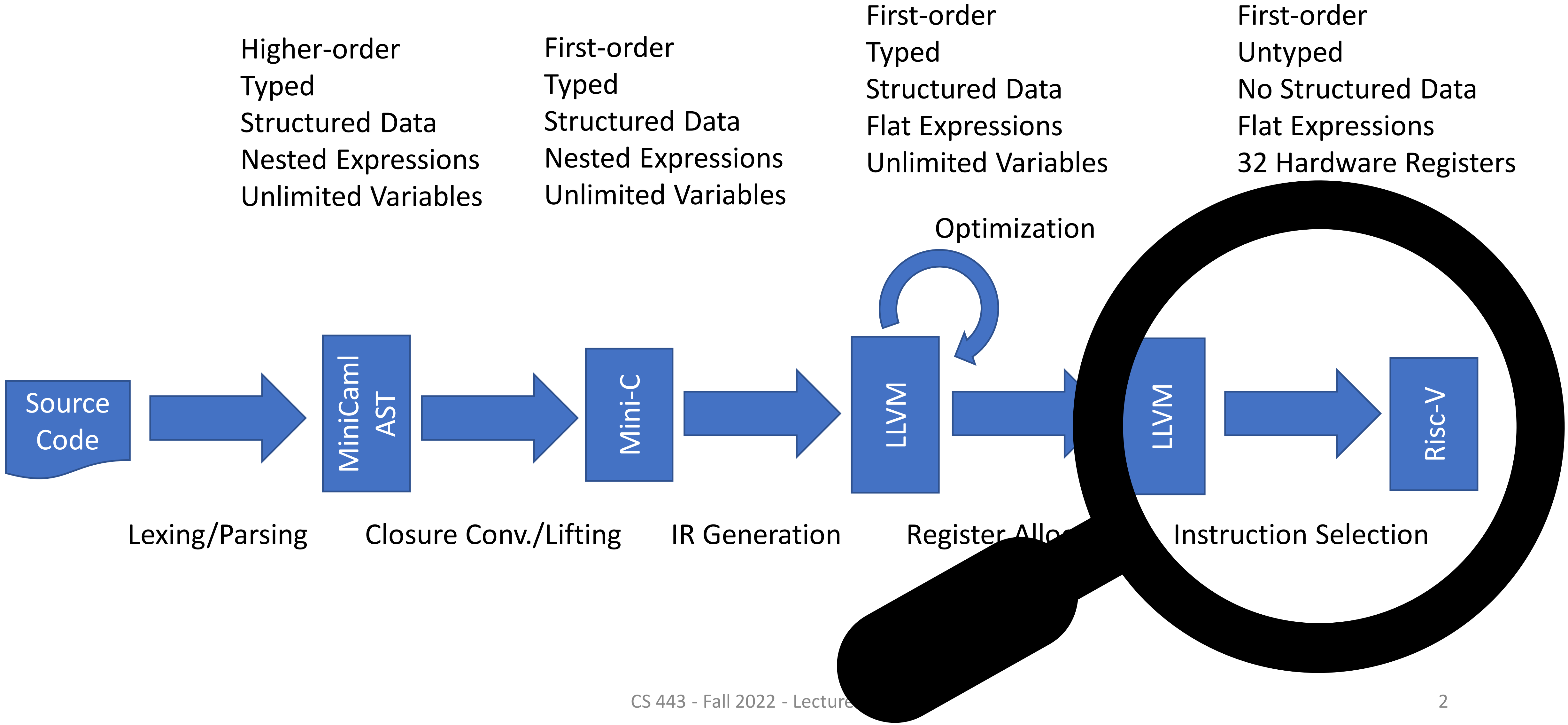
CS443: Compiler Construction

Lecture 21: Risc-V ISA

Stefan Muller

Based on material by Yan Garcia and Rujia Wang

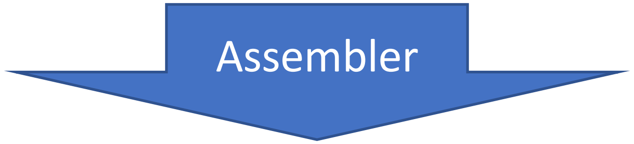
You are here


Higher-order
Typed
Structured Data
Nested Expressions
Unlimited Variables
First-order
Typed
Structured Data
Nested Expressions
Unlimited Variables
First-order
Typed
Structured Data
Flat Expressions
Unlimited Variables
First-order
Untyped
No Structured Data
Flat Expressions
32 Hardware Registers
Optimization
Source Code
MiniCaml AST
Mini-C
LLVM
LLVM
Risc-V
Lexing/Parsing
Closure Conv./Lifting
IR Generation
Register Allo
Instruction Selection

