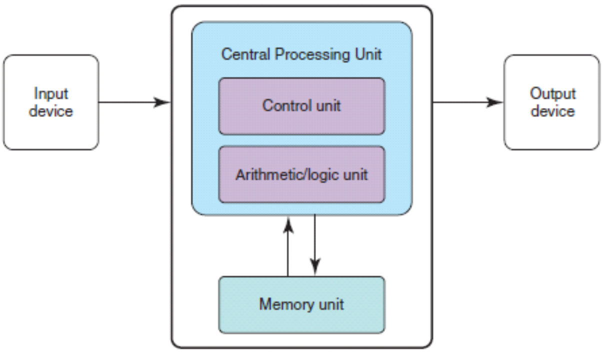
CSCU9A2
The Brookshear Machine

- A simple, hypothetical computer
- Invented by J Glenn Brookshear for his series of introductory text books on Computer Science
  - Now in the 11th edition (library has 10th and 11th)
- To illustrate how typical high level language constructs are implemented
  - Cannot handle complex language features without extension
  - Cannot handle input/output without extension
- Some implementations are available
  - All awkward in some way
  - Some with extensions, including input/output
- First a reminder from CSCU9A1…

CSCU9A1: Basic architecture of a computer

- Memory unit contains both *program and data* for running programs
- CPU interprets program and executes instructions using ALU
- This concept is called a "von Neumann computer"
- Input/output: keyboard/screen/network/disks/…



CSCU9A1: Memory (RAM)

Memory :

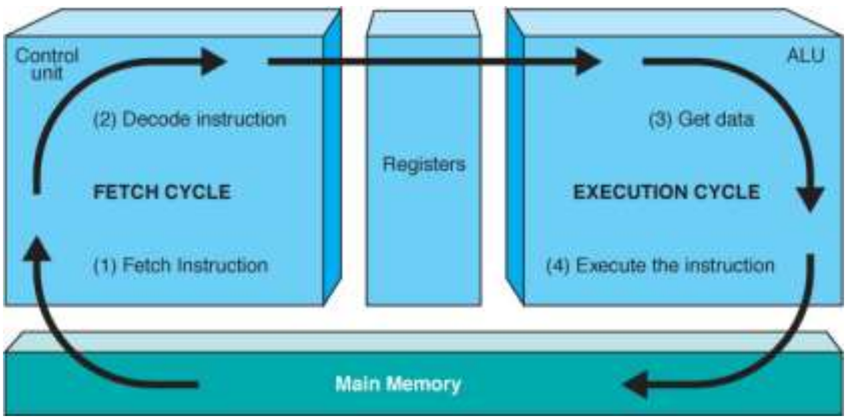
- A collection of cells, each with a unique *physical address*
- Each cell is made up of a number of bits

  8, 16 24, 32, 64

  binary digits (bits)
- Addresses start at 0, and are usually contiguous
- Both addresses and contents are in binary

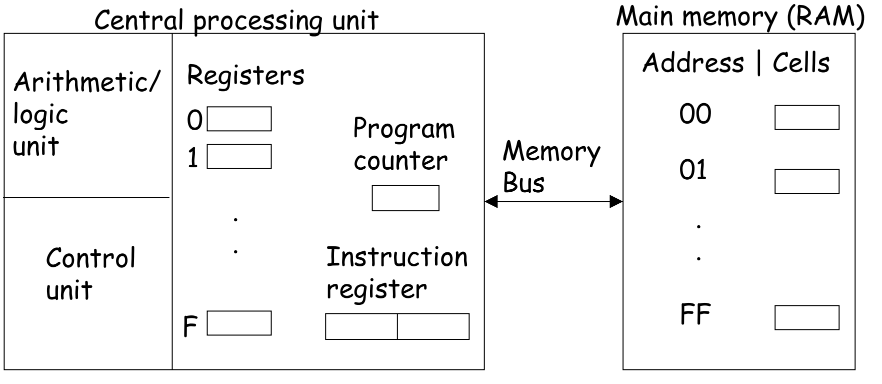
Address	Contents
00000000	11100011
00000001	10101001
⋮	⋮
11111100	00000000
11111101	11111111
11111110	10101010
11111111	00110011

