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

Instructors:
Sören Schwertfeger & Chundong Wang

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkley's CS61C

Last lecture

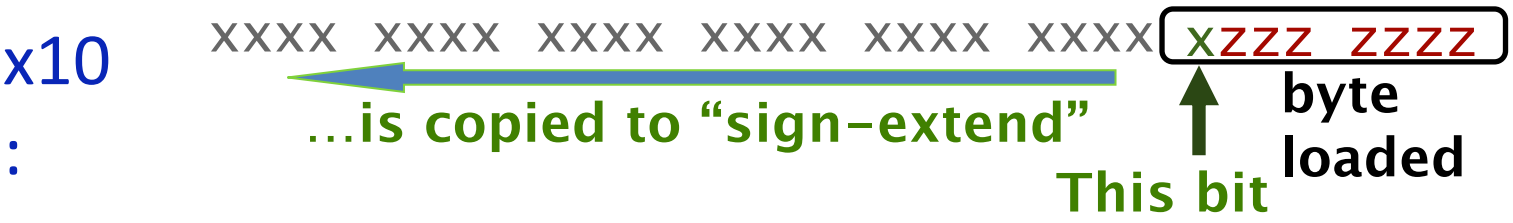
- In RISC-V Assembly Language:
  - Registers replace C variables
  - One instruction (simple operation) per line
  - Simpler is Better, Smaller is Faster
- In RV32, words are 32bit
- Instructions:
  `add, addi, sub, lw, sw, lb`
- Registers:
  - 32 registers, referred to as x0 – x31
  - Zero: x0

Loading and Storing Bytes

- In addition to word data transfers (`lw`, `sw`), RISC-V has byte data transfers:
  - load byte: `lb`
  - store byte: `sb`
- Same format as `lw`, `sw`

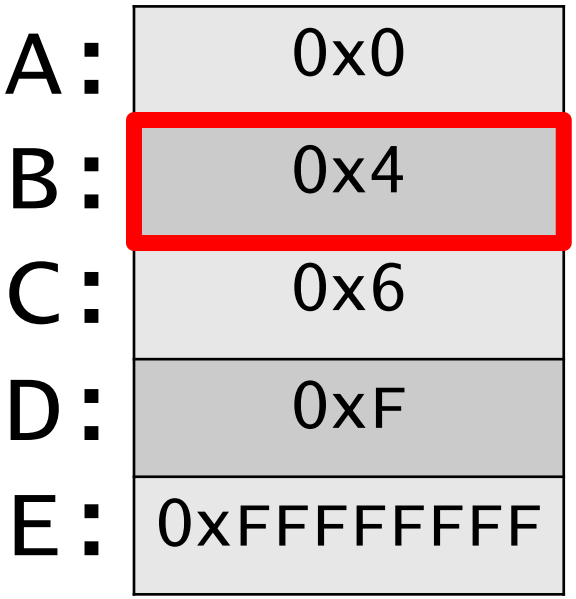
RISC-V also has "unsigned byte" loads (`lbu`) which zero extends to fill register. Why no unsigned store byte `sbu`?

- E.g., `lb x10,3(x11)`
  - contents of memory location with address = sum of "3" + contents of register x11 is copied to the low byte position of register x10.



Question! What's in x12?

```
addi  x11,x0,0x4F6
sw    x11,0(x5)
lb    x12,1(x5)
```



Question! What's in x12?

```
addi  x11,x0,0x85BCF6
sw    x11,0(x5)
lb    x12,2(x5)
```

Actually:
invalid instruction:
immediate is too large;
details covered later

A:	0x8
B:	0x85
C:	0xC
D:	0xBC
E:	0XFFFFFF85
F:	0XFFFFFFF8
G:	0XFFFFFFC
H:	0XFFFFFFBC

Question!
Which of the following is TRUE?

