Georgia Tech Computer Science

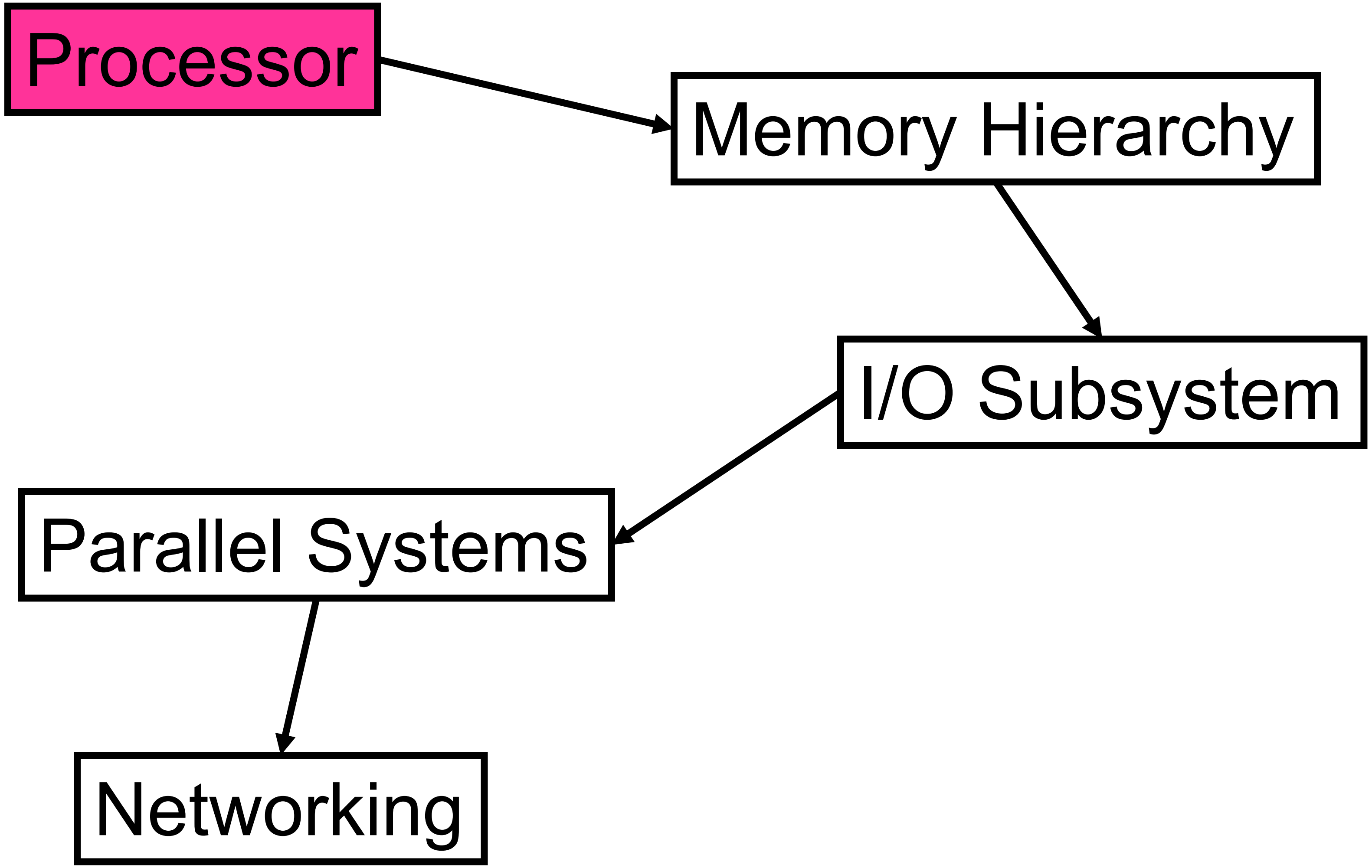
CS2200
Systems and Networks
Spring 2022

Lecture 1: Processors

Alexandros (Alex) Daglis
School of Computer Science
Georgia Institute of Technology
adaglis@gatech.edu

Lecture slides adapted from Leahy, Lively, Ramachandran of Georgia Tech

Our Road Map



Processor

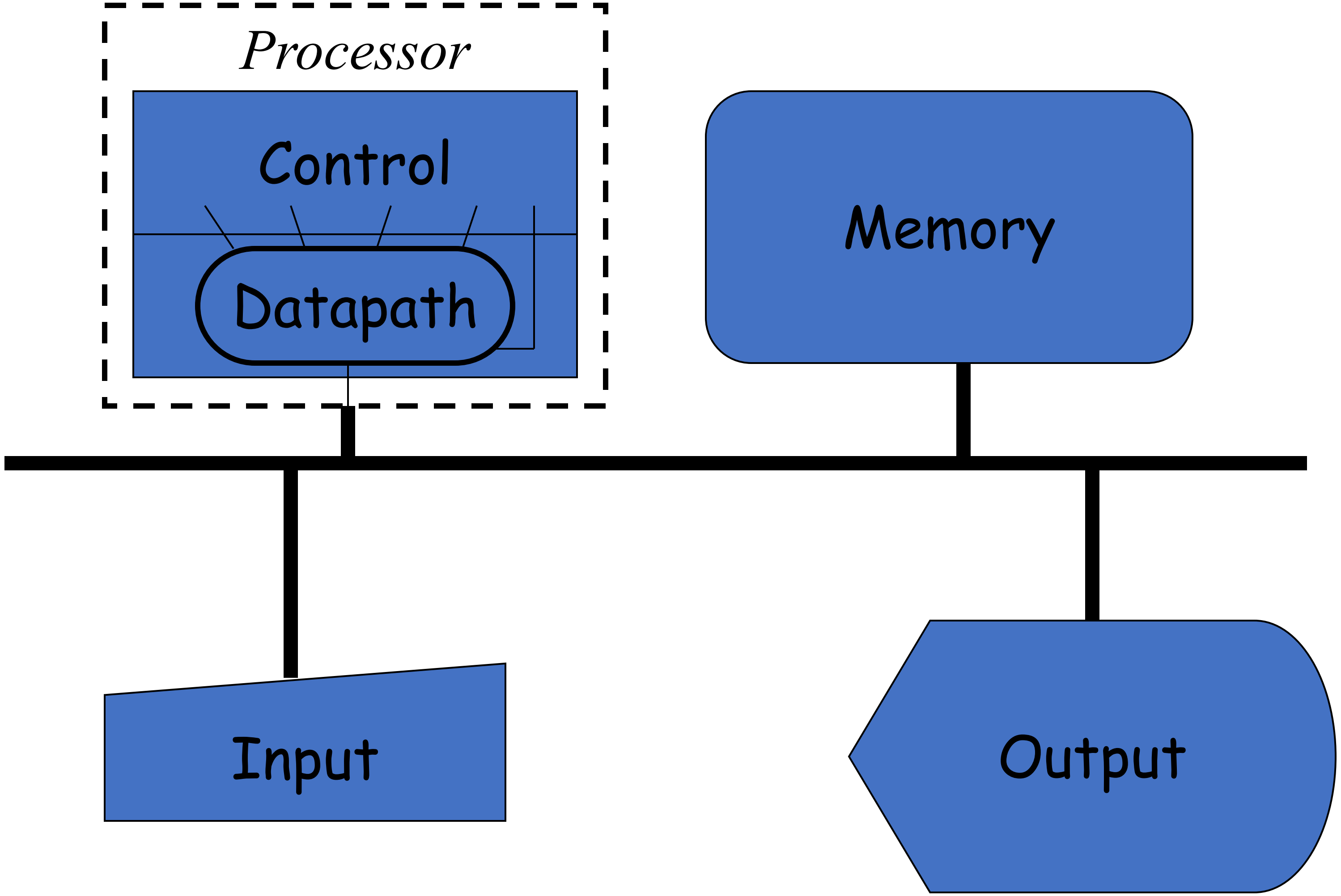
Memory Hierarchy

I/O Subsystem

Parallel Systems

Networking

Five Classic Components

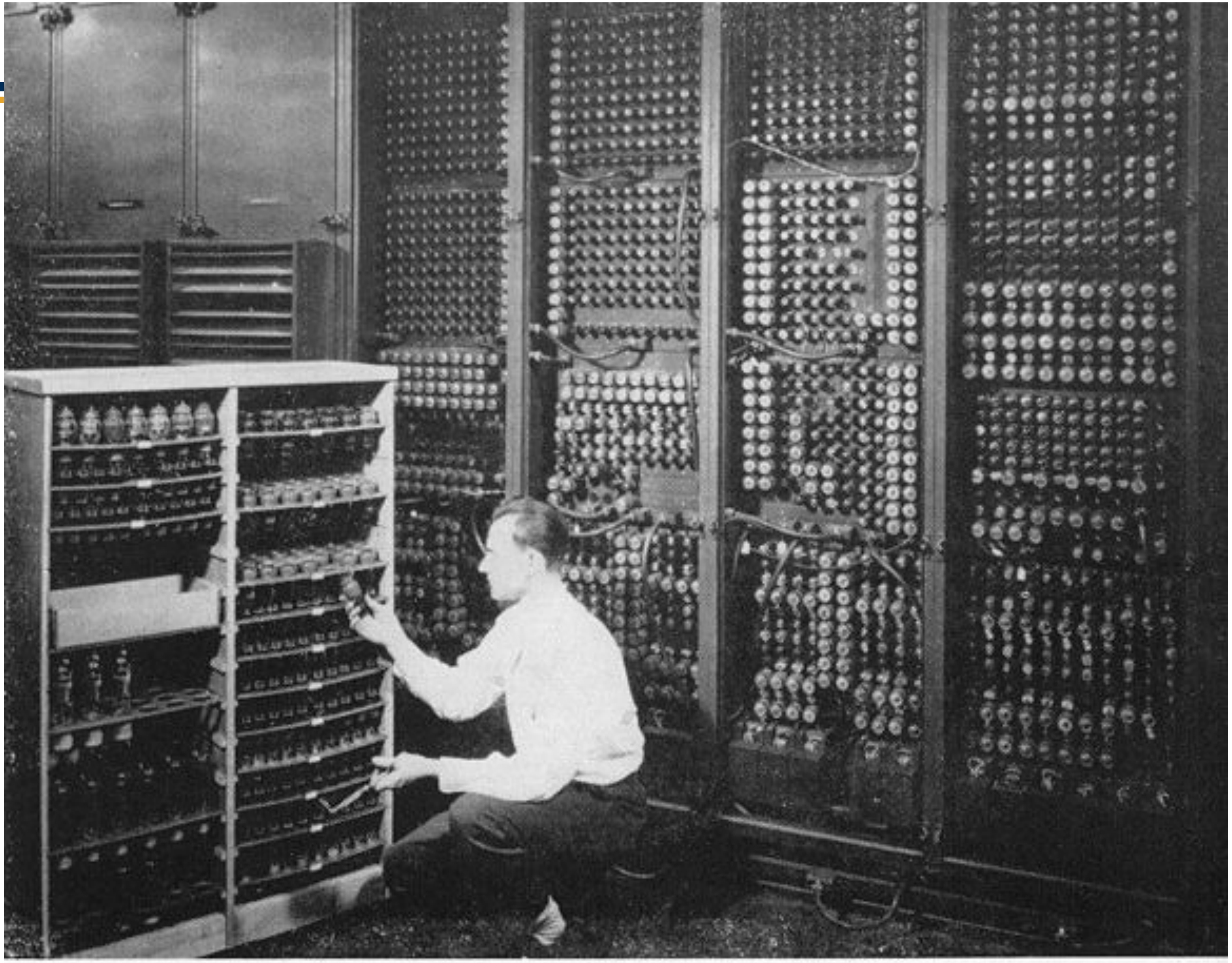


What does the processor do?

