CSC 3210 Computer organization and programming chapter 2

Chunlan Gao

Georgia State University

Synchronization in RISC-V

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Result depends on order of accesses
- Load reserved: `lr.w rd,(rs1)`
  - Load value from address in rs1 to rd
  - Place reservation on memory address
- Store conditional: `sc.w rd,(rs1),rs2`
  - Store from rs2 to address in rs1 in memory
  - Succeeds if location not changed since the `lr.w`
    - Returns 0 in rd
  - Fails if location is changed
    - Returns non-zero value in rd

Synchronization in RISC-V

- Example 1: atomic swap (to test/set lock variable)

```
again:   lr.w x10,(x20)
         sc.w x11,(x20),x23 // X11 = status
         bne  x11,x0,again  // branch if store failed
         addi x23,x10,0     // X23 = loaded value
```

- Example 2: lock

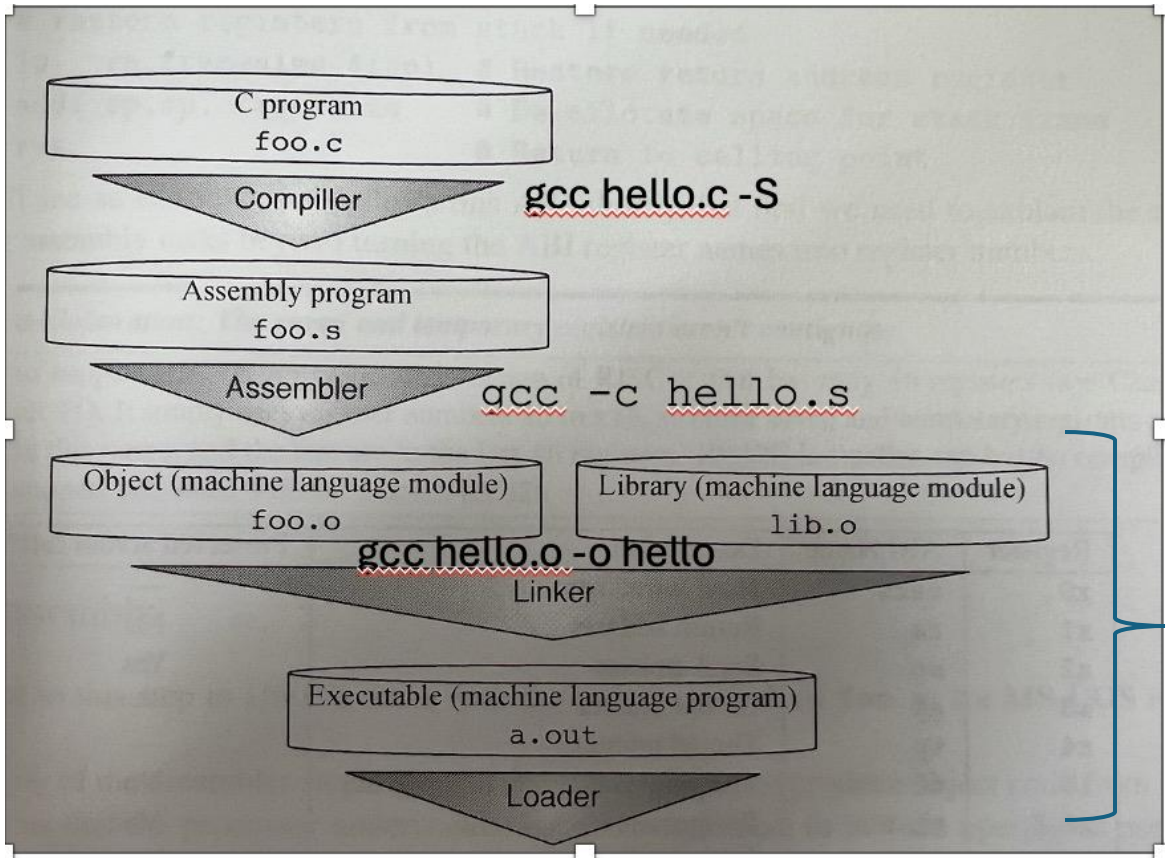
```
         addi x12,x0,1              // copy locked value
again:   lr.w x10,(x20)             // read lock, we use x20 as a lock
         bne  x10,x0,again  // check if it is 0 yet
         sc.w x11,(x20),x12 // attempt to store
         bne  x11,x0,again  // branch if fails
        ****                                              lw x13, (x30)
        Do something here                                 addi x13, x13, 1
        ****                                               sw x13, (x30)
```

