Von Neumann Machine and RISC

→ Reduced Instruction Set Computer (RISC): DLX style

- arithmetic unit: 32 32-bit registers (reg[0], ..., reg[31]) (=0!)
- control unit:
  - instruction register (ir)
  - program counter (pc)
  - reg[31] is link register (for return addresses)
- store (memory): n 32-bit words, byte-addressed (mem[0], ..., mem[n-1])

contains code, global variables, constants, heap, stack

→ implement loader and interpreter:

1. load code and initialize pc
2. fetch instruction from mem[pc/4] into ir (byte-addressed!)
3. decode instruction into op, a, b, c (parameters)
4. execute instruction
5. goto 2.

op code

Instruction formats:

64 op codes; 32 registers; immediate addressing (for data)

	6 bits	5 bits	5 bits	16 bits sign-extended to 32 bits
F1	op	$0 \le a \le 2^5-1$	b	$-2^{15} \le c \le 2^{15}-1$

register addressing (for data)

	6 bits	5 bits	5 bits	11 bits	→ 5 bits
F2	op	a	b	unused	c

absolute jumps

	6 bits	26 bits
F3	op	$0 \le c \le 2^{26}-1$

Decoding Instructions

```
int op;
int a;
int b;
int c;

int instruction;

decode() {
 // in linker and emulator!
 // works for format F1 and F2 but not F3
 // assuming: 0 <= instruction <= 2^32-1
 op = (instruction >> 26) & 63; // 0x3F: 6 lsbs
 a = (instruction >> 21) & 31; // 0x1F: 5 lsbs
 b = (instruction >> 16) & 31; // 0x1F: 5 lsbs
 c = instruction & 65535; // 0xFFFF: 16 lsbs
 if (c >= 32768)
   c = c - 65536; // 0x10000: 2^16
}

decodeF3() {
 // in linker and emulator!
 // works for format F3 only
 // assuming: 0 <= instruction <= 2^32-1
 op = (instruction >> 26) & 63; // 0x3F: 6 lsbs
 c = instruction & 67108863; // 0x3FFFFFF: 26 lsbs
}
```

using only 6 lsbs here

5 lsbs

5-26 lsbs (plus sign)

32 bits

logical shift (>>> in Java, >> in C on unsigned int)

mask

c will be negative

sign-preserving extension to 32 bits (from 16 bits)

mask 26 lsbs

Register Instructions

Format I1: immediate addressing

a constant!

byte-addressed

ADDI a, b, c: reg[a] = reg[b] + c; pc = pc + 4;
SUBI a, b, c: reg[a] = reg[b] - c; pc = pc + 4;
MULI a, b, c: reg[a] = reg[b] * c; pc = pc + 4;
DIVI a, b, c: reg[a] = reg[b] / c; pc = pc + 4;
MODI a, b, c: reg[a] = reg[b] % c; pc = pc + 4;
CMPI a, b, c: reg[a] = reg[b] - c; pc = pc + 4;

sign-extended to 32 bits!

→ reg[a] == 0 if reg[b] == c
reg[a] >= 0 if reg[b] >= c
reg[a] > 0 if reg[b] > c
reg[a] =< 0 if reg[b] =< c
reg[a] < 0 if reg[b] < c
reg[a] != 0 if reg[b] != c

no overflow!
→ unlike SUBI (and DIVI) which may trap

hw: just report overflow

Format I2: register addressing

ADD a, b, c: reg[a] = reg[b] + reg[c]; pc = pc + 4;
SUB a, b, c: reg[a] = reg[b] - reg[c]; pc = pc + 4;
MUL a, b, c: reg[a] = reg[b] * reg[c]; pc = pc + 4;
DIV a, b, c: reg[a] = reg[b] / reg[c]; pc = pc + 4;
MOD a, b, c: reg[a] = reg[b] % reg[c]; pc = pc + 4;
CMP a, b, c: reg[a] = reg[b] - reg[c]; pc = pc + 4;

other instructions (not needed here):

- CHK: check array bounds
- LSH, ASH: logical, arithmetic shift
- boolean operators

Memory Instructions

Format F1: load and store word

byte-addressed

LDW a, b, c: reg[a] = mem[(reg[b] + c)/4]; pc = pc + 4;

