.d`	R	XOR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] ^ R[rs2]	9)
`lr.w,lr.d`	R	Load Reserved	R[rd] = M[R[rs1]], reservation on M[R[rs1]]	
`sc.w,sc.d`	R	Store Conditional	if reserved, M[R[rs1]] = R[rs2], R[rd] = 0; else R[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|10:5]		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 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
`f0-f7`	`ft0-ft7`	FP Temporaries	Caller
`f8-f9`	`fs0-fs1`	FP Saved registers	Callee
`f10-f11`	`fa0-fa1`	FP Function arguments/Return values	Caller
`f12-f17`	`fa2-fa7`	FP Function arguments	Caller
`f18-f27`	`fs2-fs11`	FP Saved registers	Callee
`f28 f31`	`ft8 ft11`	R[rd] = R[rs1] + R[rs2]	Caller

IEEE 754 FLOATING-POINT STANDARD

$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent - Bias})}$

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:

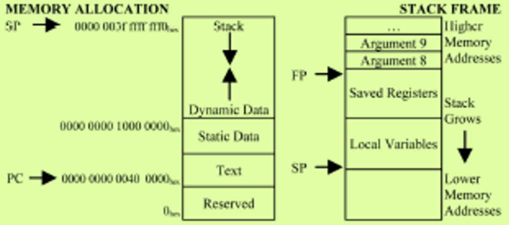
S	Exponent	Fraction
15	14 10	9 0

S	Exponent	Fraction
31	30 23	22 0

S	Exponent	Fraction …
63	62 52	51 0

S	Exponent	Fraction …
127	126 112	111 0

MEMORY ALLOCATION **STACK FRAME**



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 before:

```
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…

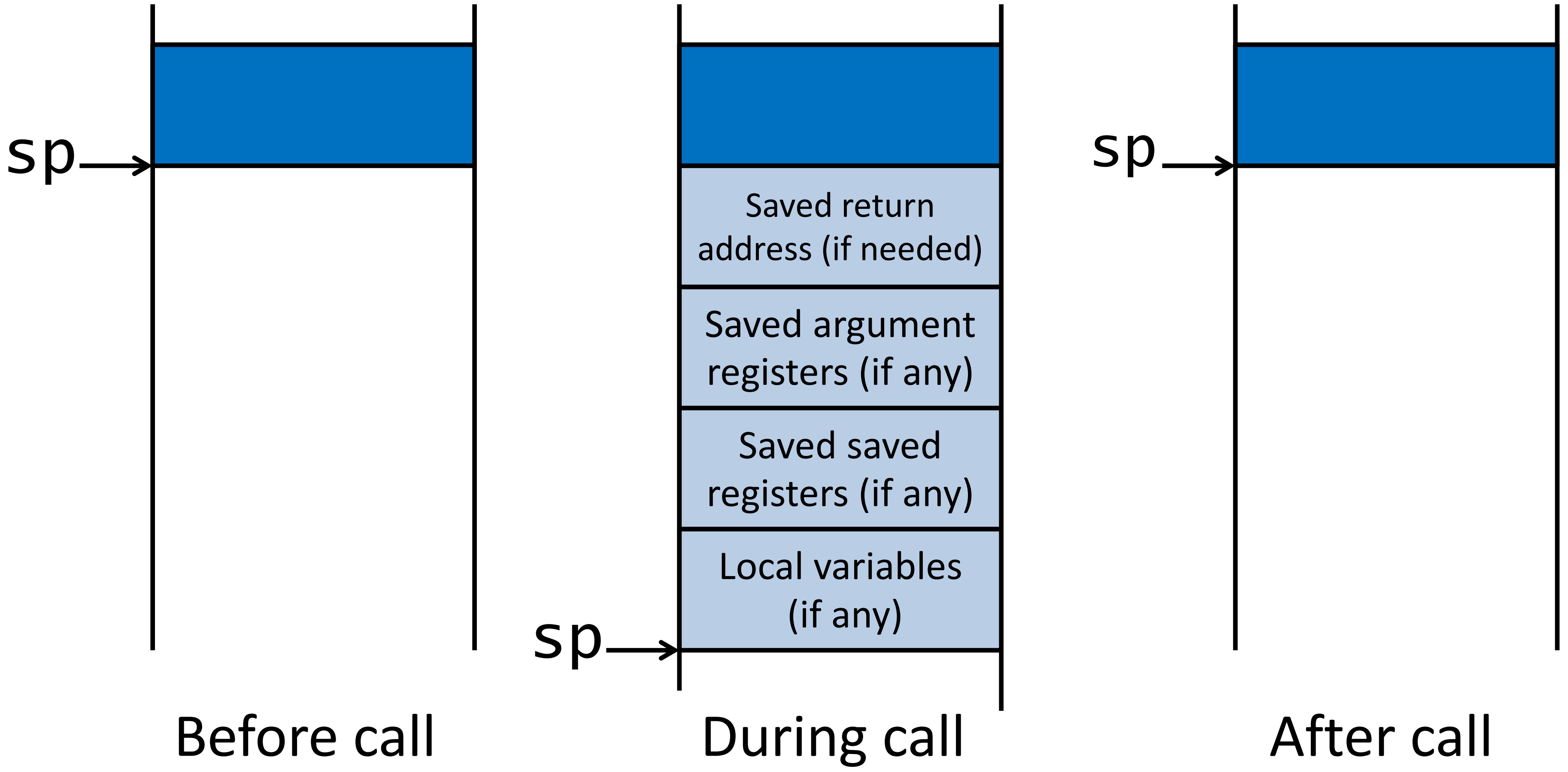
- We could have just as easily used **t0** and **t1** instead…