- Knows where it is in program
- Can get and put data into memory
- Can do some arithmetic
- Can make tests and take different paths depending on the results

- Do you need a language to make a computer run?

A Little History



Replacing a bad tube meant checking among ENIAC's 19,000 possibilities.

A Little History

- First computers programmed by hand

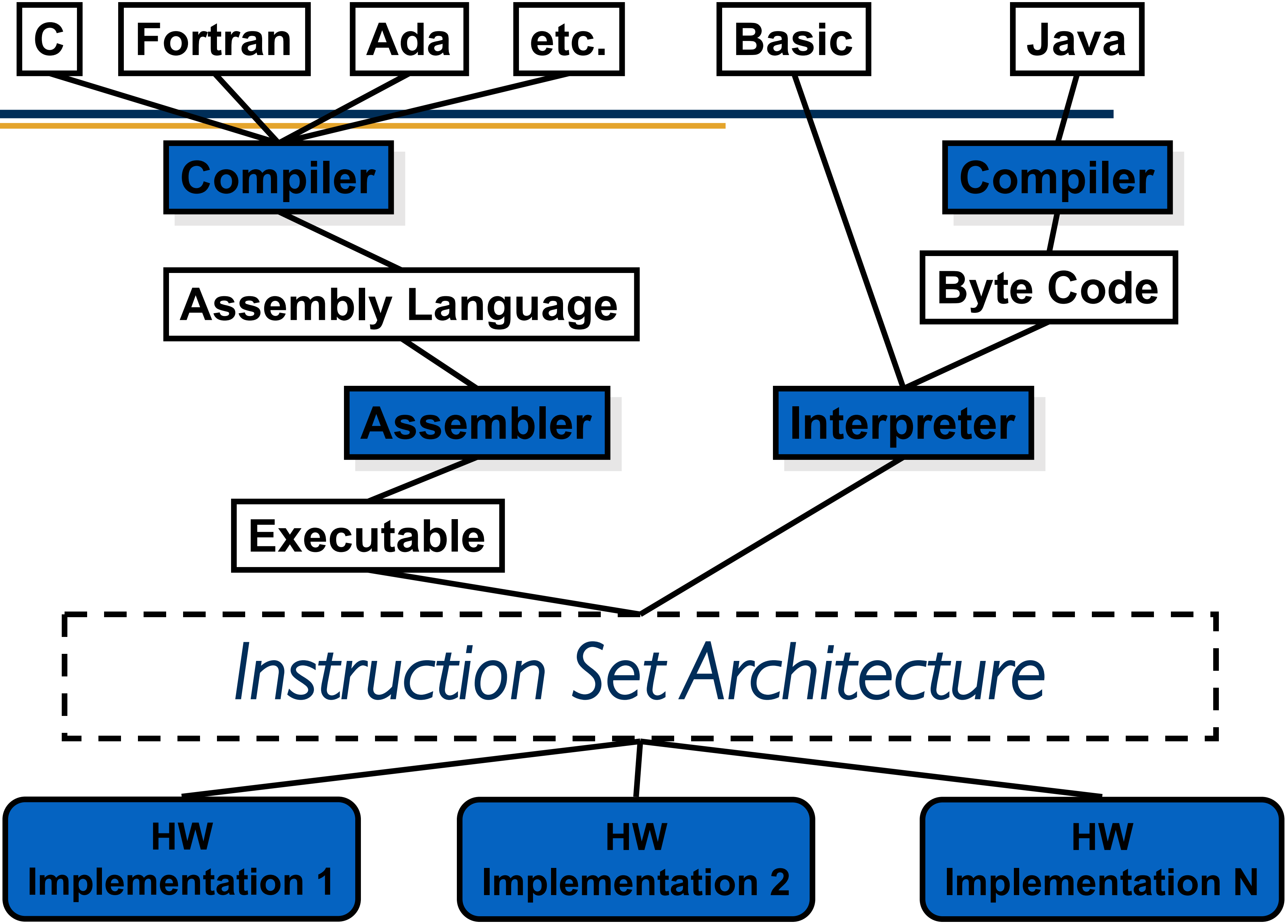
  1000110010100000

- Somewhat tedious, so invented:
- Assembler

  add A,B

- If we can convert from Assembly Language to machine code why not from some higher level language to Assembler?

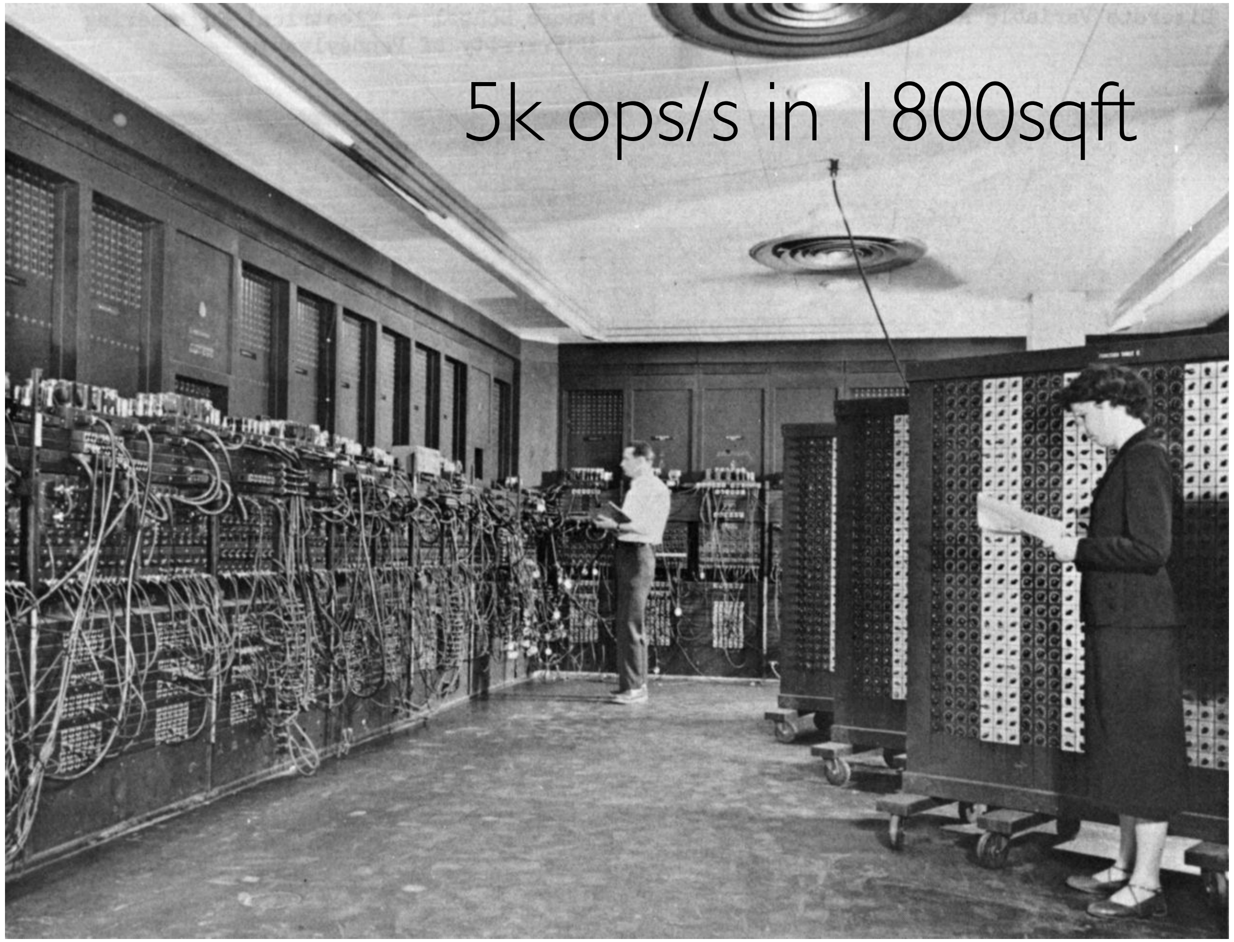
  A + B


C
Fortran
Ada
etc.
Basic
Java
Compiler
Compiler
Assembly Language
Byte Code
Assembler
Interpreter
Executable
Instruction Set Architecture
HW Implementation 1
HW Implementation 2
HW Implementation N

Instructions

- Language of the machine
- Vocabulary is the instruction set (ISA)
- Two levels
  - Human readable (assembly)
  - Machine readable (machine code)

Computing Evolution in 70 Years



5k ops/s in 1800sqft

5T ops/s in 16sq in

1 billion ENIACs in your palm

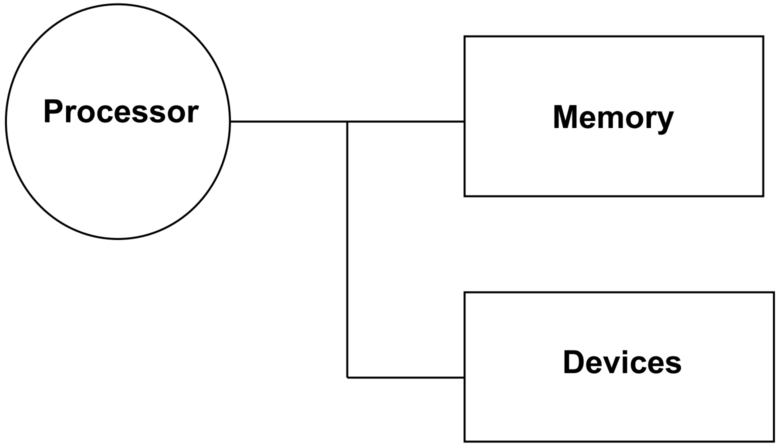
16 trillion times higher compute density

Moving forward

- Instruction set design from HLL constructs
  - Expressions, assignments => ALU instructions
  - Data abstraction => Addressing modes
  - Conditional & loop statements => Branch instructions
  - Procedure calls/returns => stack management

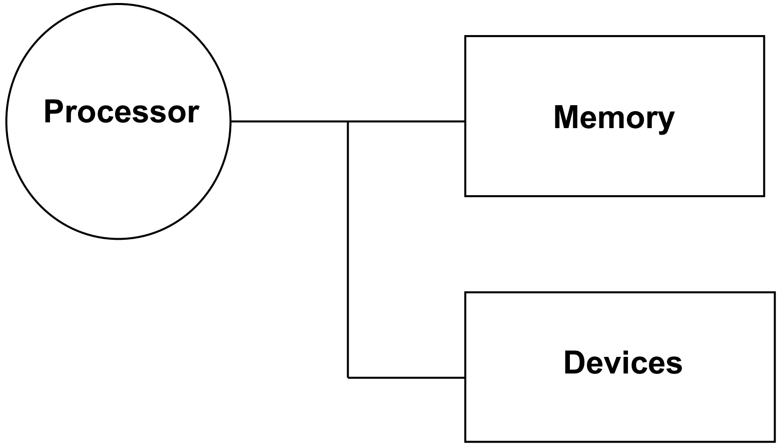
- Please note the reading assignments in the schedule: Start reading chapter 2

Simple Machine Model



- Remember the LC-3? It used a greatly simplified ARM instruction set
- We'll be introducing the LC-2200, a greatly simplified MIPS instruction set
  - Architecture that's similar, but not the same as the LC-3.