An ISA is the set of instructions a computer can execute

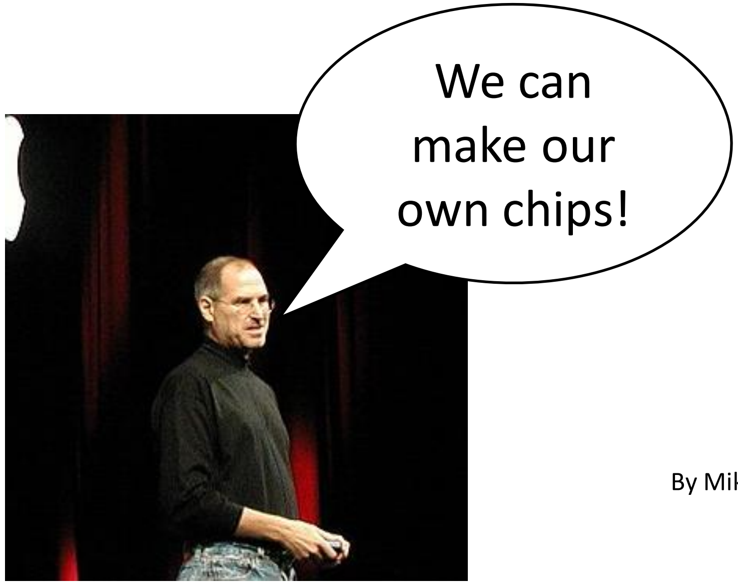
- The job of a CPU
  - Fetch an instruction from memory
  - Decode
  - Execute
  - Write results to memory
  - Repeat (basically) forever

```
add x3, x2, x0
```

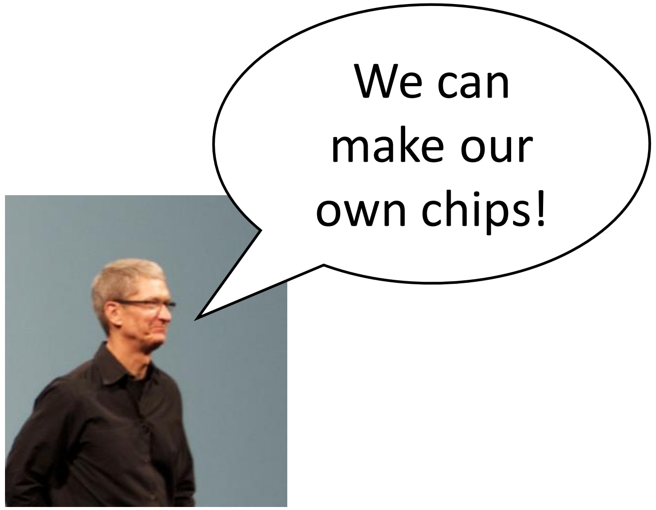


01110001110110

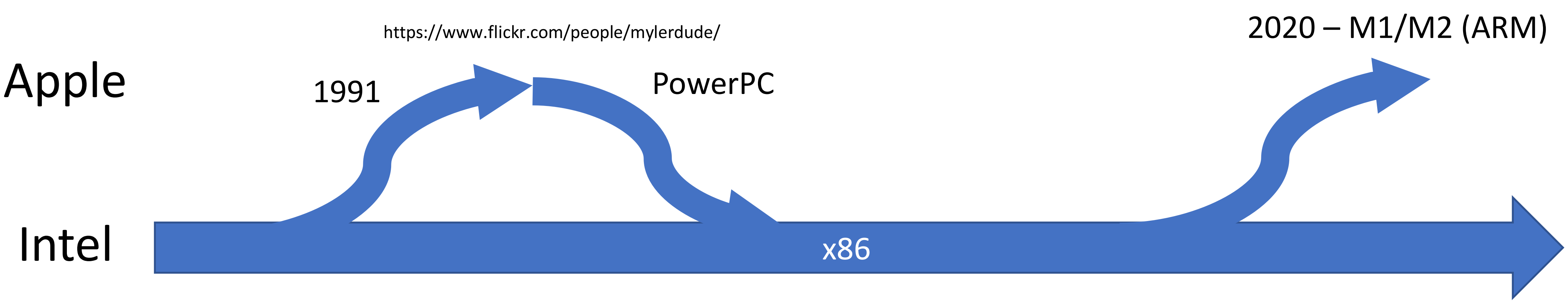
There are many different ISAs with rich histories