```
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


sp
sp
Saved return address (if needed)
Saved argument registers (if any)
Saved saved registers (if any)
Local variables (if any)
sp
Before call
During call
After call

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)

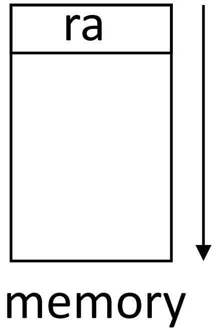
```
int sumSquare(int x, int y) {
  return mult(x,x)+ y; }
```

```
sumSquare:
         addi   sp, sp, -8   # space on stack
"push"   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…)**

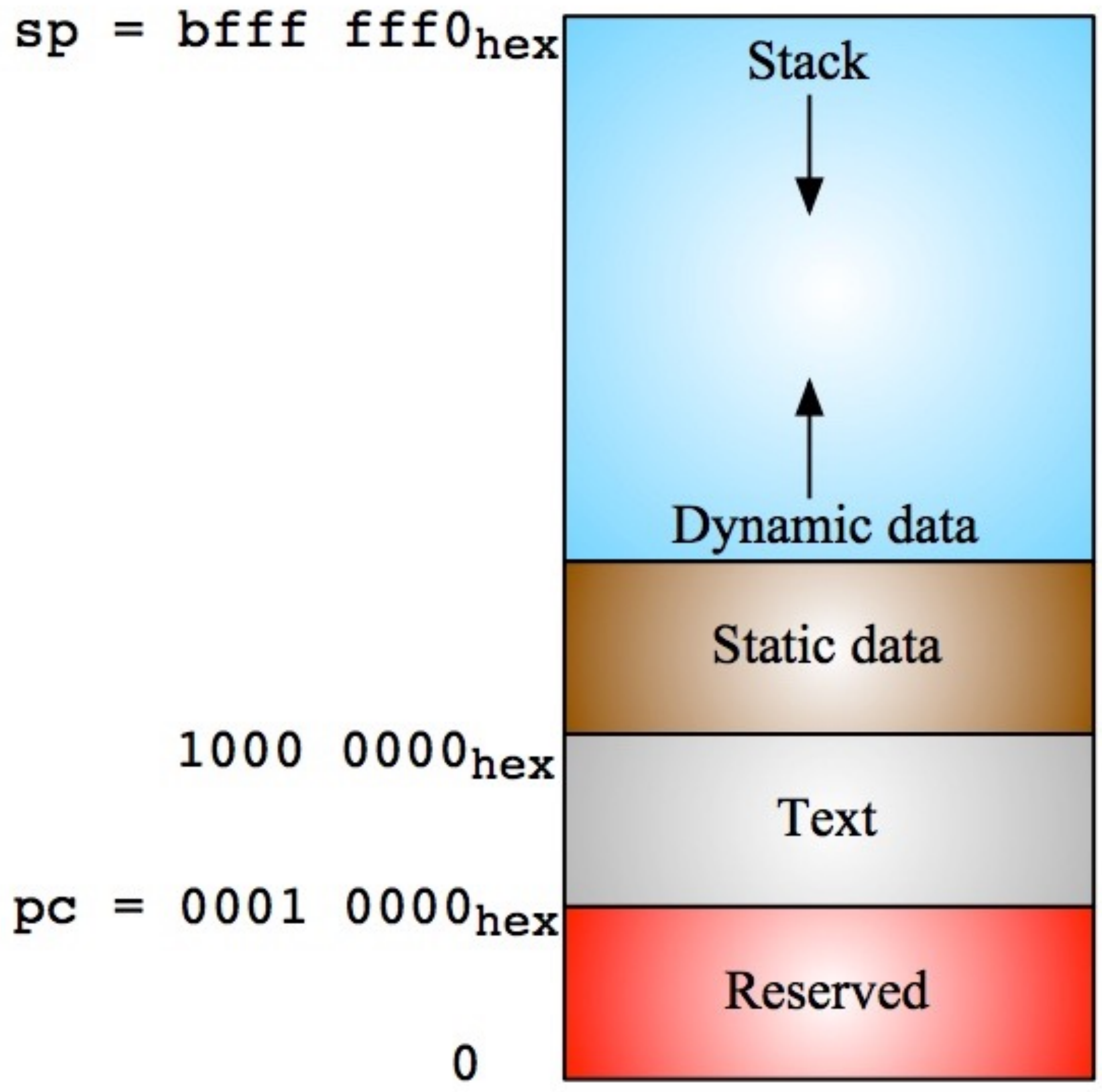
Epilogue

```
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 fff0`$_{hex}$
  - Stack must be aligned on 16-byte boundary (not true in examples above)
- RV32 programs (*text segment*) in low end
  - 0001 0000$_{hex}$
- *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 0000$_{hex}$
- *Heap* above static for data structures that grow and shrink ; grows up to high addresses

RV32 Memory Allocation



Frame Pointer!?

- As a reminder, we shove all the C local variables etc. on the stack...
  - Combined with space for all the saved registers
  - This is called the "activation record" or "call frame" or "call record”
- But a naive compiler may cause the stack pointer to bounce up and down during a function call
  - Can be a lot simpler to have a compiler do a bunch of pushes and pops when it needs a bit of temporary space: more so on a CISC rather than a RISC however
- Plus: not all programming languages can store all activation records on the stack:
  - The use of lambda in Scheme, Python, Go, etc. requires that some call frames are allocated on the heap since variables may last beyond the function call!

Convention: Use **s0** as a Frame Pointer (**fp**)

- At the start, save **s0 (x8)** and then have the Frame pointer point to one below the sp when you were called...

```
addi sp sp -20 # Initially grabbing 5 words of space
sw ra 16(sp)   #
sw fp 12(sp)   # save fp/s0/x8
addi fp sp 20  # Points to the start of this call record
…
```

- Now we can address local variables off the frame pointer rather than the stack pointer
  - Simplifies the compiler
    - Since it can now move the stack up and down easily
  - Simplifies the ***debugger***

But note...