STW a, b, c: mem[(reg[b] + c)/4] = reg[a]; pc = pc + 4;

register-relative addressing

Format F1: pop and push

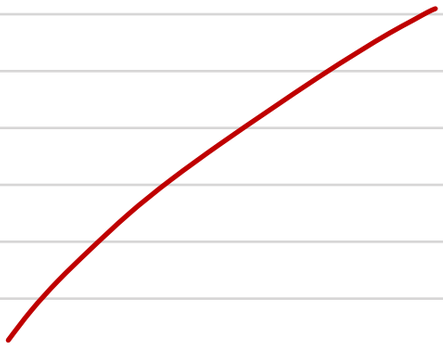
stack pointer

POP a, b, c: reg[a] = mem[reg[b]/4]; reg[b] = reg[b] + c;
pc = pc + 4;

PSH a, b, c: reg[b] = reg[b] - c; mem[reg[b]/4] = reg[a];
pc = pc + 4;

size of popped chunk

stack grows from high to low addresses



Control instructions

Format F1: conditional branching (pc-relative)

BEQ a, c: if (reg[a] == 0) pc = pc + c * 4; else pc = pc + 4;
BGE a, c: if (reg[a] >= 0) pc = pc + c * 4; else pc = pc + 4;
BGT a, c: if (reg[a] > 0) pc = pc + c * 4; else pc = pc + 4;
BLE a, c: if (reg[a] <= 0) pc = pc + c * 4; else pc = pc + 4;
BLT a, c: if (reg[a] < 0) pc = pc + c * 4; else pc = pc + 4;
BNE a, c: if (reg[a] != 0) pc = pc + c * 4; else pc = pc + 4;

byte-addressed

Format F1: unconditional branching (pc-relative)

BR c: pc = pc + c * 4;
BSR c: reg[31] = pc + 4; pc = pc + c * 4;

link register (hardware convention)

branch to subroutine

Format F2: return from subroutine

RET c: pc = reg[c];

Format F3: unconditional jump (absolute)

JSR c: reg[31] = pc + 4; pc = c;

jump to subroutine

I/O Instructions

Format F2: file managment

```
FLO a, b, c:
open file (pointer to file name string: reg[a];
           pointer to mode string "r" or "w": reg[b]) {
  ...fopen...
  reg[c] = file descriptor;
}
FLC c:
close file (file descriptor: reg[c]) {
  ...fclose...
}
```

fits C API

Format F2: reading and writing

```
RDC a, c:
read character from open file (file descriptor: reg[a]) {
  ...fread...
  reg[c] = read character;
}
WRC a, c:
write character to open file (file descriptor: reg[a];
                              character: reg[c]) {
  ...fwrite...
}
```

→ other instructions are possible

Requirement:

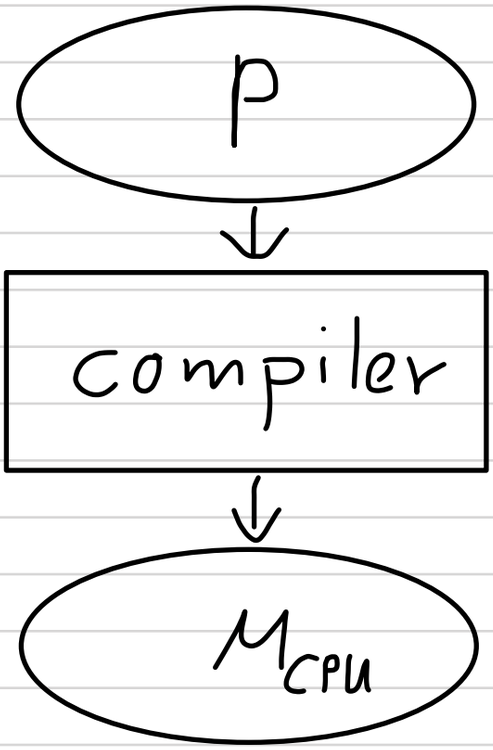
- your emulator must be "close" to real processor!

System Interface

- system procedures: open, close, read, write, printf, malloc
- there are two choices:
  1. written by hand in M_{RISC} code
  2. written by hand in language in which simulator is written

code is provided:
1. by compiler, or
2. as library, or
3. in simulator

(for both)
↝ requires understanding how procedure stack is constructed for passing parameters



code is part of simulator
↝ simulator becomes virtual machine