Control Logic II

CSCI 2050U - Computer Architecture

Randy J. Fortier
@randy_fortier

OntarioTech UNIVERSITY

Outline

- Decode
- Execute
  - Register transfer language
  - Example program execution

Decode

CSCI 2050U - Computer Architecture

OntarioTech UNIVERSITY

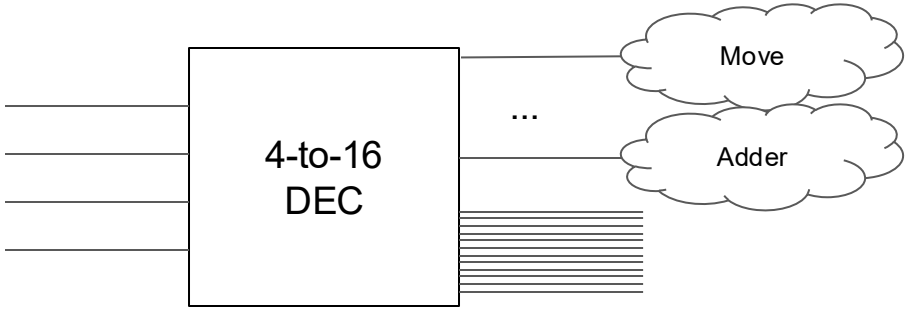
Decoding Instructions

- Decoding involves the following:
  - Determine which (e.g. arithmetic) operation is to be performed
  - Determine which registers (or memory, constants) are to be used for the operands

Decoding Instructions

- Decoding involves the following:
  - Determine which (e.g. arithmetic) operation is to be performed
    - The first 4 bits of our instruction are the opcode
    - These are fed into a 4-to-16 line decoder
    - The outputs of this decoder will be fed into enable inputs for the ALU circuits
  - Determine which registers (or memory, constants) are to be used for the operands

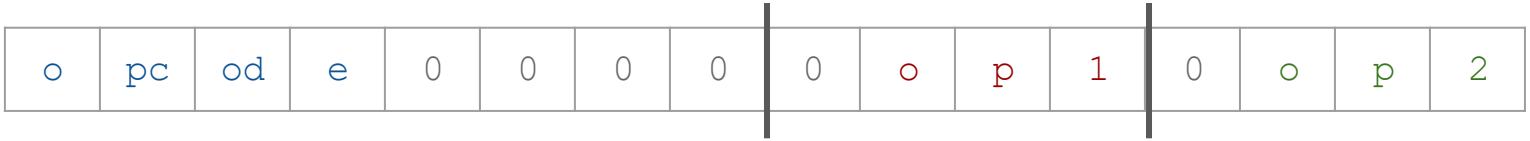
Opcode	Assembly
0001	MOV
0100	ADD
0101	SUB
1000	JMP



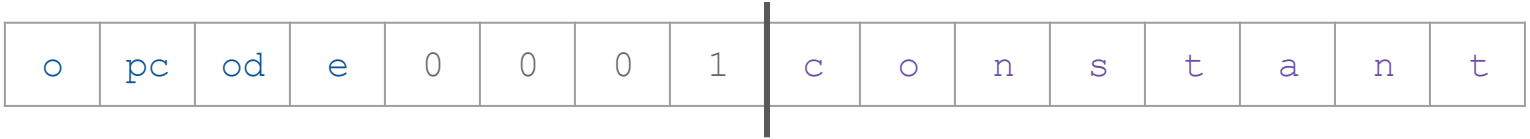
Decoding Instructions

0001	A
0010	B
0011	C
0100	D
0101	E
0110	F
0111	G

- Decoding involves the following:
  - Determine which (e.g. arithmetic) operation is to be performed
  - Determine which registers (or memory, constants) are to be used for the operands
    - The first 4 bits of the operand identify the first operand
      - `0001` - register A
      - `0010` - register B (etc.)
    - The next 4 bits of the operand are for the second operand
    - These will activate the input and output enables for the registers
      - This might also use a decoder



- The operand could also be a single constant



Execute

CSCI 2050U - Computer Architecture

OntarioTech UNIVERSITY

Register Transfer Language (RTL)

- Register transfer language
  - A language used to describe microoperations
    - Operations that are part of the instruction
  - So called because most things involve transferring data to/from registers

HAX Instruction Set: RTL

`MOV A, B`	`A ← B`
	`PC ← PC + 2`
	`IR ← M[PC]`

`ADD A, B`	`A ← A + B`
	`PC ← PC + 2`
	`IR ← M[PC]`

`SUB A, B`	`A ← A - B`
	`PC ← PC + 2`
	`IR ← M[PC]`

HAX Instruction Set: RTL

LOAD B	MAR ← B MBR ← M[MAR] A ← MBR PC ← PC + 2 IR ← M[PC]
STORE B	MAR ← B MBR ← A M[MAR] ← MBR PC ← PC + 2 IR ← M[PC]