- It isn't necessary in C...
  - Most C compilers has a -f-omit-frame-pointer option on most architectures
    - It just fubars debugging a bit
- So for our hand-written assembly, we will generally ignore the frame pointer
- The calling convention says it doesn't matter if you use a frame pointer or not!
  - It is just a callee saved register, so if you use it as a frame pointer...
    It will be preserved just like any other saved register
    But if you just use it as **s0**, that makes no difference!

The Stack Is Also For Local Variables...

- e.g. **char[20] foo;**

- Requires enough space on the stack
  - May need padding

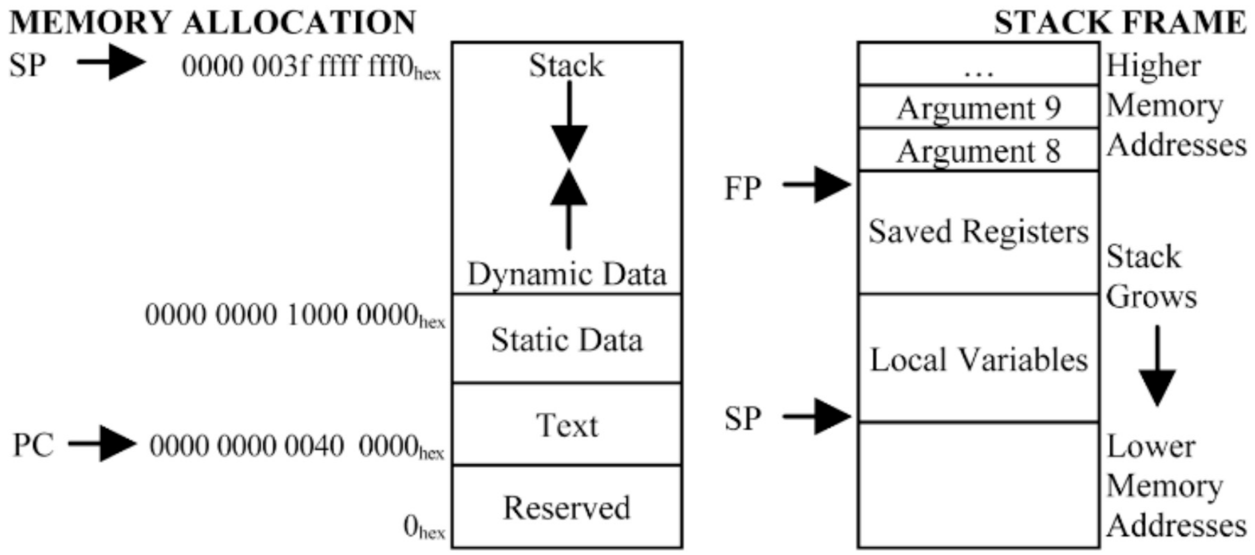
- So then to pass **foo** to something in **a0**...

```
addi a0 sp offset-for-foo-off-sp
addi a0 fp offset-for-foo-off-fp
```