https://www.flickr.com/people/mylerdude/



By Mike Deerkoski - https://www.flickr.com/photos/deerkoski/7178643521/in/photostream



There are many different ISAs with rich histories

Intel

x86

x86-64

Hey, no stealing our architecture!

x86

AMD64

...

AMD

1990

2000

RISC (Reduced Instruction Set Computer) idea: simpler, faster hardware

- Earlier philosophy (“CISC”):
  Want to do something new? Add an instruction!

- RISC: Cocke, Hennessy, Patterson (1980s)

RISC-V: A simple RISC Architecture, good for teaching

- Originally developed in 2010 at UC Berkeley for teaching
- Open-source

Assembly Language: Human-readable machine code

- Assembly language is tied to ISA
- (Roughly) 1-to-1 correspondence with ISA instructions
  - (Some assembly languages offer convenient mnemonics that expand to multiple instructions)

An instruction is an opcode and operands (registers)

$$\underbrace{\texttt{add}}_{\text{opcode}}\ \underbrace{\texttt{x3}}_{\text{dest}},\ \underbrace{\texttt{x2, x0}}_{\text{operands}}$$

```
add rd, rs1, rs2
```

- Operands can only be **registers** and sometimes constants (“immediates”)
- Registers: Limited number of single-word storage locations in hardware

Registers in RISC-V

- (Also some floating point registers we won't talk about)

Register	ABI Name
x0	zero
x1	ra
x2	sp
x3	gp
x4	tp
x5-7	t0-2
x8	s0/fp
x9	s1
x10-11	a0-1
x12-17	a2-7
x18-27	s2-11
x28-31	t3-t6

Before we dive into RISC-V: A quick recap on data representation

- Bit (<u>b</u>inary dig<u>it</u>): 0 or 1
- “Nibble”: 4 bits (1 hex digit 0x0-0xF)
- Byte: 8 bits
  - 2 hex digits: 0x00-0xFF
- Word: “Natural” size of data operated on by a computer
  - 32-bit ISA: 32 bits (4 bytes)
  - Width of registers

Integers in binary/hex

1	0	1	0

$\times 2^3 \quad + \quad \times 2^2 \quad + \quad \times 2^1 \quad + \quad \times 2^0$

2	2	10
10	a	1010
16	10	10000
32	20	100000

"Most significant" "Least significant"

Review: Endianness

- Store data one byte at a time
  - Order of bits in a byte doesn’t change!

- So do we store the most significant byte at the lowest memory address (the way we’d write it left-to-right) or the highest?
  - Lowest: “Big-endian” (e.g., IBM System/360)
  - Highest: “Little-endian” (e.g., x86, RISC-V)

Little-endian

- 0xdeadbeef

ef	be	ad	de

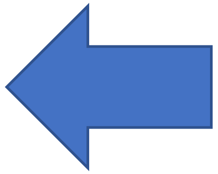
Two’s complement signed integers

- A 1 in MSB (Most significant bit) subtracts 2^{31} (instead of adding it)
- 100000…. = -2^{31}
- 011111…. = $2^{31}-1$ (highest positive # representable)
- 111111…. = -1

- Can just add two’s complement #s without casing on sign!

Two's complement means two ways to extend integers to the left

```
1010101
```



- If signed int: want to sign-extend (extend with MSB)
  - LLVM: sext
  - 101 as 3-bit int = -3 = 11101 as 5-bit int
- If unsigned: want to zero-extend (extend with 0s)

Assembly operands, registers are untyped

- Value is whatever we interpret it as – (signed/unsigned) int/char/bool, etc.

x1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
x2	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
x3	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

```
add x3, x2, x1
```

Overflow:

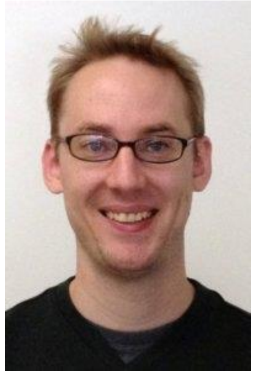
char: Yes. unsigned int: No. signed int: Yes.

