CS 3650 – Computer Systems | Prof. Peter Desnoyers
CPU Description (Lab 1 documentation)

Fixes to errors in the description of the JMP_IND, OR, and SHIFTR instructions are highlighted

Overview

The 3650 CPU is a single-core 16-bit computer able to address 2^{16} (65536) bytes of memory. It has 10 16-bit registers - R0 through R7, PC (program counter) and SP (stack pointer), plus two boolean flag registers: Z (zero) and N (negative); its state is represented by a C structure defined in the file `lab1.h`:

```
struct cpu {
    uint8_t *memory;  /* memory */
    uint16_t R[8]; /* registers */
    uint16_t PC;   /* program counter */
    uint16_t SP;   /* stack pointer */
    bool     Z;    /* zero flag */
    bool     N;    /* negative flag */
};
```

16-bit values are stored in memory in “little-endian” order[1]; you can read and write 16-bit values from memory with the following code:

```
uint16_t load2(struct cpu *c, uint16_t addr) {
    uint16_t data = (c->memory[addr] | (c->memory[addr+1] << 8));
    return data;
}
void store2(struct cpu *c, uint16_t data, uint16_t addr) {
    c->memory[addr] = data & 0xFF;
    c->memory[addr+1] = (data >> 8) & 0xFF;
}
```

You will implement the following function:

```
bool halted = emulate(struct cpu *cpu);
```

which will (a) implement the instruction, and (b) return `false` if the emulation should continue (i.e. a normal instruction) or `true` if the emulation should stop because the executed instruction was HALT.

Instruction format

Basic instructions are 16 bits (2 bytes) with a format that looks like this:

i	i	i	i												

where ‘iiii’ is a 4-bit instruction code:

- 0001 (1) SET - load a value into a register
- 0010 (2) LOAD - read from memory into a register
- 0011 (3) STORE - store a register into memory
- 0100 (4) MOVE - copy one register to another
- 0101 (5) ALU - arithmetic and logic operations (add/sub/etc.)
- 0110 (6) JMP_ABS - jump to an address specified in the 3rd and 4th instruction bytes
- 0111 (7) JMP_IND - jump to an address in a register (“indirect”)
- 1000 (8) CALL (absolute) - function call to address in 3rd and 4th bytes

1 See https://www.rfc-editor.org/ien/ien137.txt for the origin of the terms “little-endian” and “big-endian”.

- 1001 (9) CALL (register indirect) - function call to address in register
- 1010 (10/0xA) RET - return
- 1011 (11/0xB) PUSH - push a register onto the stack
- 1100 (12/0xC) POP - pop a register from stack
- 1101 (13/0xD) IN - perform input operation (read byte from stdin to register)
- 1110 (14/0xE) OUT - perform output operation (write byte from register to stdout)
- 1111 (15/0xF) HALT - return `true` from the `emulate()` function

You can read the next instruction and determine the instruction code with logic like this:

```
uint16_t insn = load2(cpu, cpu→PC);
if ((insn & 0xF000) == 0x1000) {
 ; /* SET */
}
else if ((insn & 0xF000) == 0x2000) {
 ; /* LOAD */
}
...
```

Although a few of the instructions use all 16 bits, some of them don't need that many; these "don't-care" bits will be indicated with dots in the figures below. Read the fields from the instruction that you need, and ignore the remaining bits. (they'll probably be zero)

The 3650 CPU is a "load/store" architecture – in most cases you have to read data from memory into a register (with the LOAD instruction) before you can operate on it, and then write it back to memory with the STORE instruction.

You will have to extract bitfields to interpret many of the instructions – for example the arithmetic operation instruction format looks like this:

0	1	0	1	o	o	o	c	c	c	b	b	b	a	a	a

where 'ooo' is a 3-bit operation code (indicating add/subtract/etc.), and aaa, bbb, and ccc are three register numbers. To extract these you can use the following code:

```
int op = (insn >> 9) & 7;
int c = (insn >> 6) & 7;
int b = (insn >> 3) & 7;
int a = insn & 7;
```

SET Instruction (load constant value into register)

Example: `SET R1 = 0x1234`

0	0	0	1	·	·	·	·	·	·	·	·	·	a	a	a

value

This is a 4-byte instruction. It reads a 16-bit integer from addresses PC+2 and PC+3, puts the result into register Ra, and then increments the PC by 4 and returns `false` from emulate()

LOAD instruction (read from memory address into register)

This has multiple variants – it can load a 16-bit word or an 8-bit byte, and can read from an address specified in the instruction, or take its address from a register.