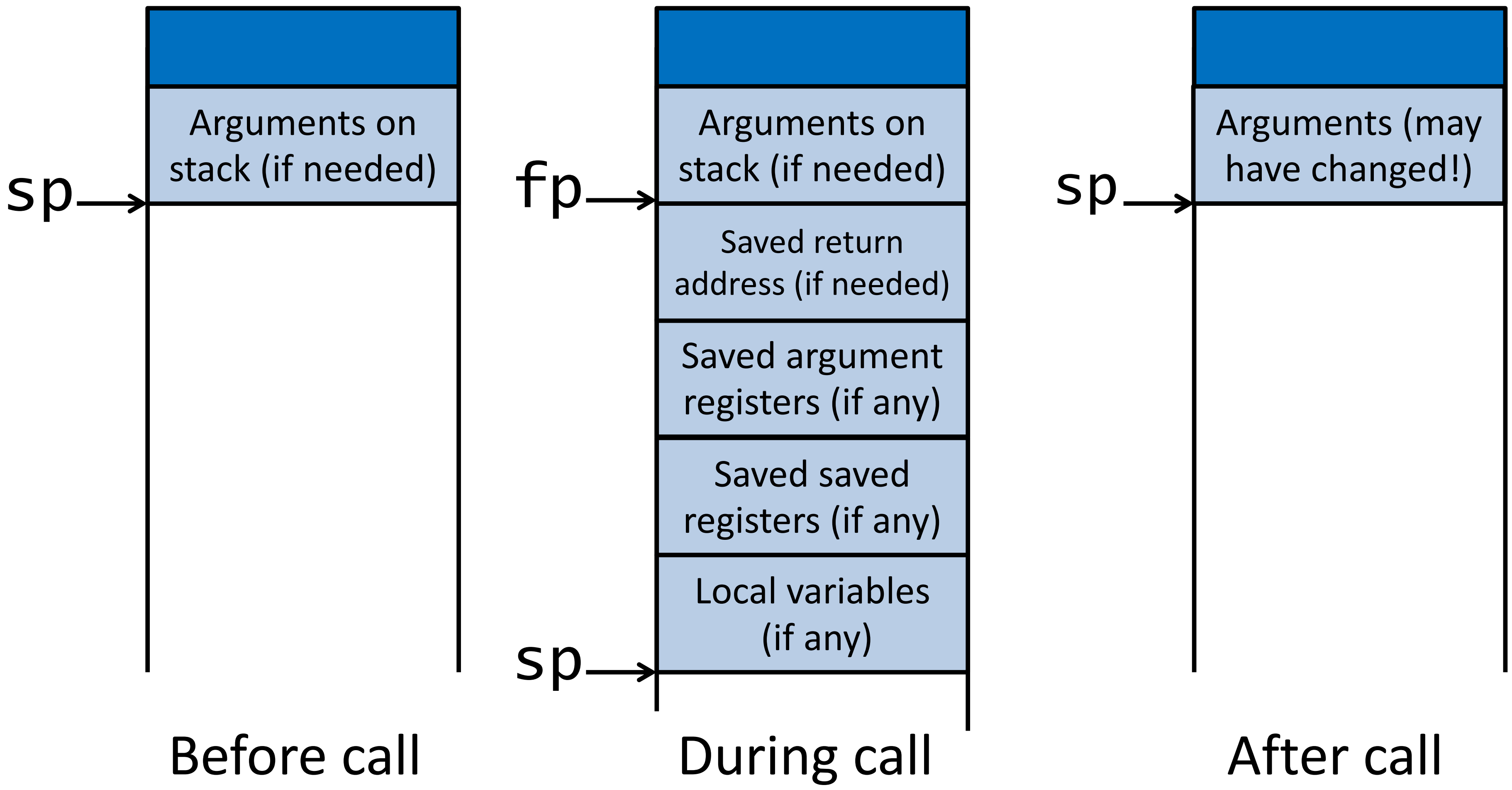
  - If you are using the frame pointer...

The Stack Is Also For Arguments

- Arguments 1-8 are passed in **a0-a7**
- But what about a 9th argument or more?
- But what about complex structs as arguments?
  - Pass those on the stack!
  - When the function is called,
    - `0(sp) -> arg #9`
      `4(sp) -> arg #10...`
- ALWAYS keep sp the lowest address used!
  - Because: Interrupts may use your stack!
  - => Arguments are in the frame of the caller!
- Don't need to memorize this for exams