CSCU9A1: The Fetch-Execute Cycle



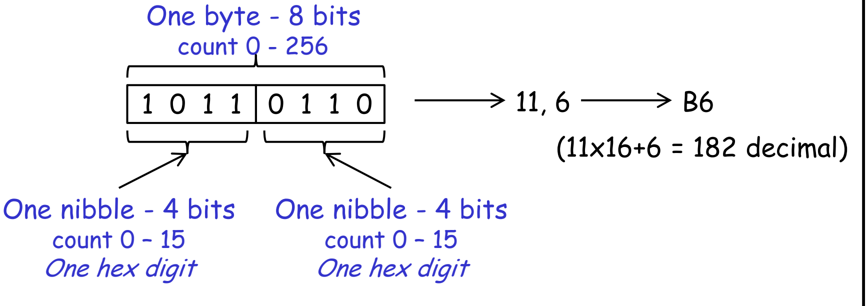
The Brookshear Machine architecture

- Memory: 256 bytes – addresses 00 – FF (hexadecimal)
- CPU: 16 one byte registers – numbered 0 – F (hexadecimal)
- PC: one byte register;   IR: two byte register



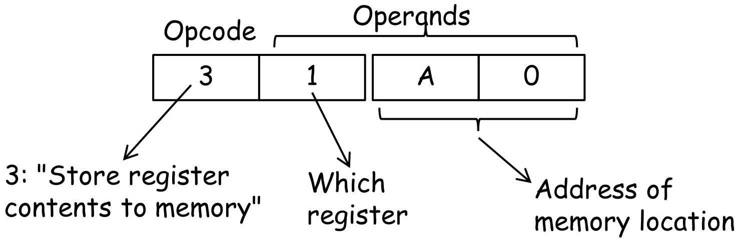
A note about hexadecimal

- It is useful to have a way to represent the numbers/bit patterns in computer memory that corresponds neatly with byte structure
- Hexadecimal is perfect (base 16)
  - 0 – 15 counted as 0 1 2 3 4 5 6 7 8 9 A B C D E F
- One byte is 8 bits, or two 4 bit *nibbles:*



Brookshear instruction format

- All machine instructions occupy *two bytes* of memory
  - So, the PC has 2 added after each fetch
  - Real CPUs have instructions of different lengths
- Each nibble has a different role:



- The operand nibbles are used differently with each instruction (opcode)

And now... The Brookshear Instruction set

In the original Brookshear machine there are 13 different instructions, with opcodes 0 – C (fits neatly in one nibble):

0iii - No-operation

1RXY - Load register **R** with contents of memory location **XY**

2RXY - Load register **R** with value **XY**

3RXY - Store contents of register **R** at location **XY**

4iRS - Move contents of register **R** to register **S**

5RST - Add contents of registers **S** and **T** as binary numbers, place result in register **R**

6RST - Add contents of registers **S** and **T** as floating-point numbers, place result in register **R**

R,S,T - Register numbers

XY - A one-byte address or data value **Z** - A half-byte value

i - Ignored when the instruction is de-coded: usually entered as **0**

7RST - **OR** together the contents of registers **S** and **T**, place result in register **R**

8RST - **AND** together the contents of registers **S** and **T** , place result in register **R**

9RST - **XOR** together the contents of registers **S** and **T** , place result in register **R**

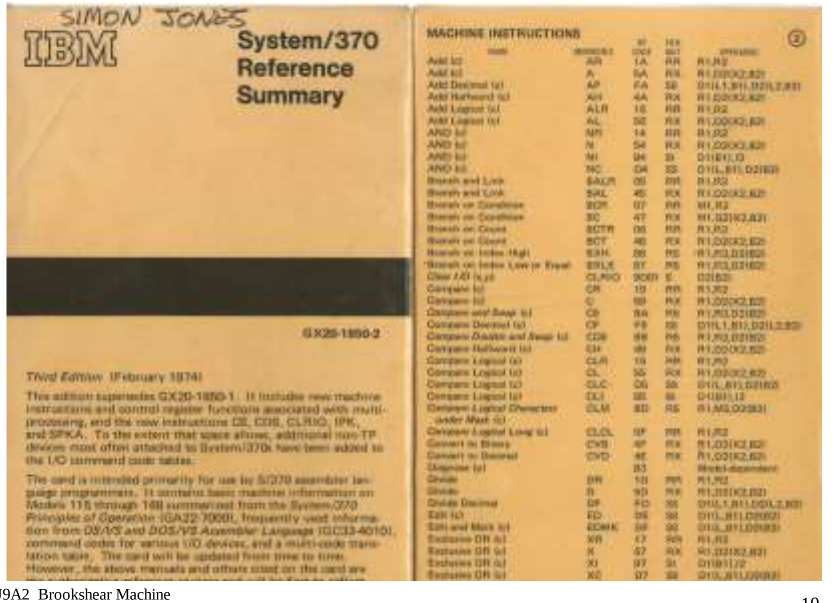
ARiZ - Rotate the contents of register **R** one bit to the right, **Z** times

