Logical Instructions

RV32 So Far…

- Add/sub
```
add rd, rs1, rs2
sub rd, rs1, rs2
```
- Add immediate
```
addi rd, rs1, imm
```
- Load/store
```
lw  rd,  rs1, imm
lb  rd,  rs1, imm
lbu rd,  rs1, imm
sw  rs1, rs2, imm
sb  rs1, rs2, imm
```
- Branching
```
beq  rs1, rs2, Label
bne  rs1, rs2, Label
bge  rs1, rs2, Label
blt  rs1, rs2, Label
bgeu rs1, rs2, Label
bltu rs1, rs2, Label
j Label
```

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`
Shift left logical	<<	<<	`sll`
Shift right logical	>>	>>	`srl`

Garcia, Nikolić

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 $\texttt{0000 00FF}_{hex}$ isolates the least significant byte
  - **andi** with $\texttt{FF00 0000}_{hex}$ isolates the most significant byte

No NOT in RISC-V

- There is no logical NOT in RISC-V
  - Use **`xor`** with `11111111`$_{two}$
  - Remember - simplicity…

Garcia, Nikolić

Logical Shifting

- Shift Left Logical (`sll`) and immediate (`slli`):

  `slli x11,x12,2 #x11=x12<<2`

  - Store in **x11** the value from **x12** shifted by 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?
- Shift Right: `srl` is opposite shift; >>

Arithmetic Shifting

- Shift right arithmetic (**`sra`**, **`srai`**) moves *n* bits to the right (insert high-order sign bit into empty bits)
- For example, if register x10 contained

$\texttt{1111 1111 1111 1111 1111 1111 1110 0111}_{two} = -25_{ten}$

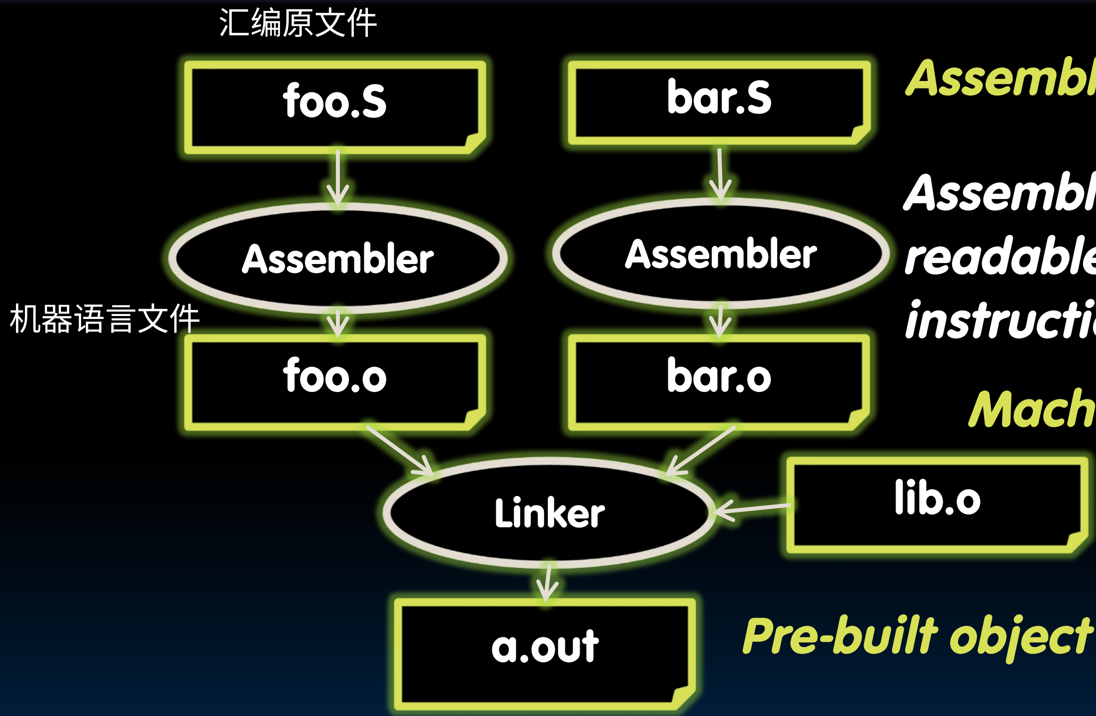
- If execute **`srai x10, x10, 4`**, result is:

$\texttt{1111 1111 1111 1111 1111 1111 1111 1110}_{two} = -2_{ten}$

- 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

A Bit About Machine Program

Assembler to Machine Code (More Later in Course)



Assembler source files (text)

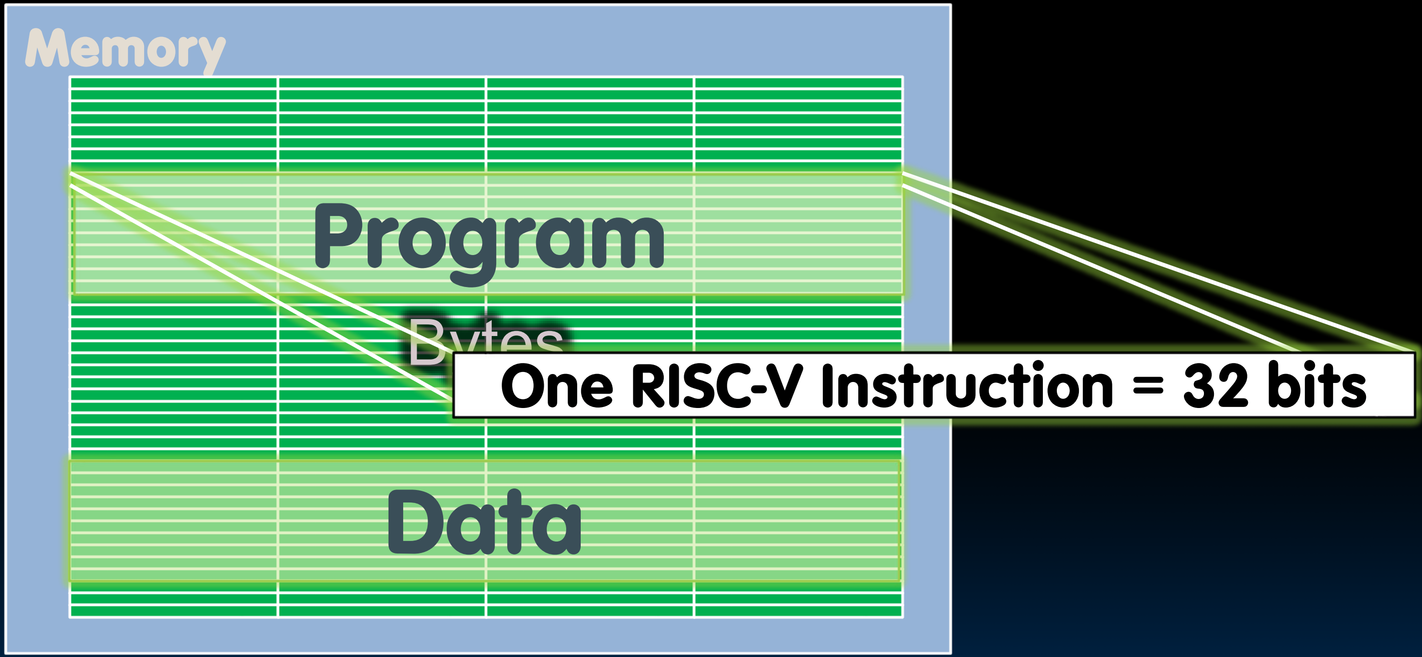
Assembler converts human-readable assembly code to instruction bit patterns