Examples:
```
LOAD R1 <- *0x5678
LOAD.B R2 <- *0x5678
LOAD R3 <- *R5
LOAD.B R4 <- *R5
```

Load from constant address: `LOAD Ra <- address`

0	0	1	0	0	B	-	-	-	-	-	-	-	a	a	a

address

Read 16 bits (if B=0) or 8 bits (if B=1) from the specified address and put the result in register Ra, then increment PC by 4 and return `false` from emulate()

Load from address in register: `LOAD Ra <- *Rb`

0	0	1	0	1	B	-	-	-	-	b	b	b	a	a	a

Read 16 bits (B=0) or 8 bits (B=1) from the address specified in register Rb, put the result in register Ra, increment PC by 2 and return `false` from emulate()

Useful code fragment:

```
int is_indirect = ((insn & 0x0800) != 0);
int is_byte = ((insn & 0x0400) != 0);
```

STORE instruction (store to memory address from register)

Again, this instruction has variants which store a 16-bit word or a single byte, and which include a constant address or which take their address from a register.

Examples:
```
STORE R1 -> *0x5678
STORE.B R2 -> *0x5678
STORE R3 -> *R5
STORE.B R4 -> *R5
```

It's basically the same as LOAD, but in the opposite direction, storing the value in Ra to the memory address specified in the instruction or in Rb, incrementing the PC by 4 or 2 when done, and again returning `false` from emulate().

Store to constant address: `STORE Ra -> address`

0	0	0	1	·	·	·	·	·	·	·	·	·	a	a	a

address

Store to address from register: `STORE Ra -> *Rb`

0	0	1	1	1	B	-	-	-	-	b	b	b	a	a	a

MOVE – copy one register to another: `MOV Rs -> Rd`

This has a different layout than the other instructions, because it allows you to specify any of the 8 general-purpose registers (R0 through R7) using values 0-7, or use 8 to specify the stack pointer. (if either register field is in the range 9-15, it's an error)

0	1	0	0	-	-	-	-	d	d	d	d	s	s	s	s

Copy the value in Rs (or SP, if ssss==8) into Rd (or SP, if dddd==8), then increment PC by 2. You can assume 0 <= S, D <= 8. Again, return `false` from emulate().

Arithmetic / logical operations

Most of these take 2 source registers and put their result in a third register.

0	1	0	1	o	o	o	c	c	c	b	b	b	a	a	a

Where the operation code (ooo) is one of:

- 000: `ADD Ra + Rb -> Rc`
- 001: `SUB Ra - Rb -> Rc`
- 010: `AND Ra & Rb -> Rc`
- 011: `OR Ra | Rb -> Rc`
- 100: `XOR Ra ^ Rb -> Rc`
- 101: `SHIFTR Ra >> Rb -> Rc` ("shift right")
- 110: `CMP Ra – Rb` ("compare": compute Ra - Rb, discard the result but set N and Z)
- 111: `TEST Ra` (set Z, N according to value in Ra)

All operations are done on 16-bit values in registers R0 through R7. When the operation is done, set the N and Z flags appropriately, put the result in the output register (except for TEST and CMP), and increment the PC by 2. Again, return `false` from emulate().