Stack Before, During, After Function

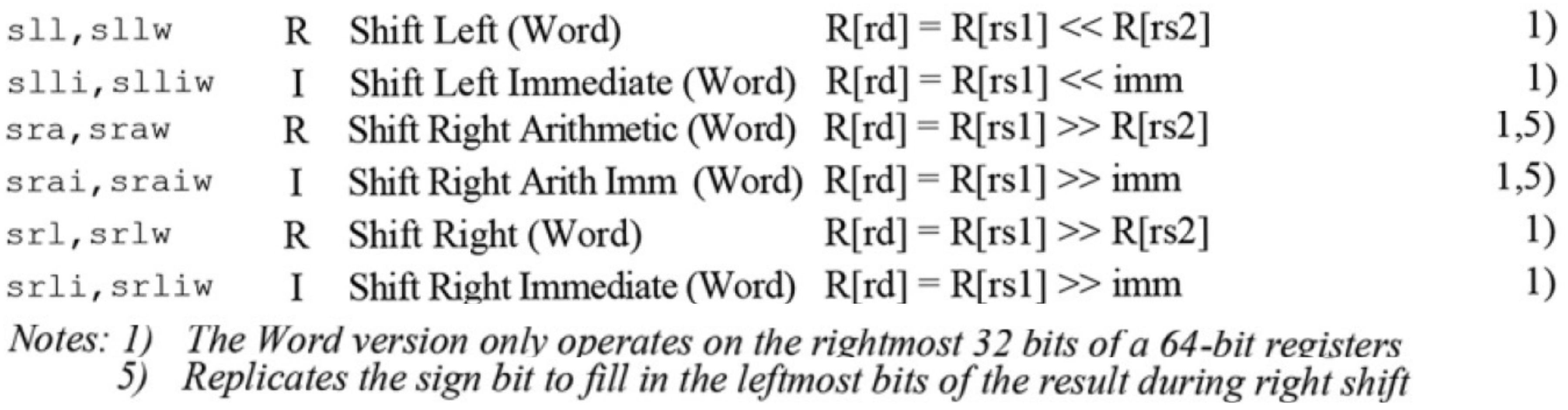

Arguments on stack (if needed)
sp
Before call
Arguments on stack (if needed)
fp
Saved return address (if needed)
Saved argument registers (if any)
Saved saved registers (if any)
Local variables (if any)
sp
During call
Arguments (may have changed!)
sp
After call

Register Allocation

- We have some set of registers that are useful for local variables, temporaries that last across function calls, etc...
- We have some other set of registers that are just for temporary use
- Which ones do we use? What do we instead save on the stack?
- This is the "Register Allocation" problem
  - Experience it in great detail in CS 131 Compilers ...
- Can either be trivial or NP-complete!

12 Shift Instructions…

- Two versions of of all shift instructions. Shift amount via:
  - Register
  - Immediate
- (On RV64: additional "word" version of instruction: only works on first 32bit of 64bit register)
- Shift Left
- Shift Right Arithmetic: Fill upper bits with **msb**
- Shift Right Logic: Fill upper bits with 0's



`sll,sllw`	R	Shift Left (Word)	R[rd] = R[rs1] << R[rs2]	1)
`slli,slliw`	I	Shift Left Immediate (Word)	R[rd] = R[rs1] << imm	1)
`sra,sraw`	R	Shift Right Arithmetic (Word)	R[rd] = R[rs1] >> R[rs2]	1,5)
`srai,sraiw`	I	Shift Right Arith Imm (Word)	R[rd] = R[rs1] >> imm	1,5)
`srl,srlw`	R	Shift Right (Word)	R[rd] = R[rs1] >> R[rs2]	1)
`srli,srliw`	I	Shift Right Immediate (Word)	R[rd] = R[rs1] >> imm	1)

Notes: 1) The Word version only operates on the rightmost 32 bits of a 64-bit registers
5) Replicates the sign bit to fill in the leftmost bits of the result during right shift

ADMIN

Admin

- HW 2: due next Tuesday – if you didn’t start yet, start today!
- Project 1.1 will be posted next Tuesday!
  - Work together with your partner!
  - Push to gitlab very often – every day you work on the project – even if it doesn’t complie!
  - <u>We will evaluate each partners contribution based on gitlab statistics</u>!