HAX Instruction Set: RTL

`JMP A`	`PC ← A` `IR ← M[PC]`

`JZ A`	`IF FLAGS[Z] == 1 THEN`
	`  PC ← A`
	`ELSE`
	`  PC ← PC + 2`
	`IR ← M[PC]`

`JNZ A`	`IF FLAGS[Z] == 0 THEN`
	`  PC ← A`
	`ELSE`
	`  PC ← PC + 2`
	`IR ← M[PC]`

HAX Instruction Set

Binary	Mnemonic	Operand(s)	Description
`0000`	`HALT`	-	Stops execution
`0001`	`MOV`	`dest, source`	Copies the value from `source` to `dest`
`0010`	`LOAD`	`addr`	Loads value from memory at `addr` into A
`0011`	`STORE`	`addr`	Stores value from A into memory at `addr`
`0100`	`ADD`	`dest, source`	`dest = dest + source`
`0101`	`SUB`	`dest, source`	`dest = dest - source`
...			
`1000`	`JMP`	`addr`	Jumps to `addr`
`1001`	`JZ`	`addr`	If the `zero` flag is set, jumps to `addr`
`1010`	`JNZ`	`addr`	If the `zero` flag is cleared, jumps to `addr`
...			

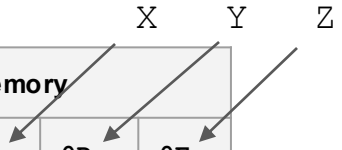
A Test Program

- Let's write a HAX assembly language program:

```
00: LOAD 0C                 ; LOAD X
02: MOV B, A
04: LOAD 0D                 ; LOAD Y
06: ADD A, B
08: STORE 0E                ; STORE Z
0A: HALT
0C: 0x04                        ; X: 4
0D: 0xFE                        ; Y: -2
0E: 0x00                        ; Z: 0
```

A Test Program

- Let's trace our HAX assembly language program:



Instr	RTL	Registers						Memory		
		A	B	PC	IR	MAR	MBR	0C	0D	0E
Initial	-	??	??	00	21	??	??	04	FE	00
LOAD 0C	MAR ← 0C MBR ← M[MAR] A ← MBR PC ← PC + 2 IR ← M[PC]	04	??	02	10	0C	04	04	FE	00
MOV B, A	B ← A PC ← PC + 2 IR ← M[PC]	04	04	04	21	0C	04	04	FE	00

A Test Program

- Let's trace our HAX assembly language program:

Instr	RTL	Registers						Memory		
		A	B	PC	IR	MAR	MBR	0C	0D	0E
LOAD 0D	MAR ← 0D MBR ← M[MAR] A ← MBR PC ← PC + 2 IR ← M[PC]	FE	04	06	40	0D	FE	04	FE	00
ADD A, B	A ← A + B PC ← PC + 2 IR ← M[PC]	02	04	08	31	0D	FE	04	FE	00
STORE 0E	MAR ← 0E MBR ← A M[MAR] ← MBR PC ← PC + 2 IR ← M[PC]	02	04	0A	00	0E	02	04	FE	02

Assembly

- Assembly is the process of converting human-readable assembly language (with mnemonics) to machine-readable machine language (in binary)
  - Disassembly is just the opposite
  - For simplicity, I will use hexadecimal instead of binary

Assembly

- Below shows the assembly of our test program:

```
00:  LOAD 0C
02:  MOV B, A
(B:0010 A:0001, 00100001, 33, 0x21)
04:  LOAD 0D
06:  ADD A, B
(A:0001 B:0010, 00010010, 18, 0x12)
08:  STORE 0E
0A:  HALT
     0000
0C:  0x04                          ; 4
     04FE
0D:  0xFE                          ; -2
0E:  0x00                          ; 0
     00??
```

```
00:    210C
02:    1021

04:    210D
06:    4012

08:    310E
0A:
0C:
0E:
```