- A: `add x10, x11, 4(x12)` is valid in RV32
- B: can byte address 4GB of memory with an RV32 word
- C: imm must be multiple of 4 for lw x10, imm(x10) to be valid
- D: None of the above

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 $00FF_{hex}$ isolates the least significant byte
  - `andi` with FF00 0000_{hex} isolates the most significant byte
  - `andi` with 0000 0008_{hex} isolates the 4^{th} bit ($0000\ 1000_{two}$)

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: 0x7F0

C: 0x700

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*

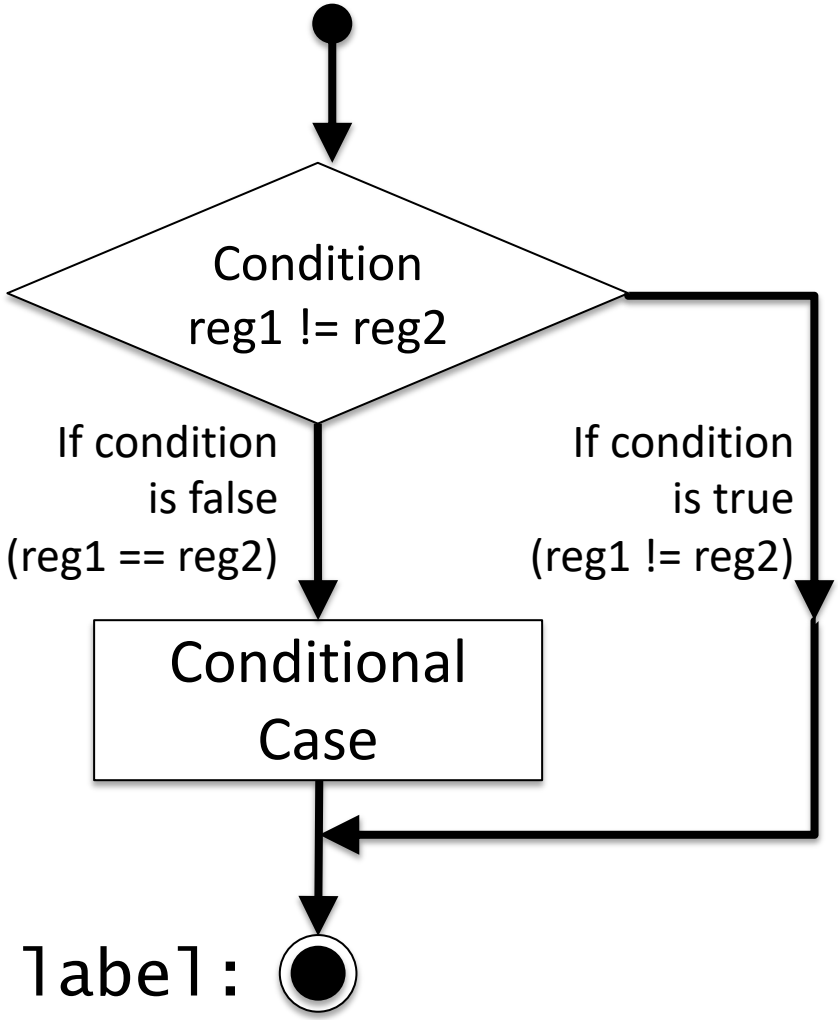
bne flowchart

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

```c
static int somedata = 10;

main(){
    int tmp = somedata;
    loop:  // label called "loop"
    tmp = tmp + 1;
    goto loop;
}
```

```
.data           # Assembler directive
                # static data

somedata:       # Label to some data "somedata"
    .word       0xA # initializa the word (32bit) with 10

.text           # code (instructions) follow here

main:           # label to first instruction of "main function"

    la x6, somedata # address of "somedata" in x6
    lw x5, 0(x6)    # (initial) value of "somedata" to x5

loop:           # label to the next instruction:
                # some jump goal in function (name "loop")

    addi, x5, x5, 1 # x5 += 1  (label loop points here)
    j loop          # jump to loop
```

Example *if* Statement

- Assuming translations below, compile *if* block

  f → x10 g → x11 h → x12

  i → x13 j → 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 → x10 g → x11 h → x12