Still want types? Never fear

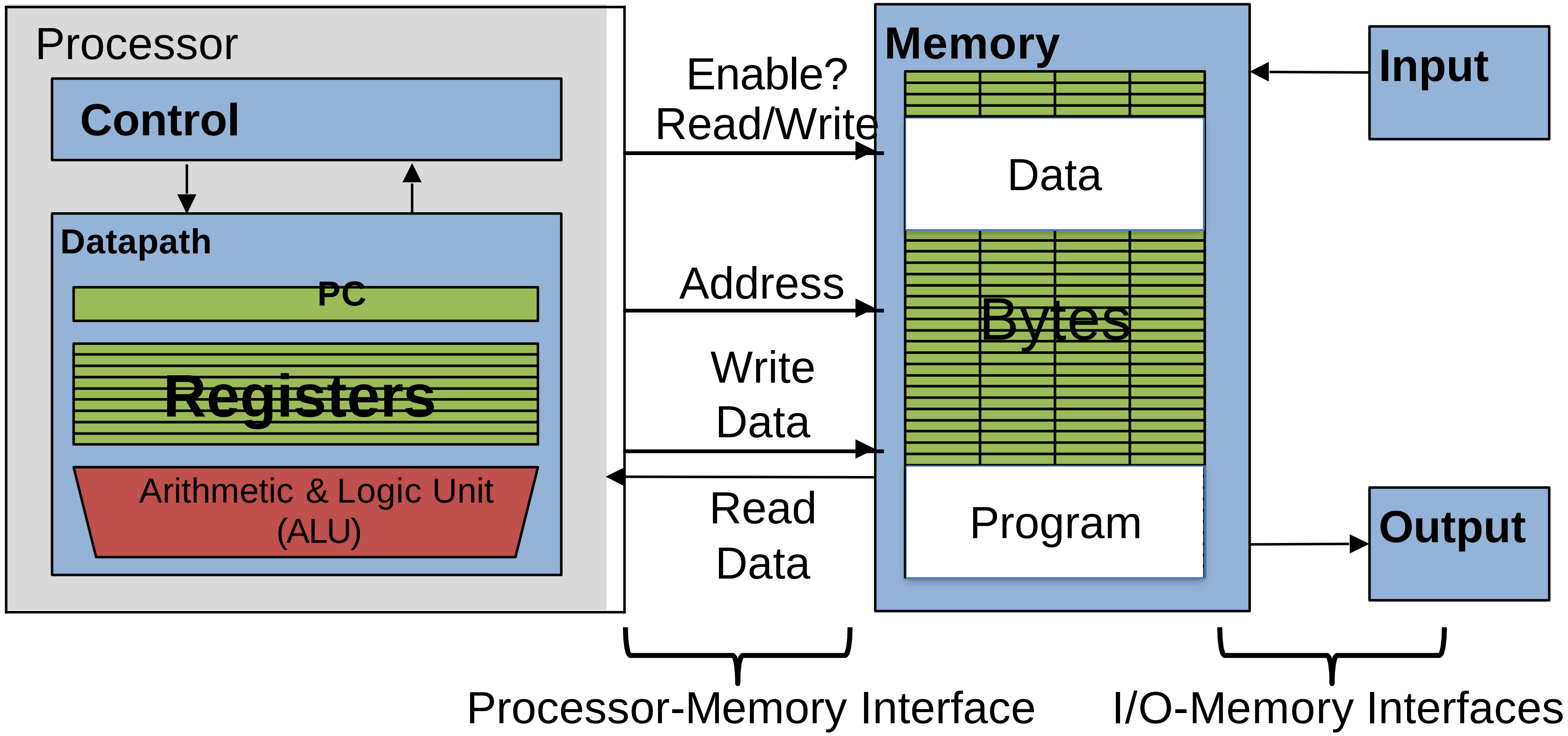
TALx86: A Realistic Typed Assembly Language*

Greg Morrisett Karl Crary† Neal Glew Dan Grossman Richard Samuels
Frederick Smith David Walker Stephanie Weirich Steve Zdancewic
Cornell University

1999



Registers are inside the processor



Q: Why not make a bigger processor with more registers?