Machine code object files

Pre-built object file libraries

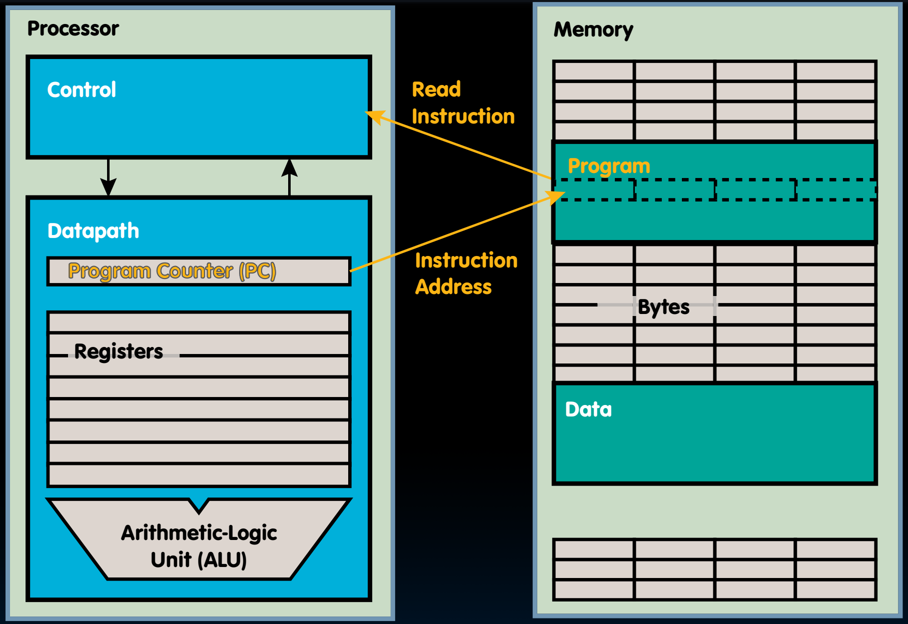
Machine code executable file 可执行文件

How Program is Stored



Garcia, Nikolić

Program Execution



- **PC** (program counter) is a register internal to the processor that holds 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 PC (default add +4 bytes to PC, to move to next sequential instruction; branches, jumps alter)

Helpful RISC-V Assembler Features

- Symbolic register names
  - E.g., `a0-a7` for argument registers (`x10-x17`) for function calls
  - E.g., `zero` for `x0`
- 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`
  - E.g., `nop = addi x0, x0, 0`

RISC-V Function Calls

C Functions

```c
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?

Garcia, Nikolić

Six Fundamental Steps in Calling a Function

1. Put arguments 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 return value in a place where calling code can access it and restore any registers you used; release local storage
6. Return control to point of origin, since a function can be called from several points in a program

Garcia, Nikolić

RISC-V Function Call Conventions

尽量避免经常使用lw，sw，从内存中读取数据大概需要50ns，所以尽可能使用寄存器

- Registers faster than memory, so use them
- **a0–a7 (x10-x17)**: eight *argument* registers to pass parameters and two 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)

Garcia, Nikolić

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 RISC-V, all 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 mv a0,s0   addi a0, s0, 0   # x = a
1004 mv a1,s1                    # y = b
1008 addi ra,zero,1016           #ra=1016
1012 j    sum                    #jump to sum
1016 …                           # next inst.
…
2000 sum: add a0,a0,a1
2004 jr  ra #new instr."jump reg"
```


Garcia, Nikolić

Instruction Support for Functions (3/4)

C

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

RISC-V

- Question: Why use `jr` here? Why not use `j`?

j指令是无条件跳转，jr是间接跳转指令，j是挑转到标签，而jr是跳转到寄存器存储的地址

- 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.

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

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`!

Garcia, Nikolić

RISC-V Function Call Instructions

- Invoke function*: jump and link* instruction (`jal`)

  (really should be `laj` *"link and jump"*)

  - "link" means form an *address* or *link* that points to calling site to allow function to return to proper address
  - Jumps to address and simultaneously saves the address of the <u>following</u> instruction in register ra

```
jal FunctionLabel
```

- Return from function: *jump register* instruction (`jr`)
  - Unconditional jump to address specified in register: `jr ra`
  - Assembler shorthand: `ret = jr ra`

Garcia, Nikolić

Summary of Instruction Support

Actually, only two instructions:

- `jal rd, Label` – jump-and-link
- `jalr rd,` $\boxed{\texttt{rs, imm} \text{ – jump-and-link register}}$

保存返回地址 对跳转的地址添加一个偏移量

`j`, `jr` and `ret` are pseudoinstructions!

- `j:` `jal x0, Label`