- Unlock:

```
         sd   x0,0(x20)             // free lock
```

Translating and Starting a Program

Unix



Many compilers produce object modules directly

Static linking

Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
- The object file for Unix system typically contain six distinct pieces:

Component	Purpose
Header	Describes the structure of the object module.
Text Segment	Contains the machine instructions (program code).
Static Data Segment	Stores global/static variables that persist for the lifetime of the program.
Relocation Info	Provides details for adjusting memory addresses during linking.
Symbol Table	Lists global symbols and external references for linking.
Debug Info	Helps map machine code back to source code for debugging.

Linker: Linking Object Modules

linker, which takes all the independently assembled machine language programs and “stitches” them together.

Three steps for the linker:

1. Merges segments(place code and data modules symbolically in memory)
2. Resolve labels (determine their addresses)
3. Patch location-dependent and external refs

- Could leave location dependencies for fixing by a relocating loader
- But with virtual memory, no need to do this
- Program can be loaded into absolute location in virtual memory space

The resulting executable file is stored on disk, but it is now ready to be run, with all symbols resolved and addresses set.

Loading a Program

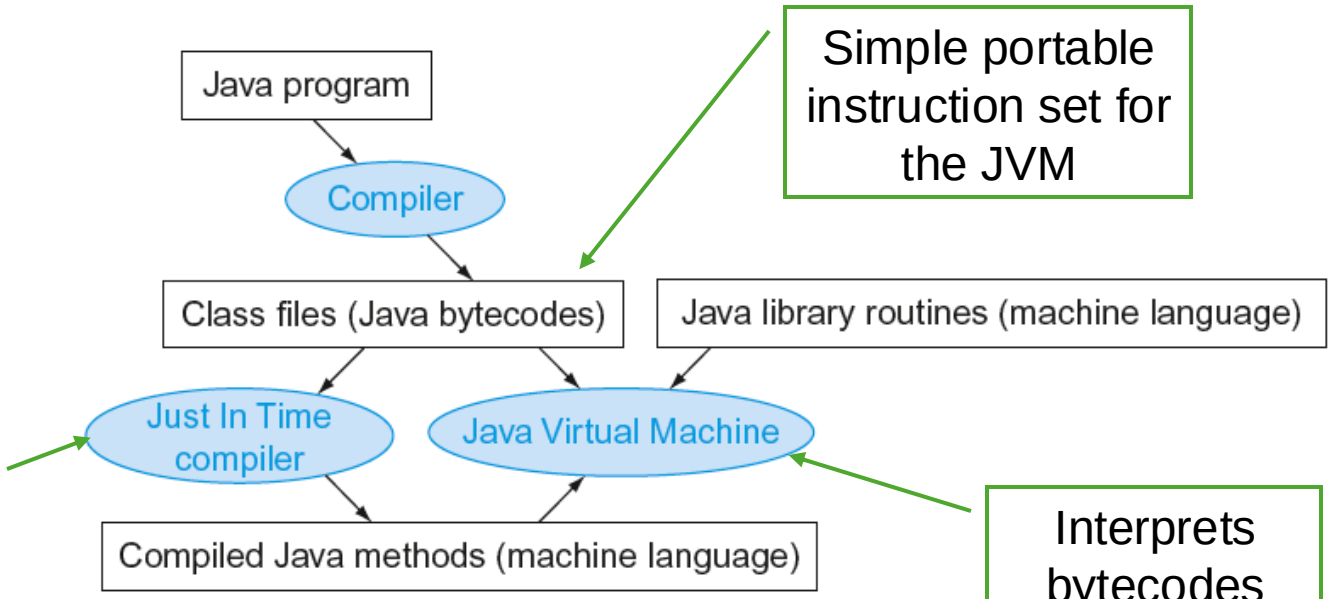
- Load from file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space
  3. Copy text and initialized data into memory
  4. Set up arguments on stack(copy the parameters to the stack)
  5. Initialize the processor registers and sets the stack pointer to the first free location
  6. Jump to startup routine
  - The **start-up routine** prepares the environment (like setting up arguments).
  - It calls the **main function** of the program.
  - Once the **main function** finishes, the **start-up routine** performs the clean-up, usually by calling the operating system's **exit system call** to terminate the program.