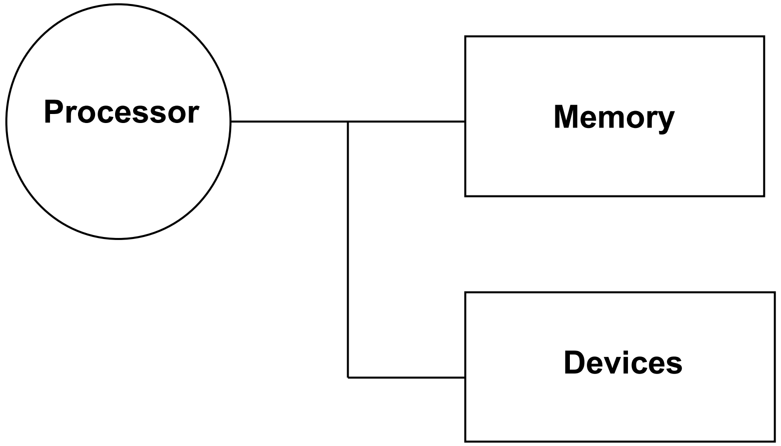
Simple Machine Model



- Let's consider the execution of a HLL

-
  ```
  a = a + 1
  c = a + b
  if (c == d) {
          …
  }
  ```

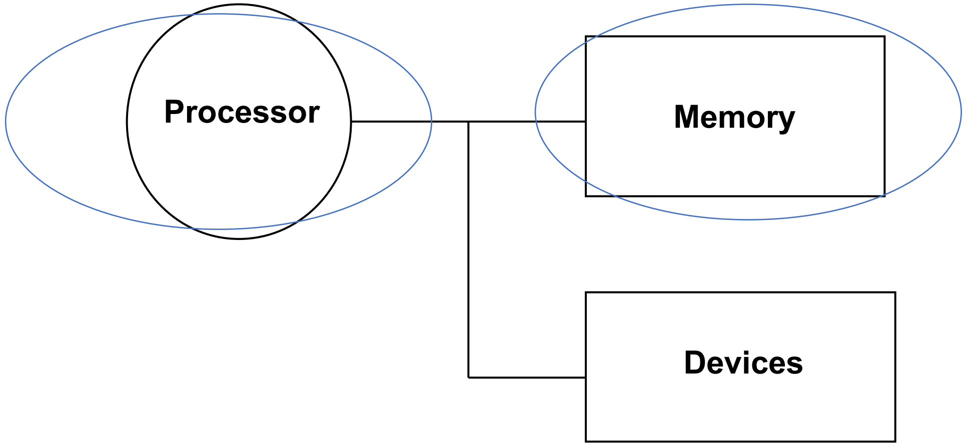
How to Design an Instruction Set?



Start thinking like a compiler writer

➔ What instructions are needed for each HLL construct?

Arithmetic/Logical Expressions



Start thinking like a compiler writer

➔ What instructions are needed for each HLL construct?

c = a + b

becomes

add c, a, b

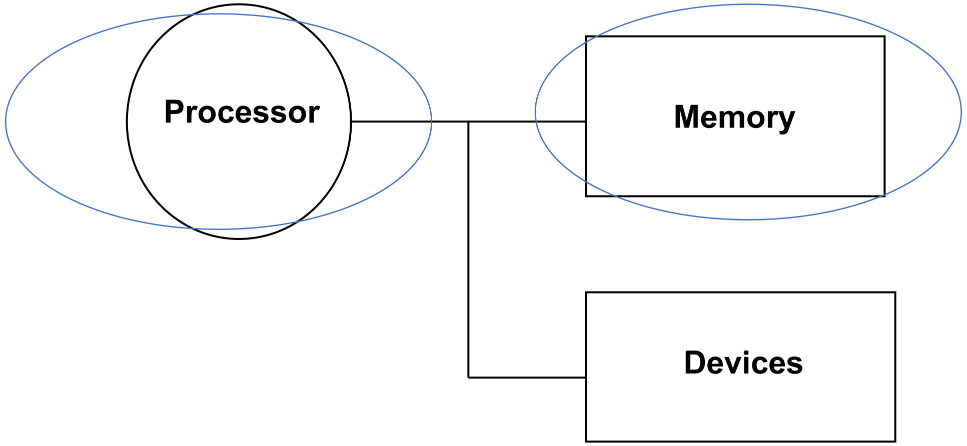
➔ memory operands

➔ memory addressing mode

What do you call these?

Where are they?

Arithmetic/Logical Expressions



Start thinking like a compiler writer

➔ What instructions are needed for each HLL construct?

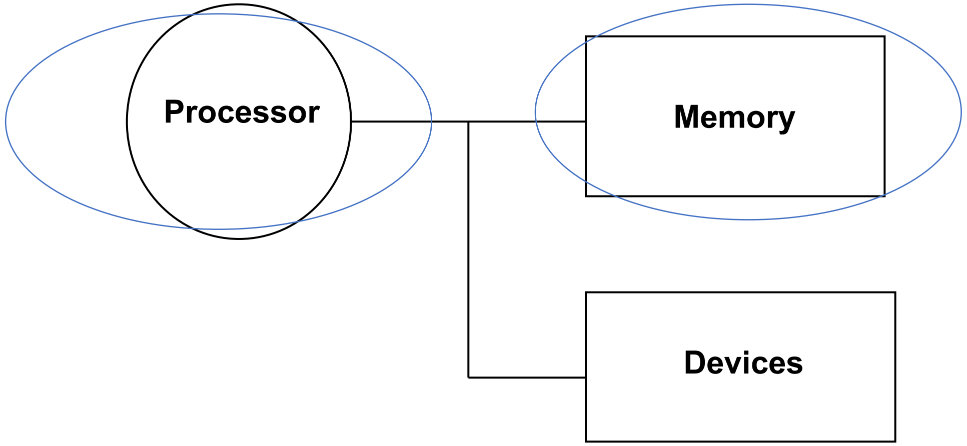
c = a + b ➔ add c, a, b

Keep adding to repertoire

c = a – b ➔ sub c, a, b

c = !(a & b) ➔ nand c, a, b

Arithmetic/Logical Expressions



Start thinking like a compiler writer

➔ What instructions are needed for each HLL construct?

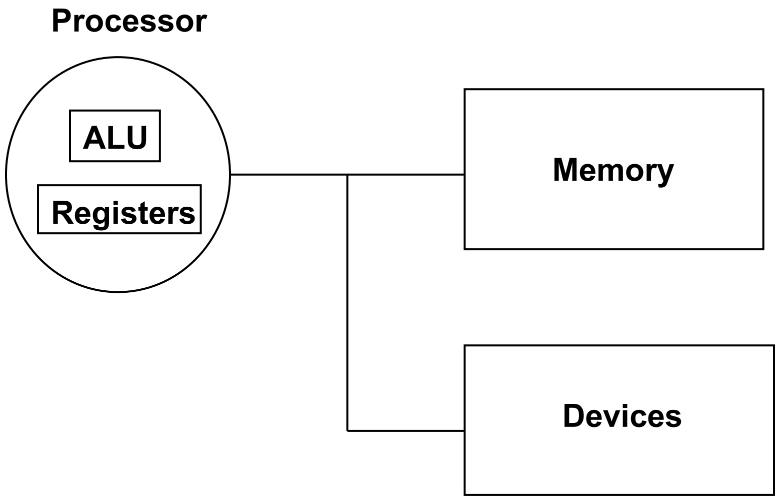
c = a + b ➔ add c, a, b

Is there a downside to operands in memory?

How can we address that?

A trip to memory is EXPENSIVE!

Operands?



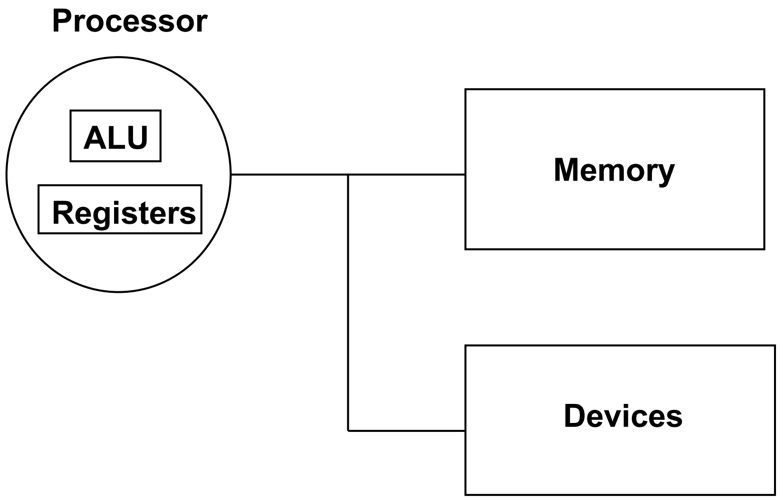
c = a + b ➔ add c, a, b

How about Load/Store instructions?

ld r_1, a

st c, r_2

Load/Store Instructions



ld r_1, a

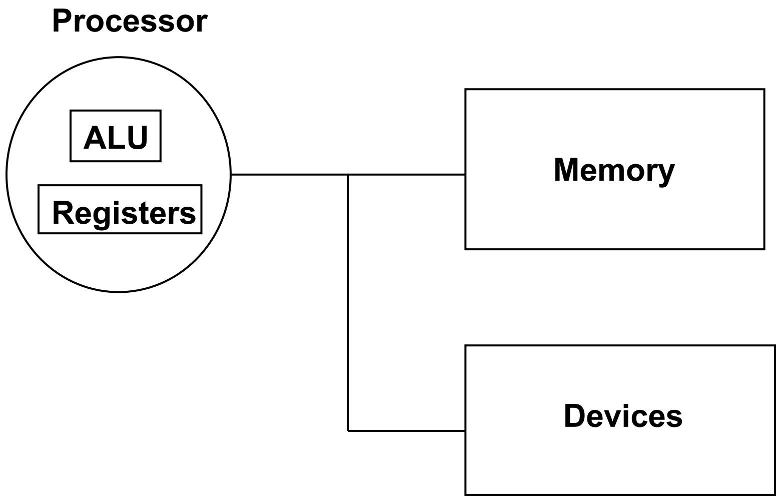
st c, r_2

➔ We've got operands in registers

➔ Register addressing mode!

So how do we compile c = a + b now?