  i → x13 j → 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;
  ```

Magnitude Compares in RISC-V

- "Branch on Greater or Equal "

  Syntax: `bge reg1, reg2, label`

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

- "Branch on Greater or Equal Unsigned"

  Syntax: `bgeu reg1, reg2, label`

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

- Conditional Branch instructions:
  - beq, bne: Branch if equal/ Branch if not equal
  - blt, bltu: Branch on less than/ unsigned
  - bge, bgeu: Branch on greater or equal/ unsigned

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

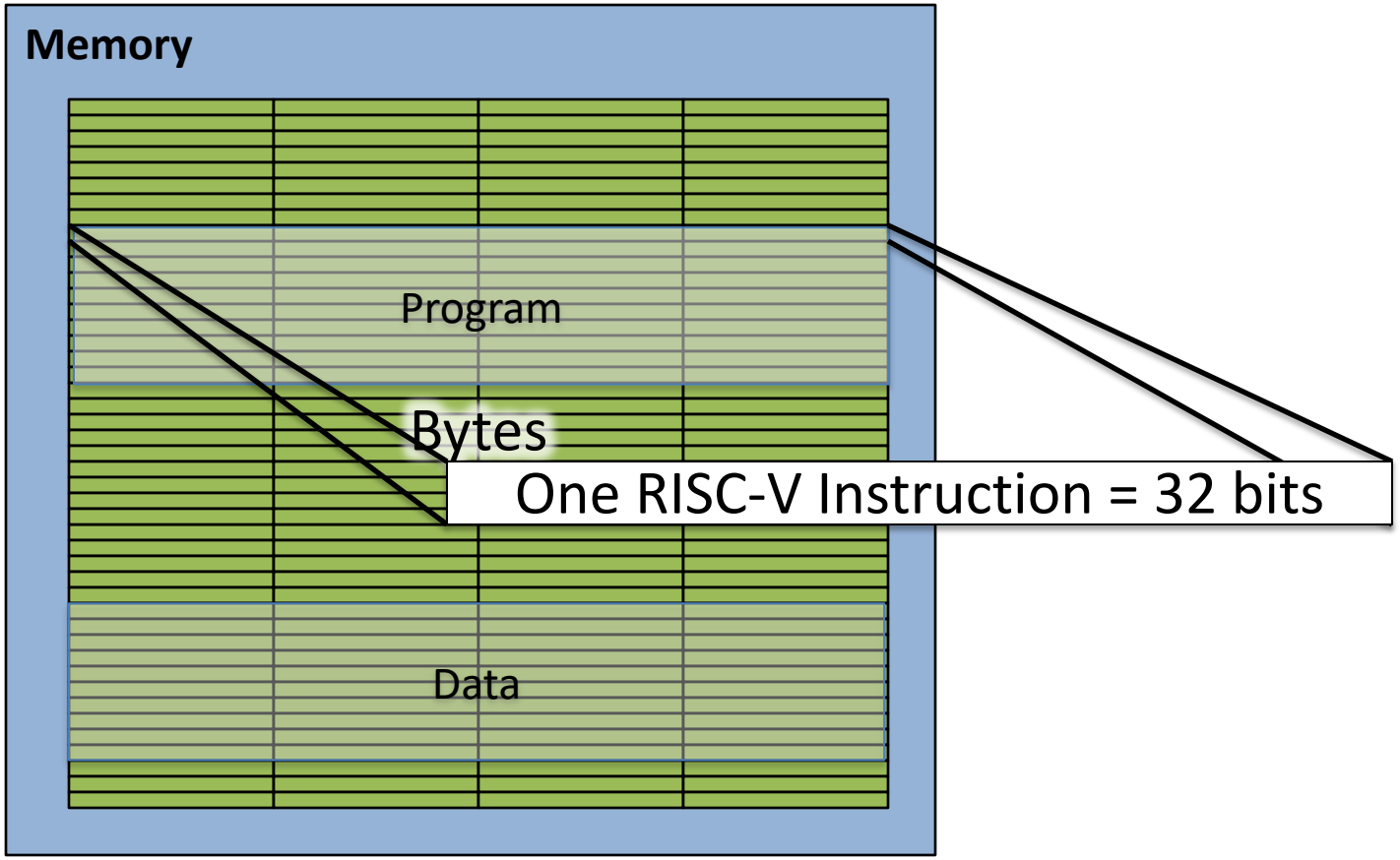
```
# Assume x8 holds pointer to A
# Assign x10=sum
add  x10, x0, x0 # sum=0
add  x11, x8, x0 # ptr = A
addi x12,x11, 80 # end = A + 80
Loop:
    lw   x13,0(x11)   # x13 = *ptr
    add  x10,x10, x13 # sum += x13
    addi x11,x11, 4   # ptr++