- Venus Tutorial Videos will be available on the website.
  - From 2020TA Ze Song

Admin

- Midterm I and II Dates:
  - March 29
  - May 3 (may change due to holiday!)
  - During lecture hours (8:15 **sharp** – 10:00)
  - Rooms: tbd.
- Midterm I content:
  - Everything till (including): RISC-V Datapath
  - Material: 1 A4 cheat-sheet handwritten by you

Levels of Representation/Interpretation

High Level Language Program (e.g., C)

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

Compiler

Assembly Language Program (e.g., RISC-V)

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

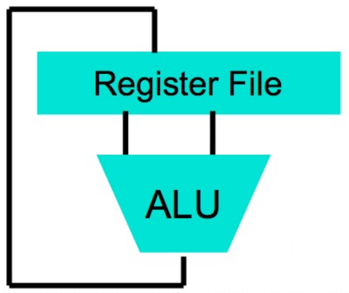
Assembler

Machine Language Program (RISC-V)

```
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
```

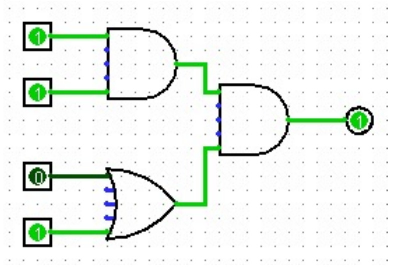
Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)



Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



Big Idea: Stored-Program Computer

First Draft of a Report on the EDVAC
by
John von Neumann
Contract No. W–670–ORD–4926
Between the
United States Army Ordnance Department and the
University of Pennsylvania
Moore School of Electrical Engineering
University of Pennsylvania

June 30, 1945

- Instructions are represented as bit patterns - can think of these as numbers
- Therefore, entire programs can be stored in memory to be read or written just like data
- Can reprogram quickly (seconds), don't have to rewire computer (days)
- Known as the "von Neumann" computers after widely distributed tech report on EDVAC project
  - Wrote-up discussions of Eckert and Mauchly
  - Anticipated earlier by Turing and Zuse

Consequence #1: Everything Addressed

- Since all instructions and data are stored in memory, everything has a memory address: instructions, data words
  - both branches and jumps use these
- C pointers are just memory addresses: they can point to anything in memory
  - Unconstrained use of addresses can lead to nasty bugs; up to you in C; limited in Java by language design
- One register keeps address of instruction being executed: **“Program Counter” (PC)**
  - Basically a pointer to memory: Intel calls it Instruction Pointer (a better name)

Consequence #2: Binary Compatibility

- Programs are distributed in binary form
  - Programs bound to specific instruction set
  - Different version for ARM (phone) and PCs
- New machines want to run old programs ("binaries") as well as programs compiled to new instructions
- Leads to "backward-compatible" instruction set evolving over time
- Selection of Intel 8086 in 1981 for 1st IBM PC is major reason latest PCs still use 80x86 instruction set; could still run program from 1981 PC today