RISC-V Instructions are 32 bits

- 6 types of instructions:

31	30 … 25	24 … 21	20	19 … 15	14 … 12	11 … 8	7	6 … 0	
funct7		rs2		rs1	funct3	rd		opcode	R-type
imm[11:0]				rs1	funct3	rd		opcode	I-type
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type

R-type instruction: Destination, two register operands

Risc-V	LLVM	C
add x1, x2, x3	%x1 = add i32 %x2 %x3	x1 = x2 + x3
sub x3, x4, x5	%x3 = sub i32 %x4 %x5	x3 = x4 – x5

Also: xor, or, and, mul, div
divu (div unsigned)
sll (shift left logical)
srl (shift right logical) – fill left with 0s
sra (shift right arithmetic) – fill left with sign bit
slt (set rd to 1 iff rs1 < rs2)

x0 is always 0, writes are ignored

- Why would you want to read from x0?
  - mv rd, rs = add rd rs x0

- Why would you want to write to x0?
  - nop = add x0 x0 x0
  - (There are other ways to write a no-op instruction, but this is the conventional one)

I-type instructions: Destination, register, immediate

Risc-V	LLVM	C
addi x1, x2, *n*	%x1 = add i32 %x2 *n*	x1 = x2 + *n*
sub x3, x4, *n*	%x3 = sub i32 %x4 *n*	x3 = x4 – *n*

Also: xori, ori, andi, (NO muli, divi)
slti
slli (shift left logical)
srli (shift right logical) – fill left with 0s
srai (shift right arithmetic) – fill left with sign bit

Example

x1 <- x   x2 <- y

```
%x = mul i32 %y 2
```

- `add x1, x2, x2`
- `slli x1, x2, 2`
- ```
  addi x1, x0, 2
  mul x1, x2, x1
  ```
  ```

Remember: You only get 12 bits for immediate (not very big)

- In RISC-V immediates are "sign extended"
  - So the upper bits are the same as the largest bit
  - Remember sign extended 2’s complement..
- So for a 12b immediate...
  - Bits 31:12 get the same value as Bit 11

31 30 25	24 21 20	19 15	14 12	11 8 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type

If you need big immediates, need 2 insts

Risc-V | C
lui x1, *n* | x1 = n << 12 (x1 = n * 4096)

```
%x = add i32 %y, 5000
```

x1 <- x   x2 <- y

```
4096 = 1 0011 1000 1000

lui x1, 1
addi x1, x1, 904
add x1, x1, x2
```

Control flow in LLVM: similar to LLVM, but less structured

Assembly:

```
loopforever:
   add x0, x0, x0
   j loopforever
```

After assembling/linkning:

```
add x0, x0, x0
j -4
```

Offset: *Position independent*

`j` isn't actually an instruction

- It's a "pseudoinstruction" that gets expanded into other instructions by the assembler (like mv, nop)
- We'll see more about this next week

B-type instructions (Conditional branches): 2 registers and a label/offset

Risc-V	LLVM	C
beq x1, x2, *addr*	%x3 = icmp eq i32 %x1 %x2 br i1 %x3, label *addr*, ???	if (x1 == x2) goto *addr*

Also: bne, blt, bge, (bltu, bgeu)

NO ble, bgt

Example

```
%x1 = icmp lt i32 %x2, %x3
br i1 %x1, label ltrue, label lfalse

blt x2, x3, ltrue
j lfalse

slt x1, x2, x3
bne x1, x0, ltrue
j lfalse
```

Unlike LLVM, control “falls through” to next instruction

Example