BRXY - Jump to instruction at **XY** if contents of register **R** equal contents of register **0**

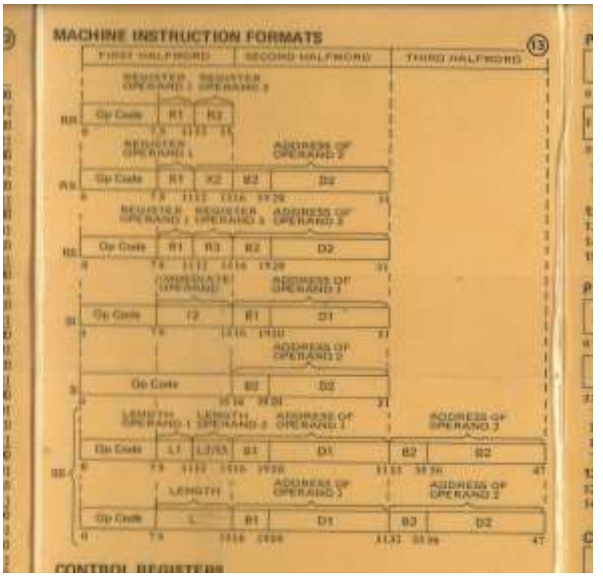
Ciii - Halt

- Apparent omissions:
  - Subtraction, multiplication, division, < comparison, indirect/indexed addressing
  - All can be done with sufficient cleverness, but real CPUs have them built-in

Programmer's reference card for IBM 370 mainframe



- The instruction formats:



A simple Brookshear program

"Add one to the contents of location 80" ("n = n+1;")

We choose to place the program starting at address 20 in memory, so we have the "memory map":

Address Comments

```
20 1A80     ; Load RA with contents of addr 80
22 2B01     ; Load RB with the number 01
24 5CAB     ; Add Binary RA+RB -> RC
26 3C80     ; Store RC at addr 80
28 C000     ; Halt
```

Instructions

```
...
80 24       ; Initial contents of location 80
```

We "launch" the program by loading 20 into the PC and activating the fetch-execute cycle

The executable program could be held in a file (a .exe) containing the following 11 bytes:

201A802B015CAB3C80C000

Assembly language

- It would be painful programming in hexadecimal, so *"assembly language"* was invented
  - A more human readable form for machine instructions
  - Translated into hexadecimal by an *assembler* program
- For example:
  - Instead of
    1rxy (Load register **r** with contents of location **xy**)
  - We write
    MOV [xy] -> Rr
  - **[xy]** indicates that **xy** is a memory address, plain **xy** is a value, and **Rr** indicates register number **r**
  - There will be several **MOV** instructions, corresponding to the Load and Store opcodes

Brookshear Assembly instructions

- 0iii NOP
- 1rxy MOV [xy] -> Rr
- 2rxy MOV xy -> Rr
- 3rxy MOV Rr -> [xy]
- 4irs MOV Rr -> Rs
- 5rst ADDI Rs, Rt -> Rr
- 6rst ADDF Rs, Rt -> Rr
- 7rst OR Rs, Rt -> Rr
- 8rst AND Rs, Rt -> Rr
- 9rst XOR Rs, Rt -> Rr
- Ariz ROT Rr, z
- Brxy JMPEQ xy, Rr
- Ciii HALT

The "add one" example in assembly language

- So, the "add one to location 80" program becomes:

```
20 MOV [80] -> RA   ; Load RA with contents of 80
22 MOV 01 -> RB     ; Load RB with 01
24 ADDI RA, RB -> RC ; Add Binary RA+RB -> RC
26 MOV RC -> [80]   ; Store RC at 80
28 HALT
```

- The final executable code is *identical*
- Real assembly languages contain many sophisticated features to simplify the assembly language programmer's task

In subsequent lectures we will look at how typical Java fragments may be compiled into Brookshear machine language

End of section