Register Operands



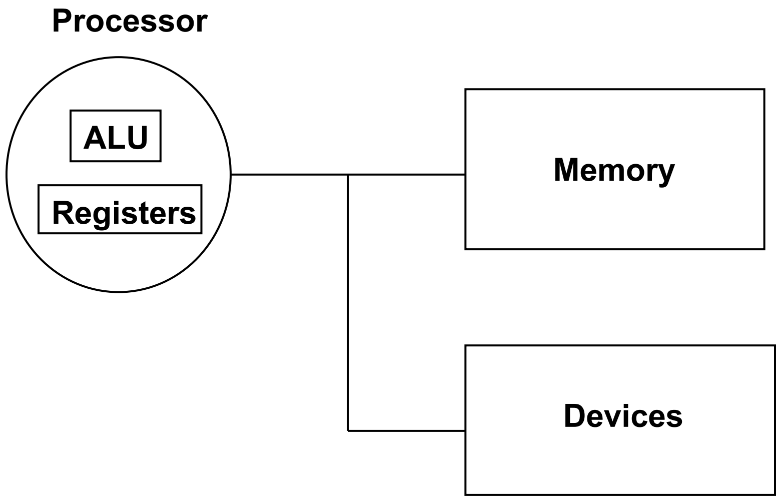
Old way:

```
    add  c, a, b
```

New way:

$$\begin{aligned}&\text{ld}\ \ r_1, a\\&\text{ld}\ \ r_2, b\\&\text{add}\ \ r_3, r_1, r_2\\&\text{st}\ \ c, r_3\end{aligned}$$

Compiling with Register Operands



Old way:

add c, a, b — 1 instruction

New way:

ld r_1, a
ld r_2, b
add r_3, r_1, r_2
st c, r_3

— 4 instructions

This looks dumb!

Not really.
Why?

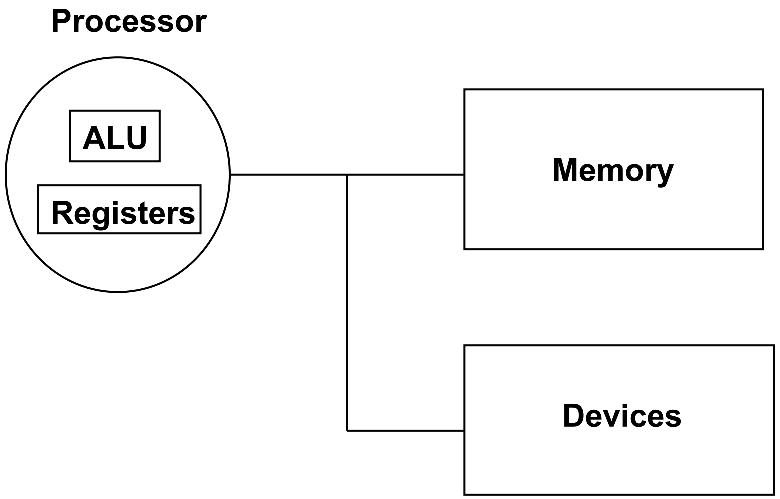
We can re-use the
values in registers!

Keep Frequently Used Tools Nearby!

...or, the principle of **locality**

http://search.coolclips.com/m/vector/vc017870/Businesswoman-of-many-trades/

Reusing Values



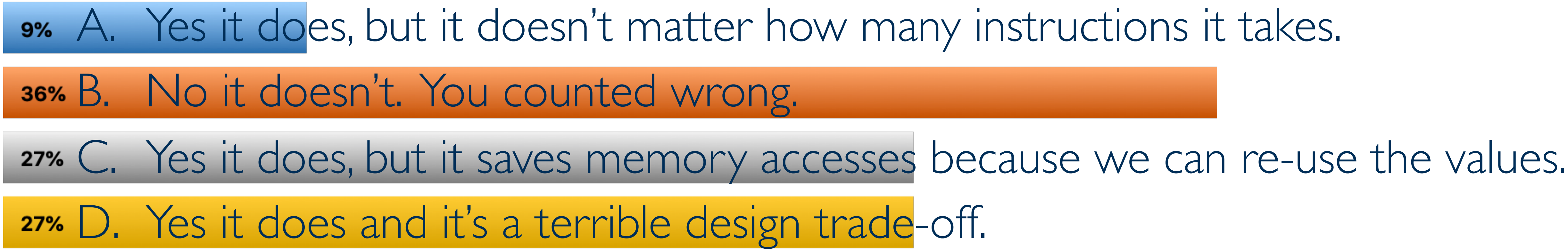
```
c = a + b
d = a * b + c
if (c == d) {
    ...
}
```

With operands left in registers, we can save three memory accesses here!

Session ID: dag2200

Why would we consider loading values into registers before computing with them?

Doesn’t that use more instructions?



9% A. Yes it does, but it doesn’t matter how many instructions it takes.

36% B. No it doesn’t. You counted wrong.

27% C. Yes it does, but it saves memory accesses because we can re-use the values.

27% D. Yes it does and it’s a terrible design trade-off.

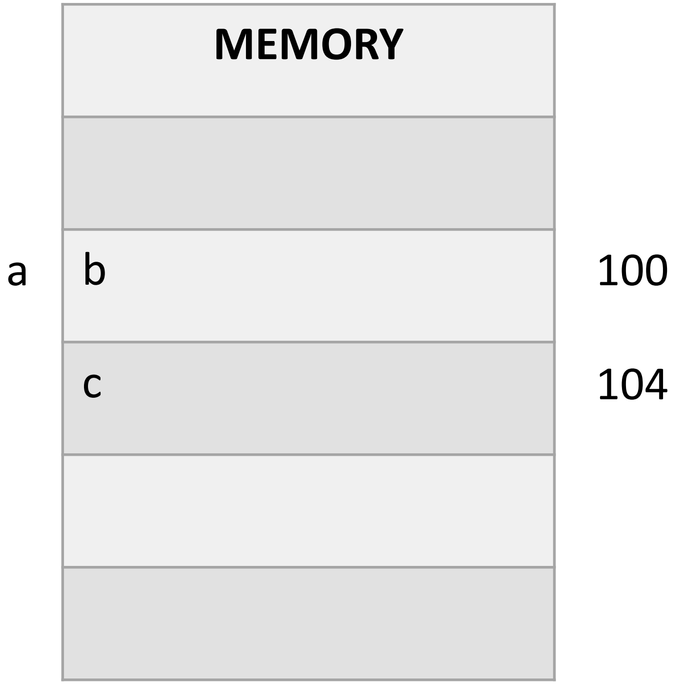
Structs in HLL

```
struct {
        int b;
        int c;
} a;
```

Elements of a struct are contiguous in memory

How do we load b and c into registers?

Let’s say &a is already in register R_1



MEMORY	
a b	100
c	104

Accessing Struct Members

```
struct {
        int b;
        int c;
} a;
```

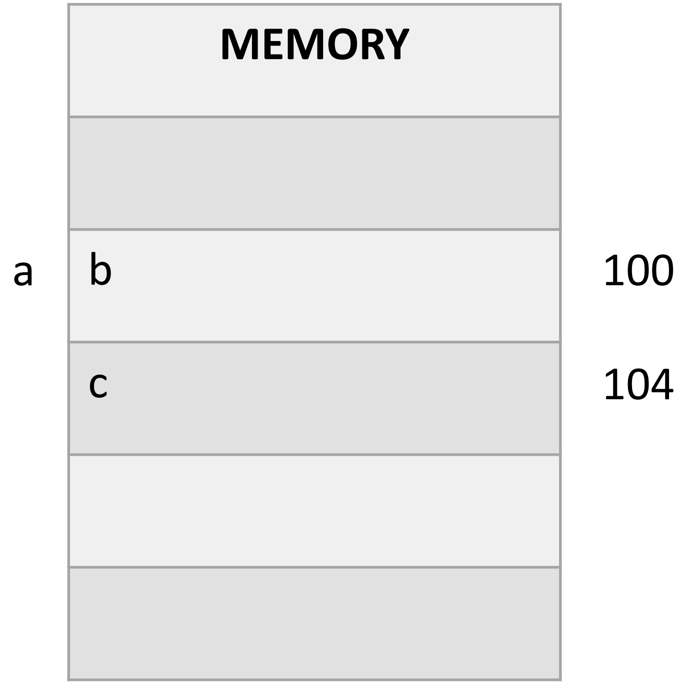
Let's say &a is already in register R_1

To load b:

$R_2 \leftarrow \text{memory}[R_1+0]$

To load c:

$R_3 \leftarrow \text{memory}[R_1+4]$



Base + Offset Address Mode

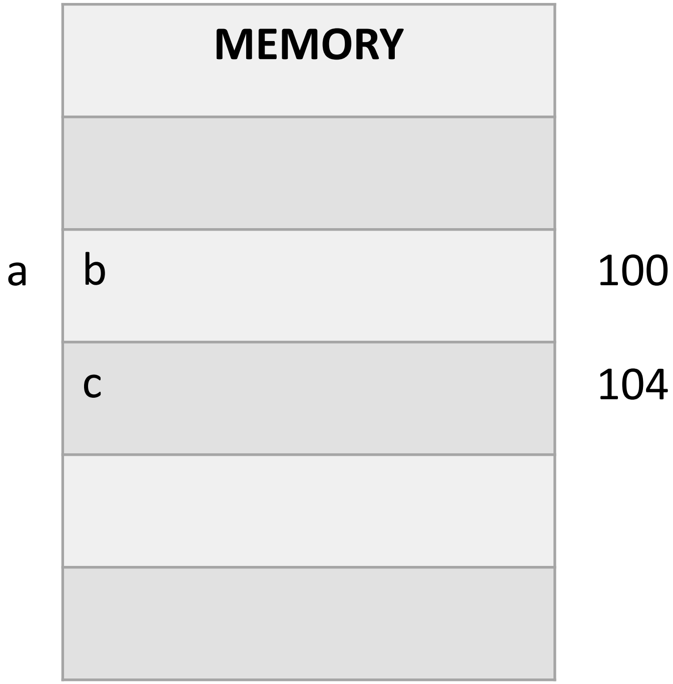
```
struct {
        int b;
        int c;
} a;
```

Let's say &a is already in register R_1

ld R_i, *offset*(R_{base})

To load b and c:

ld R_2, 0(R_1)
ld R_3, 4(R_1)



	MEMORY	
a	b	100
	c	104

Operand Granularity

char	→	8 bits	→	byte
short	→	16 bits	→	half word
int	→	32 bits*	→	word
long	→	64 bits*		

*depends on the word size of the architecture:
int=16, long=32 and others can happen

We need some instruction variants:

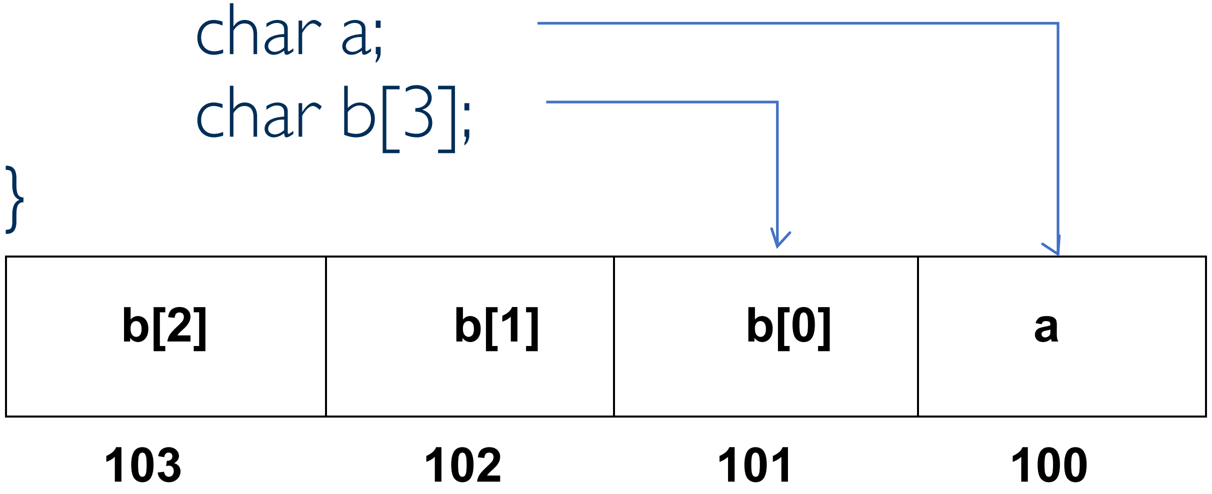
ldb, ldh, ldl, …
similar for store instructions

Operand Alignment

```
struct {
        char a;
        char b[3];
}
```

Dense Packing

```
struct {
        char a;
        char b[3];
}
```

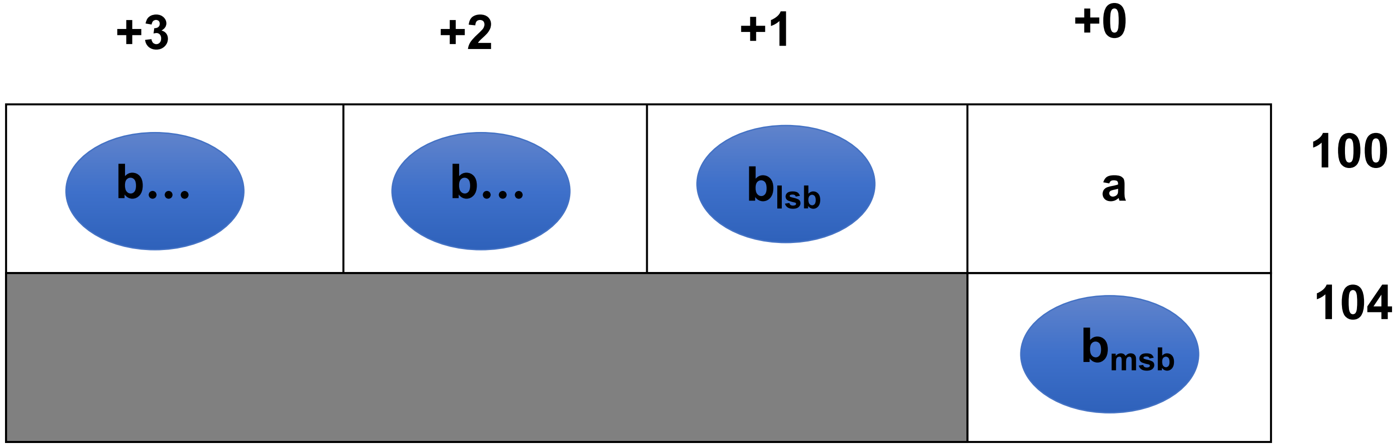


b[2]	b[1]	b[0]	a
103	102	101	100

- We're packing operands to save space.
- ISA may support ld/st with different precision!

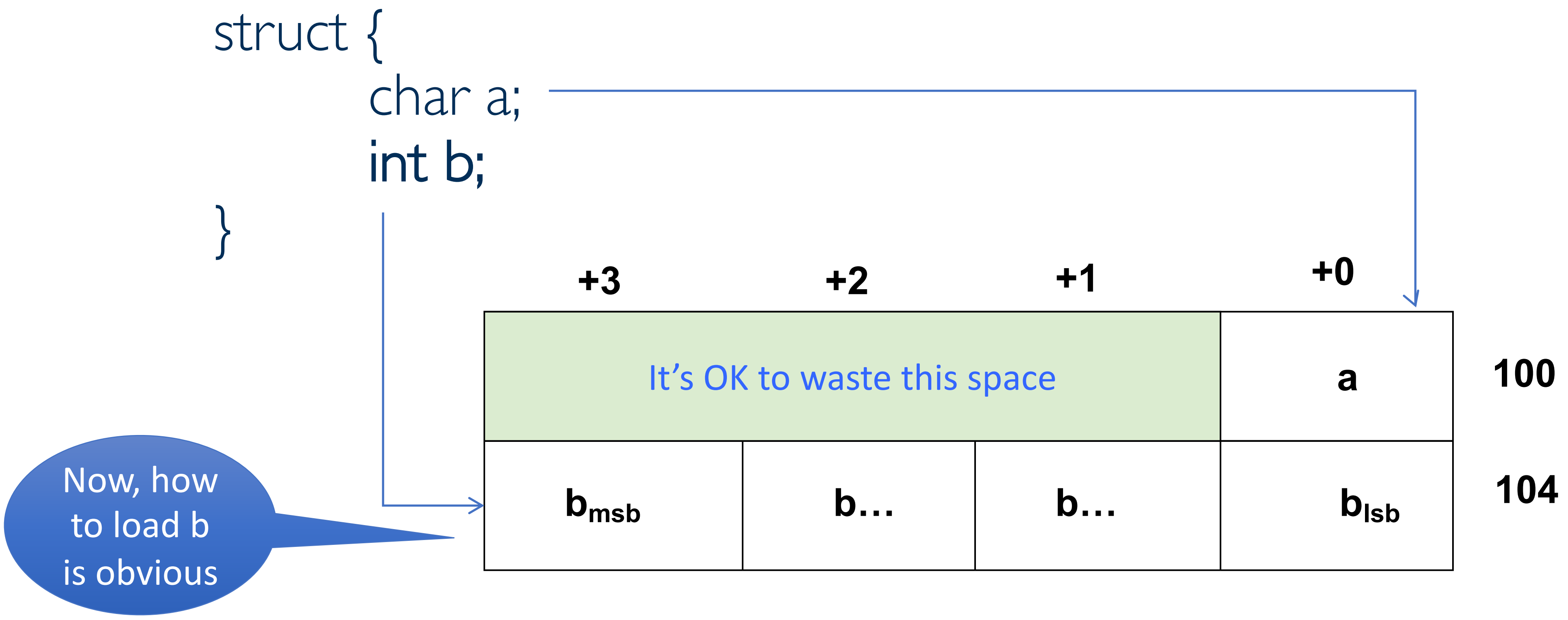
A Different Struct

```
struct {
        char a;
        int b;
}
```



If we have a 32-bit path to memory, how are we going to load b?

Why Alignment Rules Matter



```
struct {
        char a;
        int b;
}
```

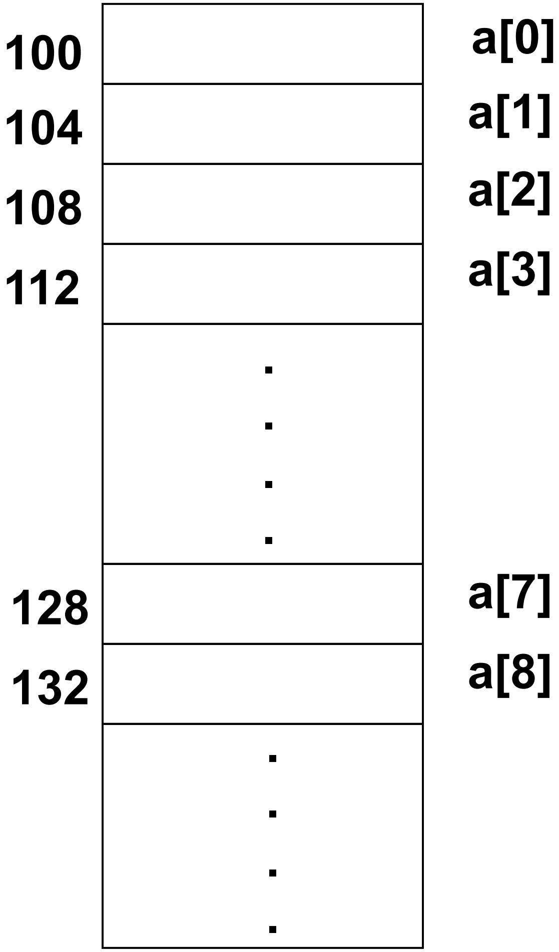
+3	+2	+1	+0	
It’s OK to waste this space			a	100
b_{msb}	b…	b…	b_{lsb}	104

Now, how to load b is obvious

This is one of many space/time tradeoffs that ISA designers must address.

Accessing Array Operands

int a[100];



One approach, use base+offset

$r_1 = 100$

To load a[8]

ld r_8, $32(r_1)$

Is this the best we can do?

Typical Array Use

- How do we typically use arrays?

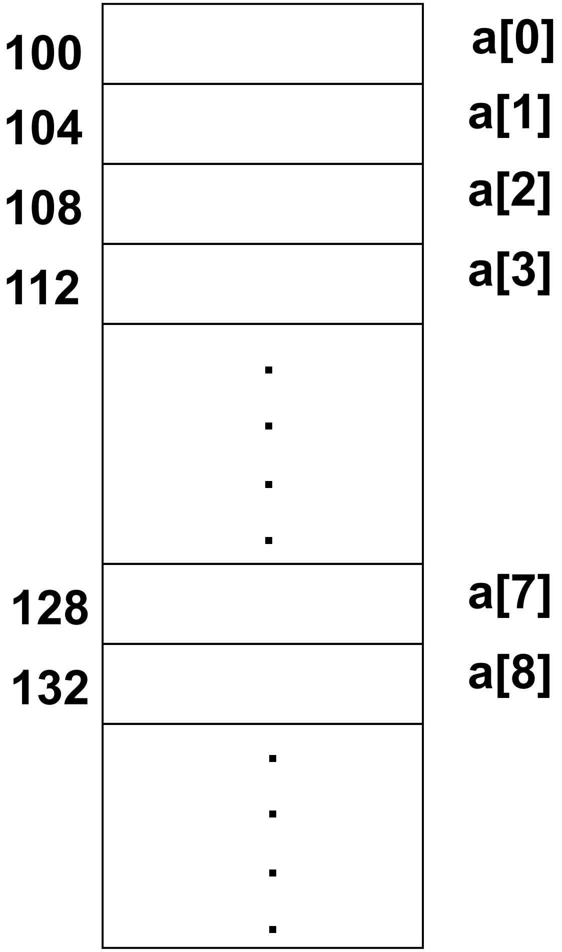
```
loop
 c[i] = a[i]
 ...
 i = i + 1
end loop
```

Very often, our subscript is a variable!

- Looks like it's time for a new addressing mode
- Let's implement base+index addressing

Accessing Array Operands

int a[100];



With base+index mode

$r_1 = 100$
$r_2 = i * 4$

Why?