```
%x1 = icmp le i32 %x2, %x3
br i1 %x1, label ltrue, label lfalse

bge x3, x2, ltrue
j lfalse
```

Announcements

- Project 5 Deadline Extended to Monday (11/14)
- OH tomorrow, 2-3, NOT today
  - May be on Zoom, I'll let you know in the morning

- Schedule for rest of semester:
  - Mon, 11/14: Project 5 Due, Project 6 Out
  - 11/17, 11/22: Memory Management
  - Thur, 11/24: Thanksgiving, no class
  - 11/29, 12/1: TBA Lectures – suggest topics!
  - Fri, 12/2: Project 6 Due
  - **Tue 12/6, 10:30am, SB <u>113</u>** – Final exam

Example

Assuming assignments below, compile *if* block

f → `x10`   g → `x11`   h → `x12`

i → `x13`   j → `x14`

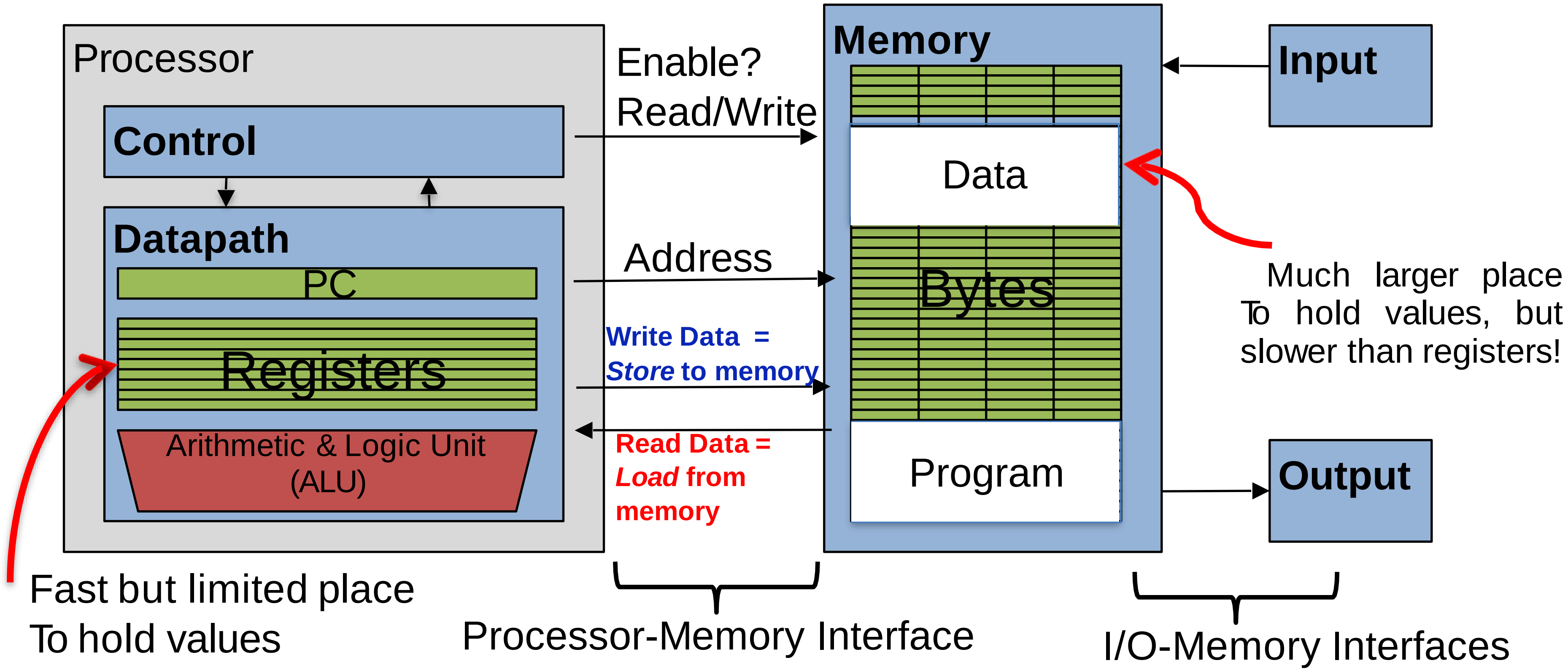
```
if (i == j)                  bne x13,x14,done
  f = g + h;                 add x10,x11,x12
                        done:
```

Unconditional jump instructions: jal, jalr

- `jal rd, imm`
  - Jump to label (or by offset)
  - Set rd = PC + 4 (next instruction after jal)
- `jalr rd, rs, imm`
  - Jump to address in rs + imm
  - Set rd = PC + 4 (next instruction after jal)

- `j imm = jal x0, imm`

Loading from and storing to memory



3
1

Memory is addressed in bytes

- (But access memory a word at a time, so in practice, will only access memory at multiples of 4 bytes)

- Generally: data >= 1 word must be *aligned* to addresses that are multiples of 4

lw loads from memory to register

`lw rd, imm(rs)`

Load word at rs + imm into rd

lw loads from memory to register

C code

