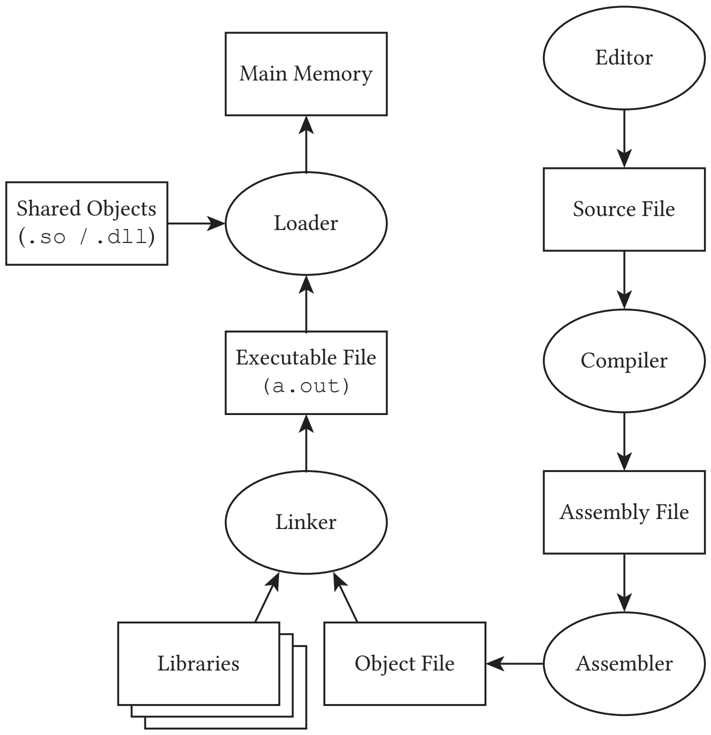
Instructions

August 28, 2024

1 How do Instructions get into Memory?



2 What kinds of Instructions should the CPU Have?

- Arithmetic Instructions
  - Add/Subtract
  - Multiply/Divide
- Logic
  - AND, OR, NOT, XOR
- Shifts
  - Left/Right Shift
  - Rotate
- Control
  - Branch (usually conditional)
  - Jump (usually unconditional)
  - Jump & Link (for procedure linkage)
  - Return from procedure
- Memory
  - Load

- Store
  - Addressing Modes
  - Granularity (Number of bytes)
  - Endianness
  - Byte vs. Word Addressability

Granularity How many bytes do you load or store when you access memory? Depends on what type you are loading or storing.

- **char** one byte
- **short** two bytes
- **int** four bytes
- **long** eight bytes
- **char***, **int***, etc four or eight bytes
- **float** four bytes
- **double** eight bytes

Alignment `Address MOD Granularity = 0` if aligned

Example: Aligned store of a short

- Address = `0x7FFF0704`
- Granularity of short = 2 bytes
- `0x7FFF0704 MOD 2` = 0, so it is an aligned store.

Example: Unaligned load of an int

- Address = `0x7FFF070A`
- Granularity of int = 4 bytes
- `0x7FFF070A MOD 4` = 2, so it is an UNaligned load.

3 Instruction Formats

RISC-V defines four types of instructions:

- R-Type: Used for register-register operations.
- I-Type: Used for register-immediate operations.
- S-Type: Alternative format for register-immediate operations.
- U-Type: Used for instructions that need long immediate values.

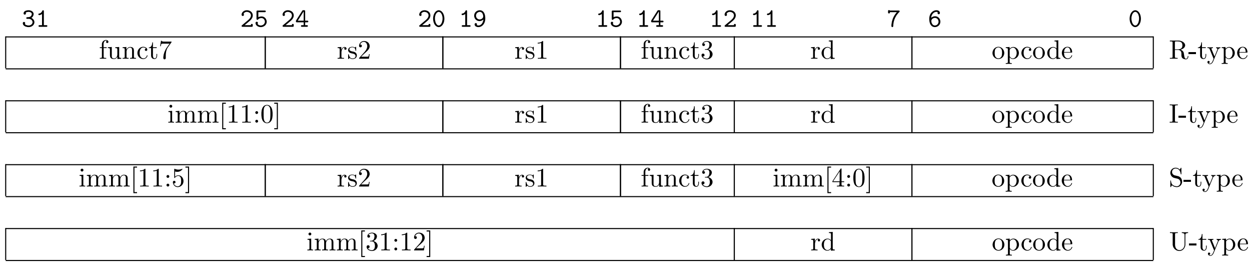

31 25 24 20 19 15 14 12 11 7 6 0
funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type
imm[11:0] | rs1 | funct3 | rd | opcode | I-type
imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type
imm[31:12] | rd | opcode | U-type