To load a[8]

ld r_8, $r_2(r_1)$

Using base + index

Endianness

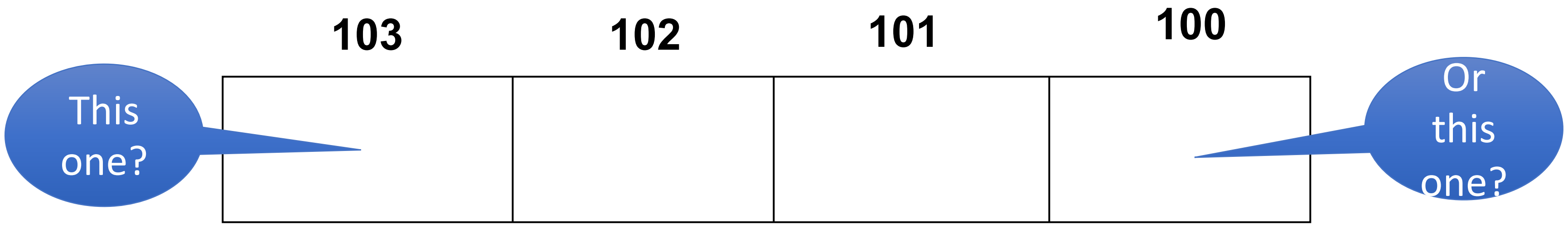
Say we have a 32-bit register, R1, that contains the value 1 in two's-complement.

We store that register into an (aligned) memory location

```
        ST    R1, Mem[100]
```

The value is stored in addresses 100-103 (of course).

Which byte contains the 1? (The other 3 will be zero, right?)



Endianness

Tip: Endianness often shows up in byte-addressable memories because we can access individual bytes out of longer data types

Little Endian ➔ addresses the LSB of the word

int b = 0x11223344;

+3	+2	+1	+0	
			a	100
44 b_{msb} 11	33 b… 22	22 b… 33	11 b_{lsb} 44	104

Big Endian ➔ addresses the MSB of the word

So What’s the Difference

The difference only shows up when taking a “word” apart (in this case, loading 8 bits from a 32-bit integer)

ldb r_1, Mem[104]

Big Endian ➔ loads 0x11 into r_1

Little Endian ➔ loads 0x44 into r_1

+3	+2	+1	+0	
			a	100
b… 11	b… 22	b… 33	b… 44	104

So What's the Difference

- The difference shows up when taking a word apart
- Let's store 0x11223344 (a 32-bit integer) at location 104

	Little Endian Result	Big Endian Result
ldb r_1, Mem[104]	0x44	0x11
ldh r_1, Mem[104]	0x3344	0x1122
ldw r_1, Mem[104]	0x11223344	0x11223344

	107	106	105	104
	b…	b…	b…	b…
Big Endian	44	33	22	11
Little Endian	11	22	33	44

So, about the LC-3

- Was it Big Endian or Little Endian?
- Think carefully…
- You can't tell!
- There are no instructions that manipulate more or fewer than 16 bits
- So there isn't any way to show this implementation detail!
- Was this accidental or on purpose?

Recap

Software	Hardware
Expressions & assignments	ALU instructions
Variable reuse	register addressing mode ld/st instructions
Data abstraction • struct • array	 base + offset addr mode base + index addr mode
Granularity of operands	ldb/ldh/ldw instructions addressability (byte, word)
Packing operands	Memory alignment (space/time tradeoff)
Endianness 0x11223344	Little (first byte is 0x44) / Big (first byte is 0x11)

Session ID: dag2200

Review Question 1

An instruction set…

36% A. Serves as a level of abstraction between software and hardware.

18% B. Provides the details of the machine implementation.

32% C. Deals with the datapath and control of the processor.

14% D. None of the above.

Session ID:
dag2200

Review Question 2

Addressing mode…

25% A. Refers to the kinds of opcodes supported in an architecture.

25% B. Refers to the way the operands are specified in an instruction.

20% C. Refers to the granularity of the memory element that can be addressed in an instruction.

10% D. Is a critical tool for USPS operations.

20% E. None of the above.

Review Question 3

Endianness of an architecture…

- 29% A. Is a key determinant of processor performance.
- 24% B. Is a key determinant of how the compiler lays out data structures in memory.
- 18% C. Matters if one declares a datatype of a particular granularity and accesses it at a different granularity.
- 6% D. Is the official name of the party held after completing this course.
- 24% E. None of the above.

What do we need for…

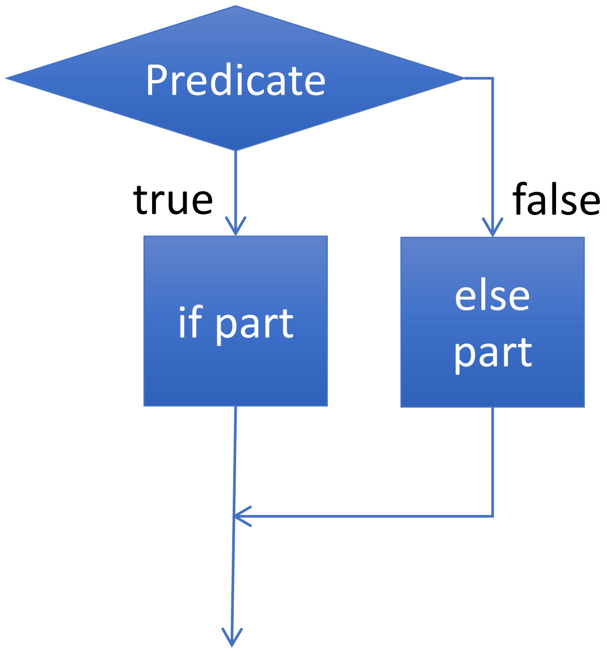
- Conditional statements
- Switch statements
- Loops
- Procedure calls
- Other considerations for ISA

Compiling Conditional Statements

- In what order are program statements normally executed?
- How do we know what instruction to execute next?
- How can we handle this high-level language construct:

```
    if(x == y) z = 7;
```

What Do We Need to Do?



- Evaluate predicate
- Break the sequential flow of instructions
- Rejoin control path

Implementing a Conditional

- Evaluate predicate
  - ALU Op
- Break sequential flow
  - Need to know where we are

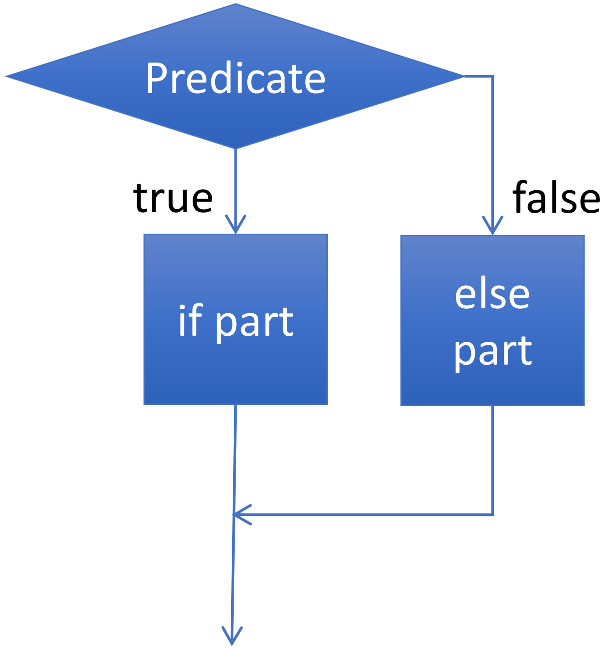
    → PC
  - Need a new instruction

    → BEQ r1, r2, offset

    → if r1 == r2 then PC = PC + offset else do nothing

    → PC relative addressing mode!
- Rejoin control flow

  → need an unconditional jump



An Example

- C

```
if(a == b)
  c = d + e;
else
  c = f + g;
```

- Assembly

```
     beq  r1, r2, then
     add r3, r6, r7
     beq r1, r1, skip*
then add r3, r4, r5
skip …
```

Assuming
r1 = a
r2 = b
r3 = c
r4 = d
r5 = e
r6 = f
r7 = g

* Effectively an unconditional branch

Outcome of Conditional Statements

- Introduction of PC
- One new instruction

  BEQ r_1, r_2, offset
- One new addressing mode: PC-relative
- (optional) an Unconditional Jump

  J r_n ; PC ← r_n
- Do we really need an unconditional jump??

Compiling Switch Statements

```
if (n==0)
        x=a;
else if (n==1)
        x=b;
else if (n==2)
        x=c;
else
        x=d;
```

```
switch (n) {
        case 0:
                x=a;
                break;
        case 1:
                x=b;
                break;
        case 2:
                x=c;
                break;
        default:
                x=d;
}
```

Do these produce essentially equivalent assembly code?

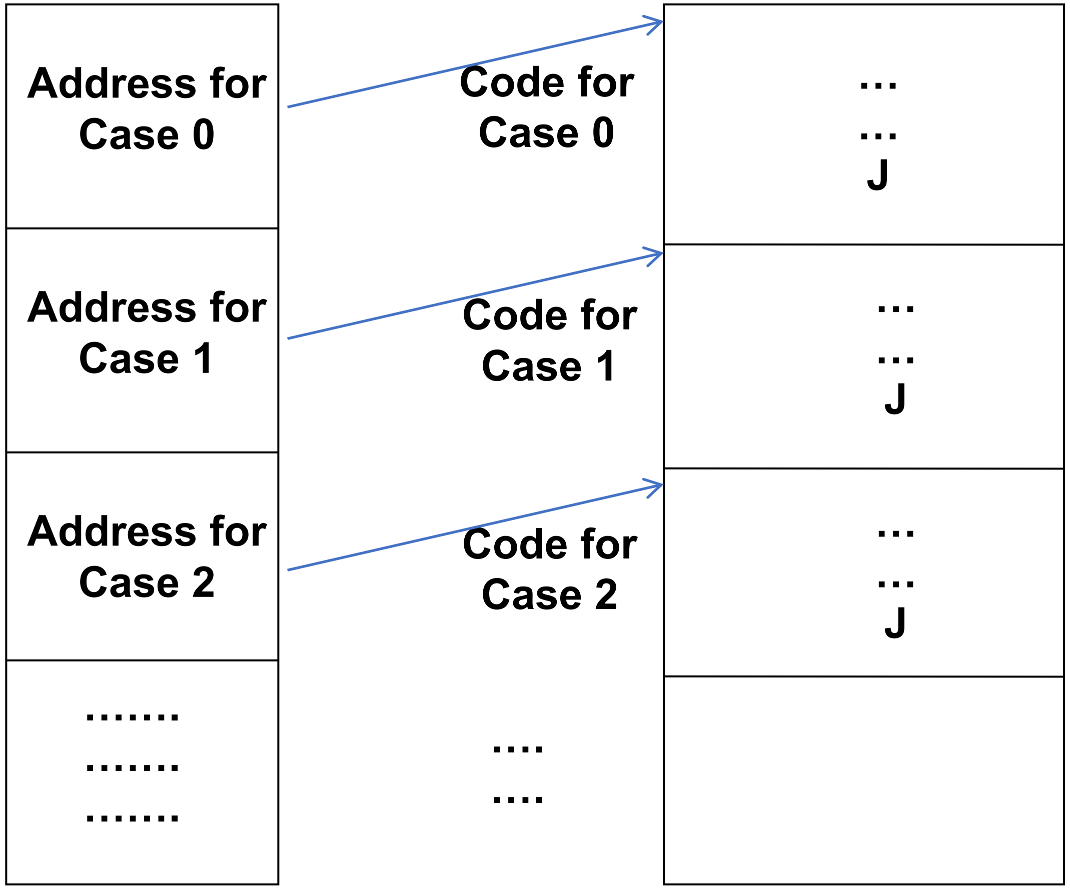
They can, but they don't have to!