0001	A
0010	B
0011	C
0100	D
0101	E
0110	F
0111	G

Two-stage Assembly

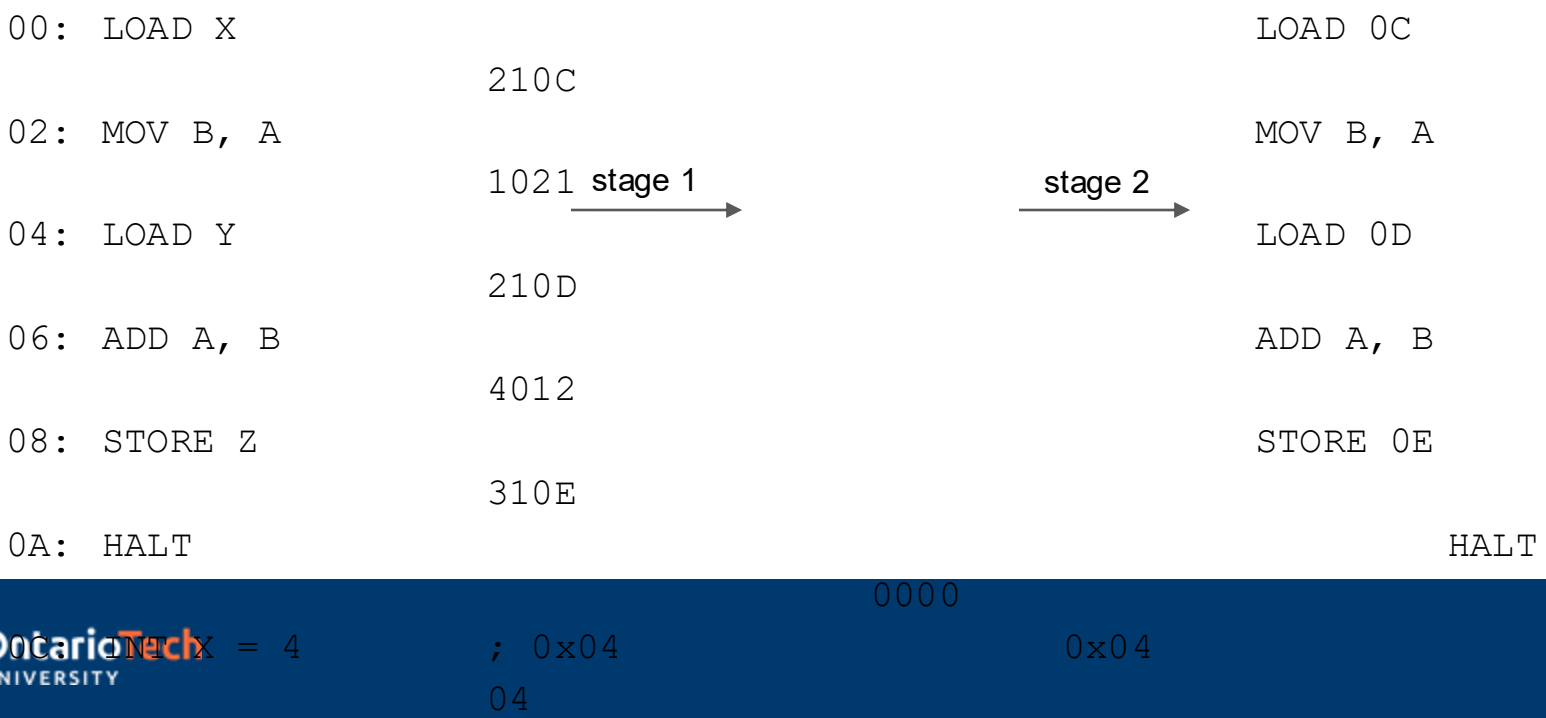
- Two-stage assembly can make our program more readable
  - Use named locations for data (i.e. variables)
  - Use named locations for jumps (i.e. labels)

- The stages:

1. Replace any symbols with the address of the symbol
2. Replace mnemonics and operands with binary equivalents (as before)

Two-stage Assembly

- Below shows the assembly of our test program:



A More Complex Program

- Here is a program that loops over an array of numbers, adding pairs of them:

```
                00:     MOV D,
          increment
                02:     LOAD D
                04:     MOV D, A
                06:     MOV E,
          count
                08:     LOAD E
                0A:     MOV E, A
                0C:     MOV C,
          numbers
loopStart:      0E:     LOAD C
                10:     MOV B, A
                12:     ADD C, D
                14:     LOAD C
                16:     ADD A, B
                18:     ADD C, D
                1A:     STORE C
                1C:     ADD C, D
                1E:     SUB E, D
                20:     JNZ
           loopStart
                22:     HALT
increment:      24:     INT 1
count:                  25:
     INT 2
numbers:                26:
     INT 4
                27:     INT -2
                28:     INT 0
                        29:
     INT -6
                2A:     INT 4
```

A More Complex Program

- Implementing these aliases in our assembler could make our program easier to understand:

```
                00:     MOV D,
          increment
                02:     LOAD D
                04:     MOV D, A
                06:     MOV E,
          count
                08:     LOAD E
                0A:     MOV E, A
                0C:     MOV C,
          numbers
loopStart:      0E:     LOAD C
                10:     MOV B, A
                12:     ADD C, D
                14:     LOAD C
                16:     ADD A, B
```

```
                1A:     STORE C
                1C:     ADD C, D
                1E:     CMP E, D
                20:     JNE
           loopStart
                22:     HALT
increment:      24:     INT 1
count:                  25:
     INT 2
numbers:                26:
     INT 4
                27:     INT -2
                28:     INT 0
                        29:
     INT -6
                2A:     INT 4
```

Wrap-Up

- Decode
- Execute
  - Register transfer language
  - Example program execution

What is next?

- Assembly language programming
  - Development tools
  - Registers
  - A basic program