Dynamic Linking

- Only link/load library procedure when it is called
  - Requires procedure code to be relocatable
  - Avoids image bloat caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions

Starting a Java Program

- Run safely on any computer, even if it might slow execution time



Simple portable instruction set for the JVM

Compiles bytecodes of "hot" methods into native code for host machine

Interprets bytecodes

The downside is lower performance

C Sort Example to put it all together

- In this part, we derive the RISC-V code from two procedures written in C
- 1. swap array elements
- 2. sort the elements

1. Leaf Procedure swap

- Swap procedure (leaf)

```
void swap(int v[], int k)
  {
     int temp;
     temp = v[k];
     v[k] = v[k+1];
     v[k+1] = temp;
  }
```

- v in x10, k in x11, temp in x5

swaps two locations in memory v[k] and v[k+1]

The RISC-V of Procedure Swap

v in x10, k in x11, temp in x5

```
swap:
   slli x6,x11,2     // reg x6 = k * 4
   add  x6,x10,x6    // reg x6 = v + (k * 4)
   lw   x5,0(x6)     // reg x5 (temp) = v[k]
   lw   x7,4(x6)     // reg x7 = v[k + 1]
   sw   x7,0(x6)     // v[k] = reg x7
   sw   x5,4(x6)     // v[k+1] = reg x5 (temp)
   jalr x0,0(x1)     // return to calling routine
```

The Sort Procedure in C

- Non-leaf (caller to swap)

```
void sort(int v[], size_t n){

size_t i, j;

for (i = 0; i < n; i++) {  // Outer loop for each element
    for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j--)
{  // Inner loop to compare elements
                swap(v, j);  // Swap elements if needed
            }
     }
}
```

- v in x10, n in x11, i in x19, j in x20

The Outer Loop

- Skeleton of outer loop:
  - for (i = 0; i <n; i += 1) {

```
   li   x19,0              // i = 0
for1tst:
   bge  x19,x11,exit1   // go to exit1 if x19 ≥ x11 (i≥n)

   (body of outer for-loop)

   addi x19,x19,1       // i += 1
   j    for1tst         // branch to test of outer loop
exit1:
```

The Inner Loop

- Skeleton of inner loop:
  - for (j = i – 1; j >= 0 && v[j] > v[j + 1]; j – = 1) {

```
    addi x20,x19,-1     // j = i –1
for2tst:
    blt  x20,x0,exit2   // go to exit2 if X20 < 0 (j < 0)
    slli x5,x20,2        // reg x5 = j * 4
    add  x5,x10,x5      // reg x5 = v + (j * 4)
    lw   x6,0(x5)       // reg x6 = v[j]
    lw   x7,4(x5)       // reg x7 = v[j + 1]
    ble  x6,x7,exit2    // go to exit2 if x6 ≤ x7
    mv   x21, x10       // copy parameter x10 into x21
    mv   x22, x11       // copy parameter x11 into x22
    mv   x10, x21       // first swap parameter is v
    mv   x11, x20       // second swap parameter is j
    jal  x1,swap          // call swap
    addi x20,x20,-1       // j -= 1
    j    for2tst          // branch to test of inner loop
exit2:
```

Preserving Registers

- Preserve saved registers:

```
addi sp,sp,-20  // make room on stack for 5 regs
sw   x1,16(sp)  // save x1 on stack      return position
sw   x22,12(sp) // save x22 on stack
sw   x21,8(sp) // save x21 on stack
sw   x20,4(sp)  // save x20 on stack
sw   x19,0(sp)  // save x19 on stack
```

- Restore saved registers:

```
exit1:
   lw   x19,0(sp)      // restore x19 from stack
   lw   x20,4(sp)  // restore x20 from stack
   lw   x21,8(sp) // restore x21 from stack
   lw   x22,12(sp) // restore x22 from stack
   lw   x1,16(sp)  // restore x1 from stack
   addi sp,sp, 20  // restore stack pointer
  jalr x0,0(x1)
```