Switch Can Use a Jump Table

- Think of a C array of pointers to the individual cases
- To do this we need an indirect addressing mode

  J @r_1

➔ PC ← Mem[r_1]



Jump table

Loops

- Do we need anything new in the ISA?
- Not really.

Compiling Loops

- C

```
while(j ! = 0)
{
    /* loop body */
    t = t + a[j--];
}
```

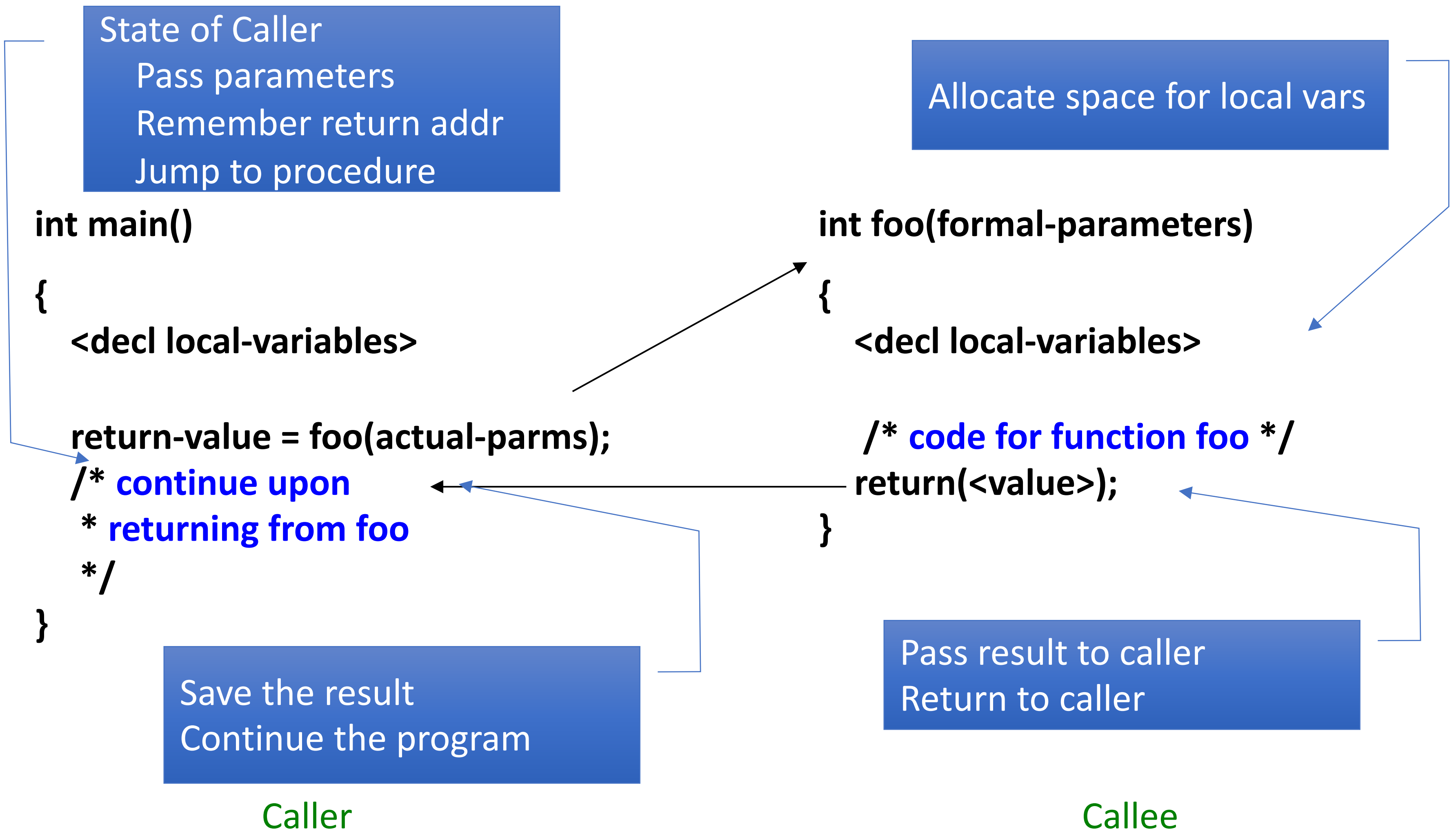
- Assembly

```
loop   beq r1,r0,done
       ; loop body
       …
       beq r0, r0, loop
done   …
```

Summary

Software	Hardware
Expressions & assignments	ALU instructions, LD/ST instructions
Data abstraction • struct • array	register addr mode base + offset addr mode base + index addr mode
Conditionals & Loops	PC-relative addr mode branch/jump instruction (register or PC-relative) Indirect addr mode (optional)

How Do We Compile Function Calls?



State of Caller
- Pass parameters
- Remember return addr
- Jump to procedure

```
int main()
{
   <decl local-variables>

   return-value = foo(actual-parms);
   /* continue upon
    * returning from foo
    */
}
```

Save the result
Continue the program

Caller

Allocate space for local vars

```
int foo(formal-parameters)
{
   <decl local-variables>

    /* code for function foo */
   return(<value>);
}
```

Pass result to caller
Return to caller

Callee

Remembering the Return Address

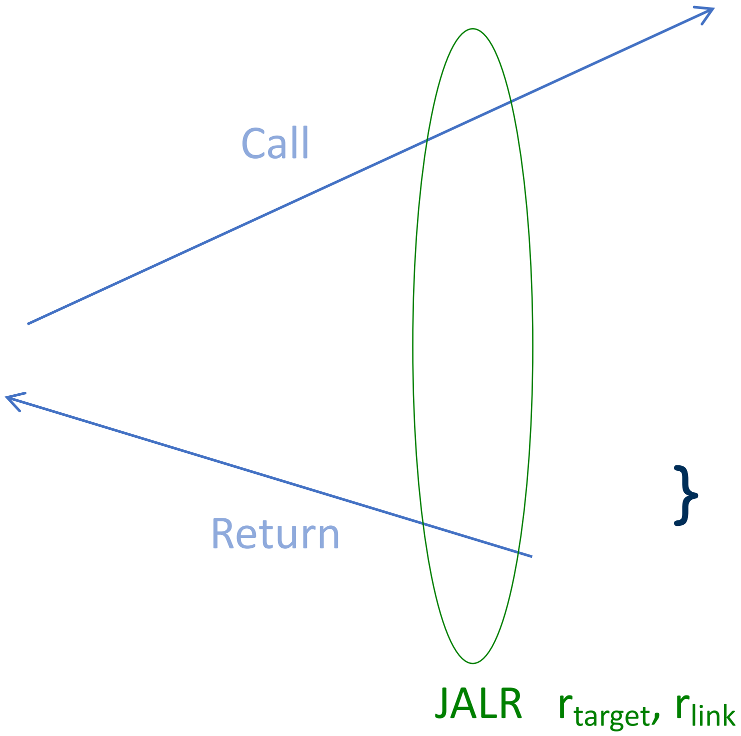
- Have we needed to do this before?

- Add a Jump & Link instruction
  - JALR $r_{target,}$ r_{link} ; r_{link} <= PC, PC <= r_{target}
- Recall

  J r_{target} ; PC <= r_{target}
- Do we need this instruction anymore?

Control Flow

```
main() {
.
.
    foo();
.
.
}
```



Call

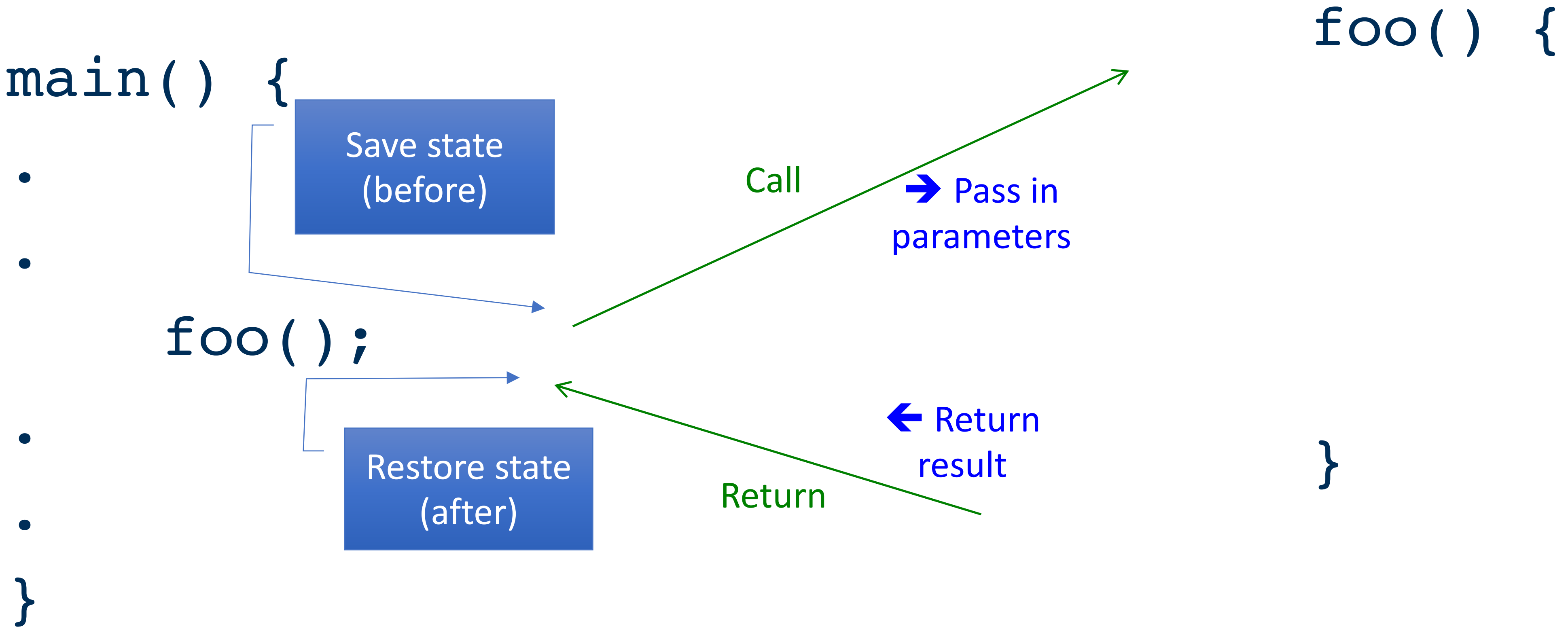
Return

JALR r_{target}, r_{link}

```
foo() {

}
```

Control Flow



```
main() {
.
.
    foo();
.
.
}
```

```
foo() {

}
```

Save state (before)

Restore state (after)