```
int A[100];
g = h + A[3];
```

Register, register, immediate: lw is an I-type instruction

Using Load Word (`lw`) in RISC-V:

- `lw x10,12(x13) # Reg x10 gets A[3]`
- `add x11,x12,x10 # g = h + A[3]`

- Assume: x13– base register (pointer to A[0]) Note: 12– offset in bytes
- Offset must be a constant known at ***assembly time***

sw transfers from register to memory

C

```
int A[100]
A[10] = h + A[3]
```

RISC-V

```
lw x10, 12(x13)
add x10, x12, x10
sw x10, 40(x13)
```

Note:

- x13 – base register (pointer)
- 12, 40 – offsets in bytes
- x13 + 12 and x13 + 40 must be multiples of 4 to maintain alignment

Example

```
addi x11,x0,0xfeed
addi x12,x0,0xbeef
addi x6,x5,4
sw x11,0(x5)
sw x12,4(x5)
lw x12,0(x6)
```

- What’s the value in x12?

Answer: 0xbeef

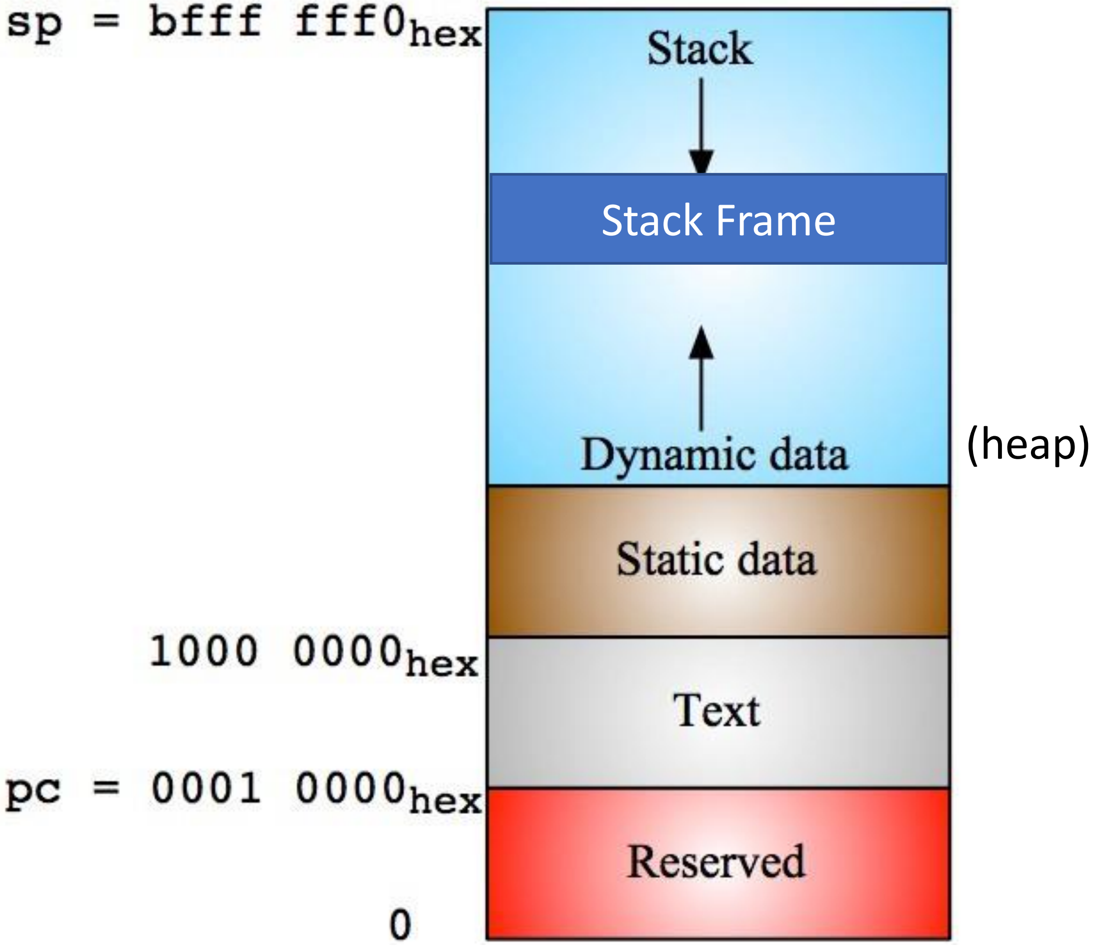
Example