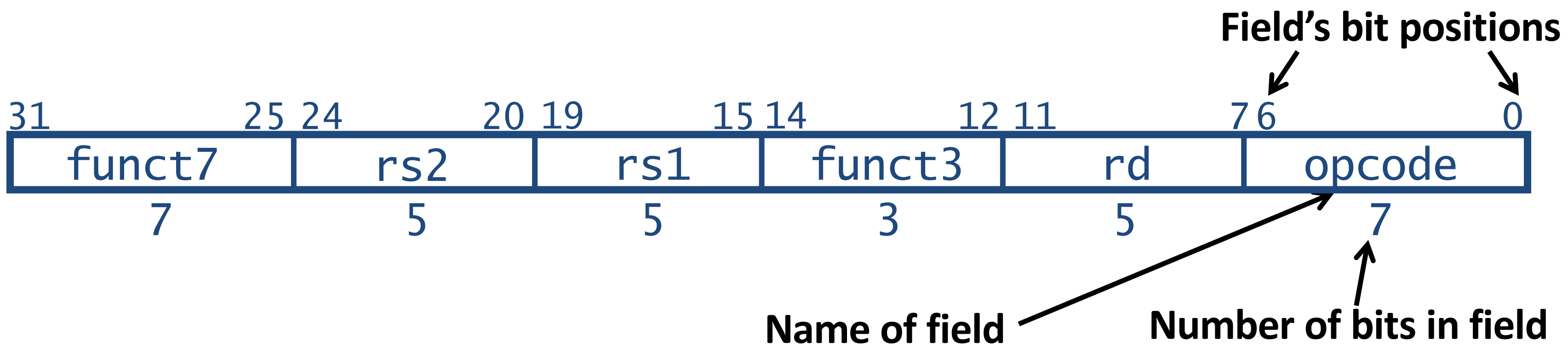
Instructions as Numbers (1/2)

- Currently most data we work with is in words (32-bit chunks):
  - Each register is a word.
  - `lw` and `sw` both access memory one word at a time.
- So how do we represent instructions?
  - Remember: Computer only understands 1s and 0s, so "`add x10,x11,x0`" is meaningless.
  - RISC-V seeks simplicity: since data is in words, make instructions be fixed-size 32-bit words, too
    - Same 32-bit instructions used for RV32, RV64, RV128

Instructions as Numbers (2/2)

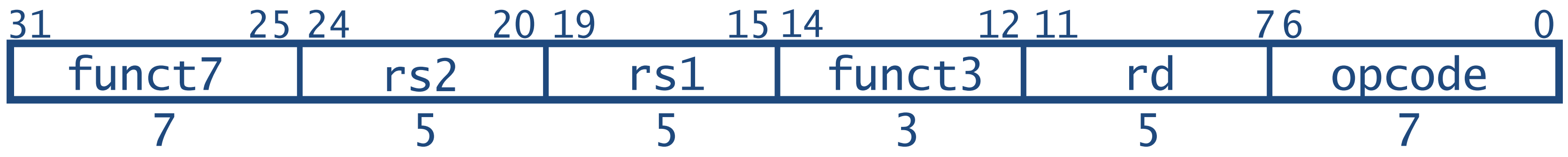
- One word is 32 bits, so divide instruction word into "fields".
- Each field tells processor something about instruction.
- We could define different fields for each instruction, but RISC-V seeks simplicity, so define 6 basic types of instruction formats:
  - `R-format` for register-register arithmetic operations
  - `I-format` for register-immediate arithmetic operations and loads
  - `S-format` for stores
  - `B-format` for branches (minor variant of S-format, called SB before)
  - `U-format` for 20-bit upper immediate instructions
  - `J-format` for jumps (minor variant of U-format, called UJ before)

R-Format Instruction Layout



- 32-bit instruction word divided into six fields of varying numbers of bits each: 7+5+5+3+5+7 = 32
- Examples
  - opcode is a 7-bit field that lives in bits 6-0 of the instruction
  - rs2 is a 5-bit field that lives in bits 24-20 of the instruction

R-Format Instructions opcode/funct fields



- `opcode`: partially specifies what instruction it is
  - Note: This field is equal to `0110011`$_{\text{two}}$ for all R-Format register-register arithmetic instructions
- `funct7+funct3`: combined with `opcode`, these two fields describe what operation to perform

- **Question: You have been professing simplicity, so why aren't `opcode` and `funct7` and `funct3` a single 17-bit field?**
  - **We'll answer this later**

R-Format Instructions register specifiers

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7

- <u>`rs1`</u> (Source Register #1): specifies register containing first operand
- <u>`rs2`</u> : specifies second register operand
- <u>`rd`</u> (Destination Register): specifies register which will receive result of computation
- Each register field holds a 5-bit unsigned integer (0-31) corresponding to a register number (`x0`-`x31`)