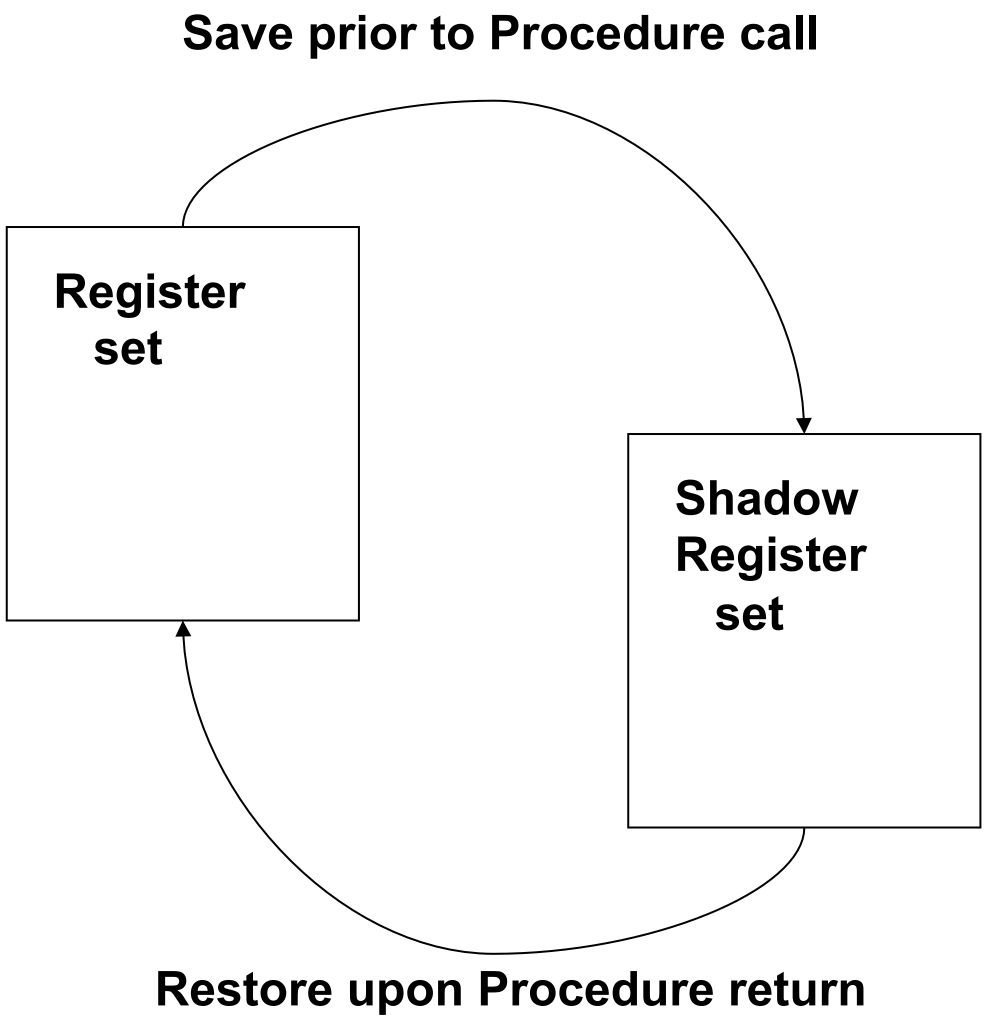
Call

➔ Pass in parameters

🡐 Return result

Return

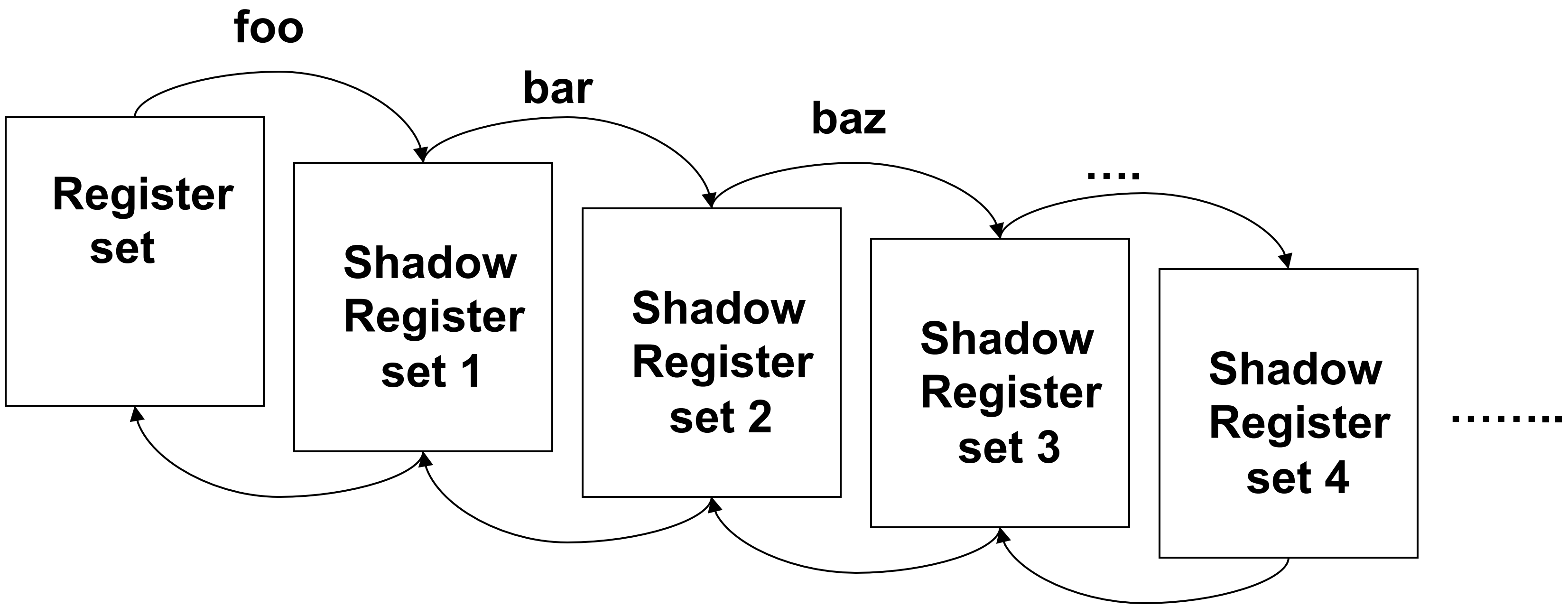
Another Way to Save State



Save prior to Procedure call

Restore upon Procedure return

Shadow Register Sets



- foo() calls bar() who calls baz(), etc.
- The Big Deal: No memory accesses!
  (but we need lots of extra registers)
- Another form of this is called **register renaming**

Saving State

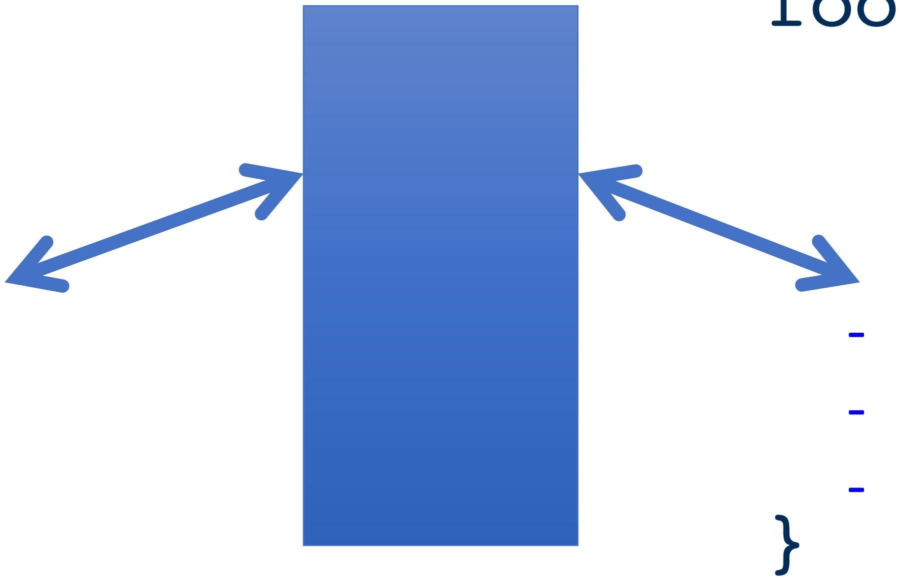
- If we don't have shadow registers, where are we going to save all that state?
- A stack

Where are we going to put the stack?

- In memory
- But in small cases, could we hold the state in a few extra registers? (another space/time tradeoff)

Use a Stack to Communicate

```
main() {
.
.
     foo();
.
.
}
```



```
foo() {
```

- Save/restore state
- Pass parameters
- Return results

```
}
```

- What else is needed in ISA?
- Nothing new..

Saving Registers During a Procedure

- We can have the **caller** save all the registers

  -or-

  We can have the **callee** save all the registers
- What's wrong with those choices?
  - Not everything needs to be saved every time…

Saving Registers During a Procedure

- If we split the assignment of the registers, then most of the time, the **caller** and **callee** can each save fewer registers based on what they actually need to use
- In the LC-2200 case, we'll functionally divide the working register set
  - **s0-s2** registers which the **callee** must preserve if it wants to use them
  - **t0-t2** registers which the **caller** must preserve if it wants their values to persist over a function call
- This division of responsibility saves memory accesses.

Stack as Communication Area

- Saving/restoring state over a procedure call

  Who does it?

  ➔ Split between Caller and Callee

Stack as Communication Area

- Returning results

  Do we really need to put them on the stack?

  ➔ Use registers
  (We'll call this register v0)

Stack as Communication Area

- Parameter Passing

  Do we really need to put them on the stack?

  ➔ Use registers
  (We'll call these registers **a0-a2**)

Stack as Communication Area

- Will we need the stack at all for parameters and results?
- What if we run out of registers?
- We use the stack if we run out
- Here we're trading time for complexity

Moral of the Story

- Use the stack sparingly
  - LD/ST instructions are expensive
    (i.e. memory access is slow)
- Software calling convention
  - Used by the compiler to keep track of the use of the stack and registers
  - Better have one!

Software Convention for LC-2200

Use:
Program
Data

- Registers s0-s2 are the caller's saved registers
- Registers t0-t2 are the temporary registers
- Registers a0-a2 are the parameter passing registers
- Register v0 is used for return value

Use:
Bookkeeping

- Register ra is used for return address (r_{link})
- Register at is used for target address (r_{target})
- Register sp is used as a stack pointer

Session ID: dag2200

Review Question 1

Saving and restoring of registers on a procedure call…

- 20% A. Is always done by the caller.
- 20% B. Is always done by the callee.
- 20% C. Is never done since hardware implicitly takes care of it.
- 20% D. Is done on a need basis partly by the caller and partly by the callee.
- 20% E. What is a caller/callee?

Session ID: dag2200

Review Question 2

On the LC-2200, how are actual parameters passed to a function?

20% A. On the stack.

20% B. On the heap.

20% C. Up to 3 in registers, the rest on the stack.

20% D. Up to 6 in registers, the rest on the stack.

20% E. None of the above.

Session ID: dag2200

Review Question 3

We store some values in registers during a procedure call...

25% A. Because we like to mix things up – variety is good!

25% B. Because it reduces memory references.

25% C. It makes the stack shorter so it reduces the danger of overflow.

25% D. It makes for prettier code.