blt x11, x12, Loop:  # ptr < end
```

Premature Optimization...

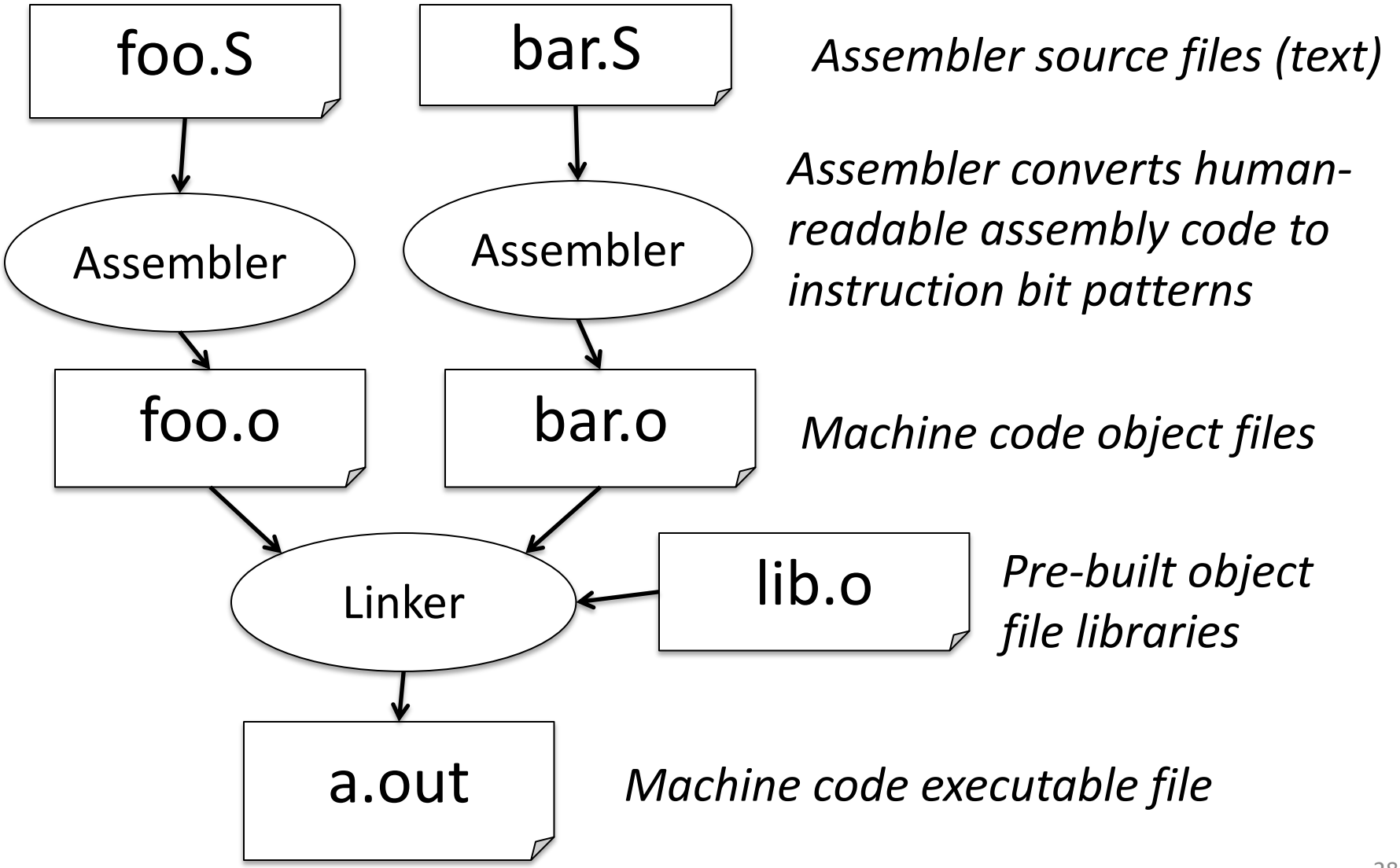
- In general we want ***correct*** translations of C to RISC-V
- It is ***not*** necessary to optimize
  - Just translate each C statement on its own
- Why?
  - Correctness first, performance second
    - Getting the wrong answer fast is not what we want from you...
  - We're going to need to read your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

PROCEDURES IN RISC-V

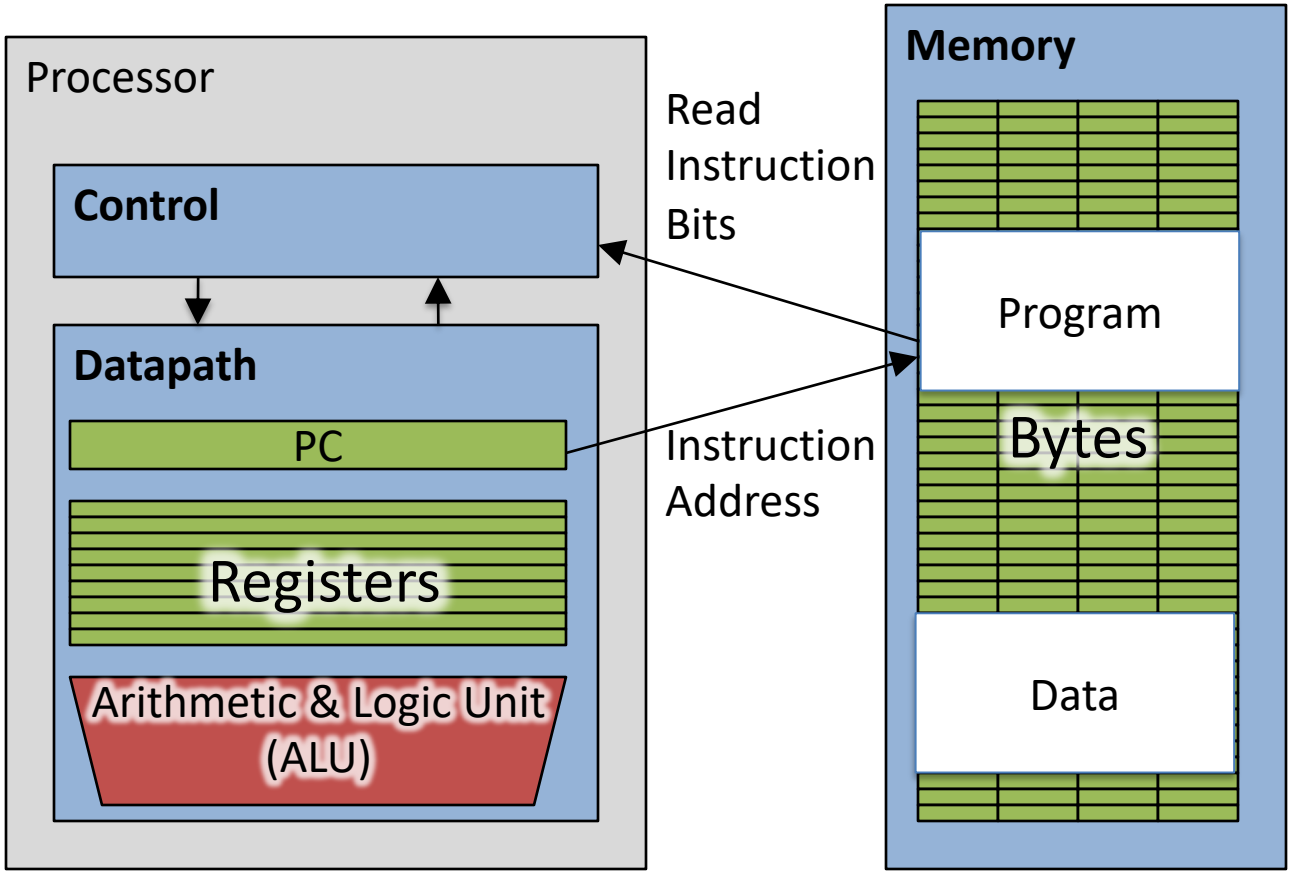
How Program is Stored



Assembler to Machine Code (more later in course)



Executing a Program



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

C Functions

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

/* 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 information must compiler/programmer keep track of?

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)

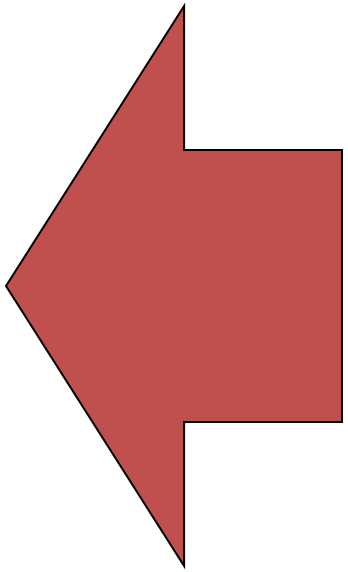
Instruction Support for Functions (1/4)

C

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

RISC-V

```
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)

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 (3/4)

C

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

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

RISC-V

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

Instruction Support for Functions (4/4)

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

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

- After:

```
1008 jal sum  # ra=1012, goto sum
```

- Why have a `jal`?
  - Make the common case fast: function calls very common.
  - Reduce program size
  - Don't have to know where code is in memory with `jal`!