NOTE – since we're using 2s-complement arithmetic, a value is negative if its high-order bit is one. Thus:

```
uint16_t val = cpu->R[a] - cpu->R[b];
bool is_negative = (val & 0x8000) != 0;
```

More useful code fragments:

```
if ((insn & 0x0E00) == 0x0000) /* ADD */
if ((insn & 0x0E00) == 0x0200) /* SUB */
if ((insn & 0x0E00) == 0x0400) /* AND */
if ((insn & 0x0E00) == 0x0600) /* OR */
if ((insn & 0x0E00) == 0x0800) /* XOR */
if ((insn & 0x0E00) == 0x0A00) /* SHIFT right */
if ((insn & 0x0E00) == 0x0C00) /* CMP */
if ((insn & 0x0E00) == 0x0E00) /* TEST */
```

JMP - jump to address

Examples: `JMP 0x1234`
`JMP_NZ R3`

There are two forms of this instruction, taking an explicit address (4 bytes) or taking an address from a register (2 bytes). In addition the jump can be conditional, setting PC to the given address only if the specified condition is true, and otherwise incrementing it by 2 or 4.

Conditions:

- 000 : JMP (unconditional)
- 001 : JMP_Z (jump if zero, i.e. Z true)
- 010 : JMP_NZ (jump if non-zero, i.e. Z false)
- 011 : JMP_LT (jump less than, i.e. N true)
- 100 : JMP_GT (jump greater than, i.e. N false and Z false)
- 101 : JMP_LE (jump less or equal, i.e. N true or Z true)
- 110 : JMP_GE (jump greater or equal, i.e. N false)
- 111 : illegal instruction

Jump to explicit address:

0	1	1	0	c	c	c	-	-	-	-	-	-	-	-	-

address

If condition is true, set PC to address specified in bytes 3 and 4 of instruction; otherwise increment PC by 4. Again, return `false` from emulate().

Jump to register address:

0	1	1	1	c	c	c	-	-	-	-	-	-	a	a	a

If condition is true, set PC to value of Ra; otherwise increment PC by 2. Again, return `false` from emulate().

CALL – jump and push return address

Like JMP, this comes in 2-byte and 4-byte variants, taking the address from a register or specifying it explicitly. Unlike JMP, there is no conditional version. It pushes the address of the next instruction (“return address”) onto the stack and then jumps to the indicated address.

Specified address: `CALL address`

1	0	0	0	-	-	-	-	-	-	-	-	-	-	-	-

address

- Decrement the SP register by 2
- Store the 16-bit value (PC+4) to the address in SP
- Read a 16-bit value from bytes 3 and 4 of the instruction and set the PC register to that value.
- Return `false` from emulate()

Address in register: `CALL *Ra`

1	0	0	1	-	-	-	-	-	-	-	-	-	a	a	a

- Decrement the SP register by 2
- Store the 16-bit value (PC+2) to the address in SP
- Set the PC register to the16-bit value in register Ra.
- Return `false` from emulate()

RET – pop return address and jump to it

1	0	1	0	-	-	-	-	-	-	-	-	-	-	-	-

- Load a 16-bit value from the address in register SP and set PC to that value.
- Increment SP by 2.
- return `false` from emulate()

PUSH, POP – push a register to stack or pop from it

1	0	1	1	-	-	-	-	-	-	-	-	-	a	a	a

`PUSH Ra:`

- Decrement SP by 2
- Store 16-bit value in register Ra to address in SP
- Increment PC by 2
- Return `false` from emulate()

1	1	0	0	-	-	-	-	-	-	-	-	-	a	a	a

`POP Ra:`

- Read 16-bit value from address in SP, store in register Ra
- Increment SP by 2
- Increment PC by 2
- Return `false` from emulate()

Input/Output instructions

1	1	0	1	-	-	-	-	-	-	-	-	-	a	a	a

`IN Ra:`

Read an 8-bit character from the terminal and put it in register Ra, using the following code:

```
cpu->R[a] = fgetc(stdin);
```

Increment PC by 2, return `false` from emulate().

1	1	1	0	-	-	-	-	-	-	-	-	-	a	a	a

`OUT Ra:`

Take the 8-bit value in the lower 8 bits of Ra and output it to the terminal, using the following code:

```
fputc(cpu->R[a], stdout)
```

Increment PC by 2, return `false` from emulate().

HALT

Returns `true` from the emulate() function, indicating that the emulated CPU should halt.

1	1	1	1	-	-	-	-	-	-	-	-	-	-	-	