Figure 1: RISC-V Instruction Types

3.1 Operands

- **Register:**
  - `value = R[rx]` (source)
  - `R[rx] = value` (destination)
- **Immediate:** `imm = sext(I[11:0])`
- **Register Indirect:** `effective address = R[rx]`
- **Base + Displacement:** `effective address = R[rx] + sext(imm)`
  Special case: PC Relative: `effective address = PC + sext(imm)`
- **Memory Indirect:** `M[M[R[rx]]]`
- **Scaled:** `effective address = R[rx] + k * R[ry]`
- **Indexed:** `effective address = R[rx] + R[ry]`
- **Absolute:** `effective address = imm`
- **Autoincrement/decrement:** `effective address = R[rx] + disp`
  `R[rx] += datatype size`
  or
  `R[rx] -= datatype size`
- **Base + Displacement with update:** `effective address = R[rx] + displacement`
  `R[rx] += displacement`

How many operands does an instruciton need? Usually 3: two source, one destination.
Exceptions:

- **Division:** two source, two destination (quotient + remainder)
- **Multiplication:** may need a double-precision result, so two destination registers
- **Multiply-Add:** $d \leftarrow a \times b + c$

4 What do we Want from an ISA?

For the Hardware: Make it easy to implement

- Fast decode, fast fetch: need fixed size instructions

- Easy to Pipeline
- Easy to parallelize: Fixed formats

For the Software: Make it easy to compile for

- **Regularity:** If you do someting one way in one place, you should do it the same way in other places. Example of an irregular instruction is bit shifts on x86. Only `CX`/`ECX`/`RCX` can be used as the shift count register.
- **Composability**: You should be able to do any operation by combining instructions.
- **Orthogonality**
  - Addressing mode is independent from instruction operation
  - This is in opposition to load/store architecture

5 x86

x86 has variable length instructions (1-15 bytes). It is not possible to have fixed field placement.

5.1 Instruction Formats

5.1.1 One-Byte Opcodes

prefix	opcode	operand1	operand2	...

Example

```
mov $0, -4(%rbp)

c7 45 fc 00 00 00 00
```

- `0xc7` MOV opcode
- `0x45` BP+OFFSET Addressing Mode
- `0xfc` OFFSET
- `0x00 00 00 00` Immediate

5.1.2 Two-Byte Opcodes

prefix	`0x0f`	opcode	operand1	operand2	...

Example

```
jne 34

0f 85 22 00 00 00
```

- `0x0f` Multibyte opcode
- `0x85` jne opcode
- `0x22 00 00 00` Jump target absolute address

6 RISC-V

6.1 Registers

Register Names	ABI Names	Description
`x0`	`zero`	Hard-wired zero
`x1`	`ra`	Return address
`x2`	`sp`	Stack pointer
`x3`	`gp`	Global pointer
`x4`	`tp`	Thread pointer
`x5`	`t0`	Temporary/alternate link register
`x6-7`	`t1-t2`	Temporary registers
`x8`	`s0/fp`	Saved register/frame pointer
`x9`	`s1`	Saved register
`x10-11`	`a0-a1`	Function argument/return value registers
`x12-17`	`a2-a7`	Function argument registers
`x18-27`	`s2-s11`	Saved registers
`x28-31`	`s3-s6`	Temporary registers

6.2 Instruction Types

R-Format Used for arithmetic and logical instructions

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

I-Format Used for arithmetic and logical instructions with one immediate operand

```
addi x1, x2, 1
```

S-Format Alternative format for instructions with immediate operand

```
sd x1, 4(x9)
```

U-Format Used for instructions that need long immediate fields

```
lui x2, 0xfffff000
```