The assembly version of procedure

Saving registers			
	sort:	addi sp, sp, -20	# make room on stack for 5 registers
		sw x1, 16(sp)	# save return address on stack
		sw x22, 12(sp)	# save x22 on stack
		sw x21, 8(sp)	# save x21 on stack
		sw x20, 4(sp)	# save x20 on stack
		sw x19, 0(sp)	# save x19 on stack

Procedure body			
Move parameters		addi x21, x10, 0	# copy parameter x10 into x21
		addi x22, x11, 0	# copy parameter x11 into x22
Outer loop		addi x19,x0, 0	# i = 0
	for1tst:	bge x19, x22, exit1	# go to exit1 if i >= n
Inner loop		addi x20, x19, -1	# j = i - 1
	for2tst:	blt x20, x0, exit2	# go to exit2 if j < 0
		slli x5, x20, 2	# x5 = j * 4
		add x5, x21, x5	# x5 = v + (j * 4
		lw x6, 0(x5)	# x6 = v[j]
		lw x7, 4(x5)	# x7 = v[j + 1]
		ble x6, x7, exit2	# go to exit2 if x6 < x7
Pass parameters and call		addi x10, x21, 0	# first swap parameter is v
		addi x11, x20, 0	# second swap parameter is j
		jal x1, swap	# call swap
Inner loop		addi x20, x20, -1	j for2tst
		jal, x0 for2tst	# go to for2tst
Outer loop	exit2:	addi x19, x19, 1	# i += 1
		jal, x0 for1tst	# go to for1tst

Restoring registers			
	exit1:	lw x19, 0(sp)	# restore x19 from stack
		lw x20, 4(sp)	# restore x20 from stack
		lw x21, 8(sp)	# restore x21 from stack
		lw x22, 12(sp)	# restore x22 from stack
		lw x1, 16(sp)	# restore return address from stack
		addi sp, sp, 20	# restore stack pointer

Procedure return			
		jalr x0, 0(x1)	# return to calling routine

Arrays vs. Pointers

- Array indexing involves (base + index * element_size)
  - Multiplying index by element size
  - Adding to array base address
- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity

Example: Clearing an Array

```
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

```
   li   x5,0       // i = 0
loop1:
   slli x6,x5,3    // x6 = i * 8
   add  x7,x10,x6  // x7 = address
                   // of array[i]
   sd   x0,0(x7)   // array[i] = 0
   addi x5,x5,1    // i = i + 1
   blt  x5,x11,loop1  // if (i<size)
                      // go to loop1
```