```
addi x11,x0,0xfeed
addi x12,x0,0xbeef
addi x6,x5,1
sw x11,0(x5)
sw x12,4(x5)
lw x12,0(x6)
```

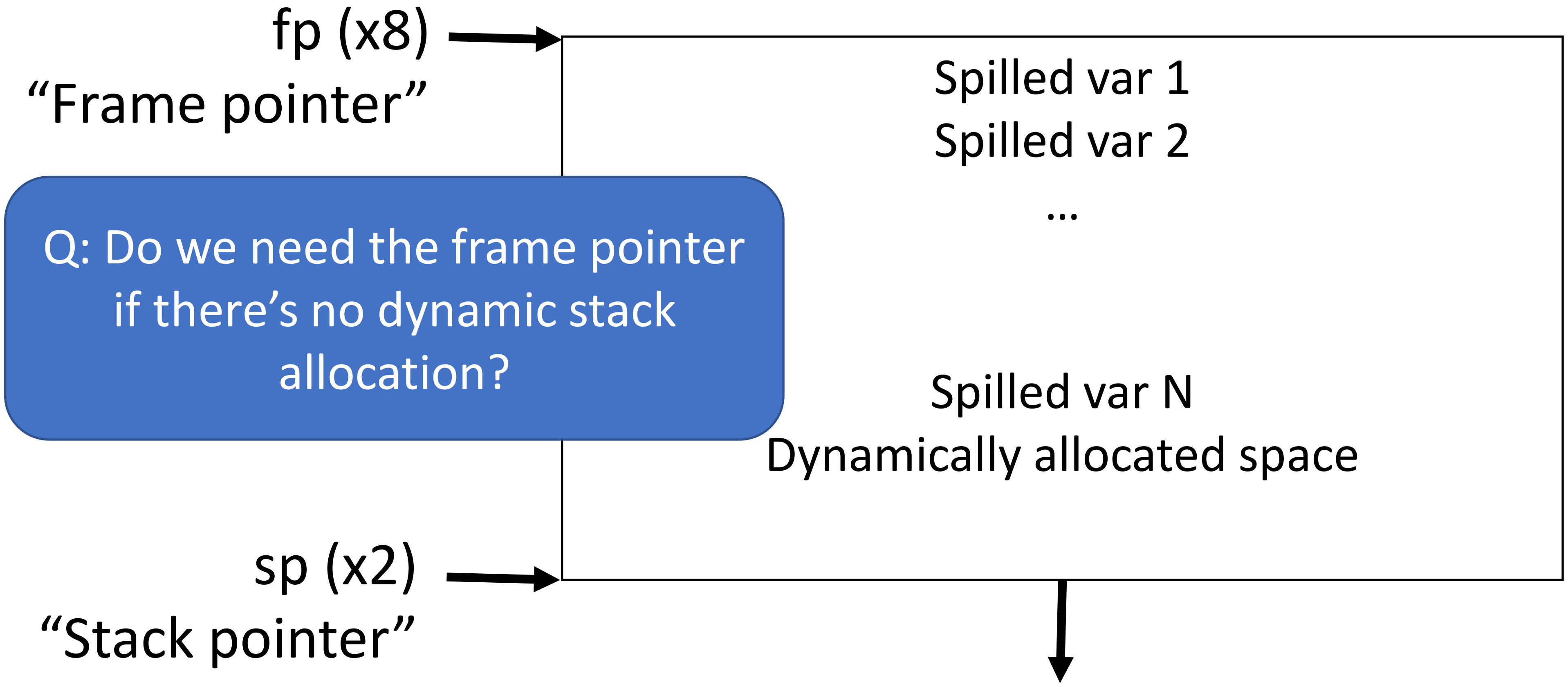
- What's the value in x12?

Answer: Undefined

Memory layout in RISC-V



A stack frame is where we store spilled locals, plus anything `alloca`'d



Q: Do we need the frame pointer if there's no dynamic stack allocation?