Specification Notes

Sunday, January 23, 2022 5:45 PM

Task: Design an ISA for our processor

- 9-bits fixed length instructions
- Specify:
  - What operations our ISA supports
    - their opcodes
  - What instruction formats we support
    - e.g. R-type, I-type
    - lay out & # of fields and bits per field
  - Specification detailed enough such that someone could write an assembler for it
    - assembler: translates assembly to machine code
  - registers
    - how many the ISA requires
    - how many are general purpose, how many are specialized
  - all internal data paths and storage will be 8 bits wide
  - addressing modes our ISA supports
    - proposition:
      - support direct addressing only
      - Pro: simple ISA
      - Con: ours is on programmer to calculate address each access (no shortcuts like indexed addressing)
        - this is especially justifiable because our processor only needs to support 3 programs, meaning the number of extra lines of assembly is O(1) (as opposed to O(n) in # of programs written as it would be for a general purpose processor)

- Instruction Formats:
  - We will support two instruction formats:
    - Mirroring MIPS (except we have no need to jmp so no j-type)

  - R-type (register type)
    - for operations on 2 registers

Op (4) (op-code)	Rd (2) (source1/dest)	Rs (2) (source2)	(1) (unused)

    - e.g. add edx, eax ⟷ 1011 10 00 (add edx eax)
      ⇒ edx = edx + eax

  - I-type (immediate type)
    - operate on a registers and an immediate value

Op (4) (op-code)	Rs (2) (source1/dest)	Immediate (3) (immediate value)

    - e.g. ldi eax, 0 ⟷ 0100 00 000 (ldi eax 0)
      ⇒ eax = 0

Note: max value representable in 3 bits is 7. We achieve values larger than this by repeatedly operating on values stored in 8-bit registers until they contain our desired value.

- Register Names → Machine Code Identifier Map:

eax : 00	general-purpose
ebx : 01	general-purpose
edx : 10	general-purpose
PC : 11	special-purpose (program counter)

- Operations Table:

opcode	format	operation	assembly abbrev.	example	machine code		
0000	I	left shift	ls	ls eax, 2	0000	00	010
0001	I	right shift	rs	rs eax, 2	0001	00	010
0010	R	AND	and	and eax, ebx	0010	00	01 X
0011	R	OR	or	or eax, ebx	0011	00	01 X
0100	I	load immediate	ldi	ldi eax, 7	0100	00	111
0101	R	load register	ldr	ldr edx, (eax)	0101	10	00 X
0110	R	store register	str	str (edx), eax	0110	10	00 X
0111	R	branch if equal	beq	beq eax, ebx	0111	00	01 X
1000	R	greater than or equal to	geq	geq eax, ebx	1000	00	01 X
1001	R	equal to	eq	eq eax, ebx	1001	00	01 X
1010	R	negate to 2's complement	neg	neg ebx, ebx	1010	01	01 X
1011	R	add	add	add eax, ebx	1011	00	01 X
1100	X	add immediate	addi	addi eax, 3	1100	00	011
1101	X	nop					
1110	X	nop					
1111	X	nop					

- Syntax / philosophy:
  - R-type instructions
    - Syntax
      - <op> <src1/dest>, <src2>
      - e.g.
        add eax, ebx → eax = eax + ebx

- Philosophy
  - use one of the two registers as both a source and the destination
  - allows more efficient use of space per instruction because we only need 4-bits to specify the registers we will use during execution
  - if you want to save both of the values in the operation (including the destination one that will be overwritten), then save the value in the other register first
    - e.g.

```
ldr ecx, eax  //ecx = eax
add eax, ebx  //eax = eax + ebx
```

- I-type Syntax
  - Syntax
    - <op> <src1/dest>, <immediate value>
      - e.g.

```
ldi eax, 0 ⟶ eax = 0
```

  - Philosophy
    - our instruction width and op-code/register widths result in 3 bits remaining to represent immediate values. This is enough to represent any 8-bit number the system may use as exemplified by the following problem:
      - say we want to access the highest memory byte in our system, byte 255. we are tempted to say

```
ldi eax, (255),
```

but this is an invalid instruction because the maximum available immediate value is 7. So we build the number:

```
ldi  eax, 7   // eax = 00000111
ls   eax, 3   // eax = eax<<3 = 00 111000
addi eax, 7   // eax = eax + 7 = 00111111
ls   eax, 2   // eax = eax<<2 = 11111100
addi eax, 3   // eax = eax+3 = 11111111 = (255)10
```

and now we can load the memory at byte 255 with:

```
ldr edx, (eax) // edx = (value stored in memory address eax = 255)
```