```
clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p < &array[size];
       p = p + 1)
    *p = 0;
}
```

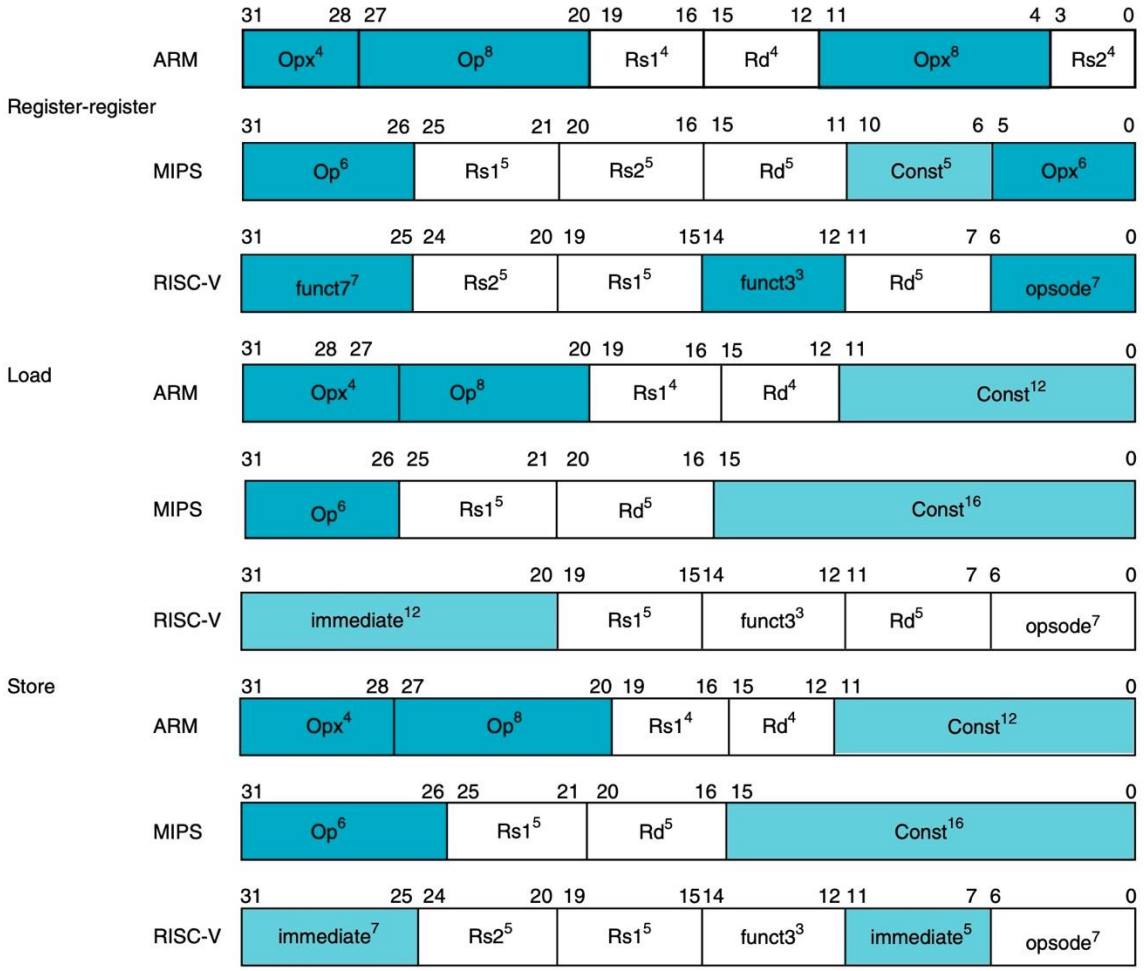
```
   mv x5,x10       // p = address
                   // of array[0]
   slli x6,x11,3   // x6 = size * 8
   add x7,x10,x6   // x7 = address
                   // of array[size]
loop2:
   sd x0,0(x5)     // Memory[p] = 0
   addi x5,x5,8    // p = p + 8
   bltu x5,x7,loop2
                   // if (p<&array[size])
                   // go to loop2
```

Comparison of Array vs. Pointer

- Multiply “strength reduced” to shift
- Array version requires shift to be inside loop
  - Part of index calculation for incremented i
  - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
  - Induction variable elimination
  - Better to make program clearer and safer

Reduced Instruction Set Computing

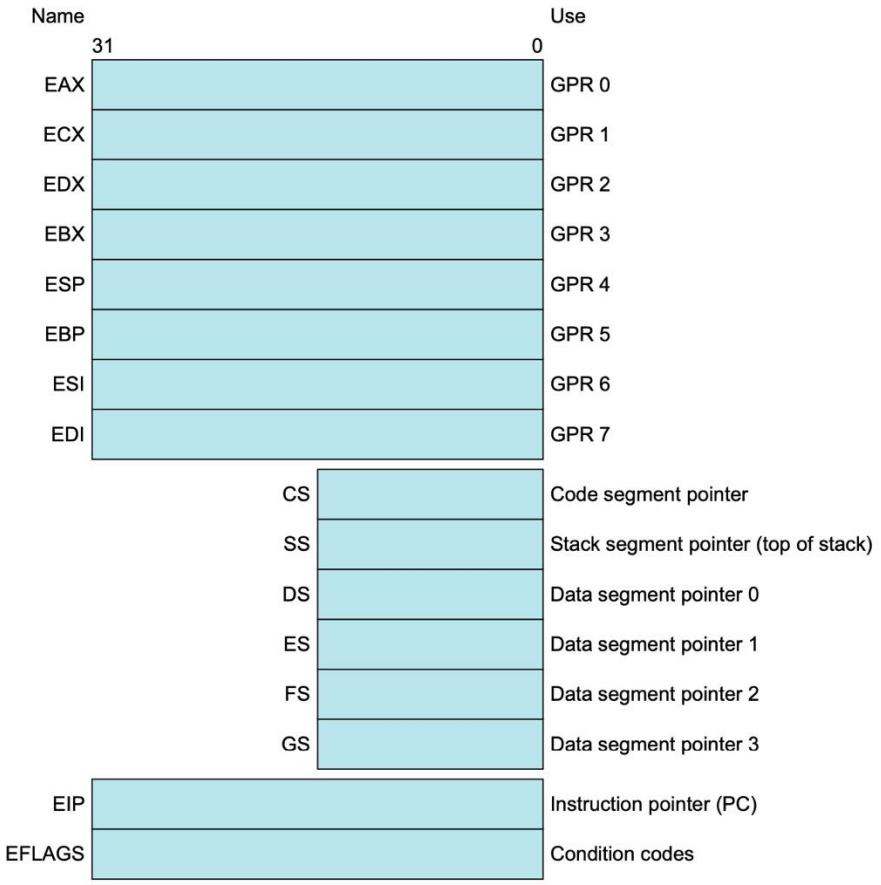
MIPS (Microprocessor without Interlocked Pipeline Stages)
ARM (Advanced RISC Machine)
RISC-V(Reduced Instruction Set Computing - Five)



Intel x86 ISA

Designers of instruction sets sometimes provide more powerful operations than those found in RISC-V and MIPS. The goal is generally to reduce the number of instructions executed by a program. The danger is that this reduction can occur at the cost of simplicity, increasing the time a program takes to execute because the instructions are slower. This slowness may be the result of a slower clock cycle time or of requiring more clock cycles than a simpler sequence.

Intel x86 ISA –register &



Instruction	Meaning
Control	**Conditional and unconditional branches**
`jnz, jz`	Jump if condition to EIP + 8-bit offset; `JNE` (for `JNZ`), `JE` (for `JZ`) are alternative names
`jmp`	Unconditional jump—8-bit or 16-bit offset
`call`	Subroutine call—16-bit offset; return address pushed onto stack
`ret`	Pops return address from stack and jumps to it
`loop`	Loop branch—decrement ECX; jump to EIP + 8-bit displacement if ECX≠0
Data transfer	**Move data between registers or between register and memory**
`move`	Move between two registers or between register and memory
`push, pop`	Push source operand on stack; pop operand from stack top to a register
`les`	Load ES and one of the GPRs from memory
Arithmetic, logical	**Arithmetic and logical operations using the data registers and memory**
`add, sub`	Add source to destination; subtract source from destination; register-memory format
`cmp`	Compare source and destination; register-memory format
`shl, shr, rcr`	Shift left; shift logical right; rotate right with carry condition code as fill
`cbw`	Convert byte in eight rightmost bits of EAX to 16-bit word in right of EAX
`test`	Logical AND of source and destination sets condition codes
`inc, dec`	Increment destination, decrement destination
`or, xor`	Logical OR; exclusive OR; register-memory format
String	**Move between string operands; length given by a repeat prefix**
`movs`	Copies from string source to destination by incrementing ESI and EDI; may be repeated
`lods`	Loads a byte, word, or doubleword of a string into the EAX register

The Rest of the RISC-VInstruction Set

- Base integer instructions (RV64I)
  - Those previously described, plus
  - auipc rd, immed // rd = (imm<<12) + pc
    - follow by jalr (adds 12-bit immed) for long jump
  - slt, sltu, slti, sltui: set less than (like MIPS)
  - addw, subw, addiw: 32-bit add/sub
  - sllw, srlw, srlw, slliw, srliw, sraiw: 32-bit shift
- 32-bit variant: RV32I
  - registers are 32-bits wide, 32-bit operations

Instruction Set Extensions

- M: integer multiply, divide, remainder
- A: atomic memory operations
- F: single-precision floating point
- D: double-precision floating point
- C: compressed instructions
  - 16-bit encoding for frequently used instructions

Concluding Remarks

- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Good design demands good compromises
- Make the common case fast
- Layers of software/hardware
  - Compiler, assembler, hardware
- RISC-V: typical of RISC ISAs
  - c.f. x86

Chapter 2

Done