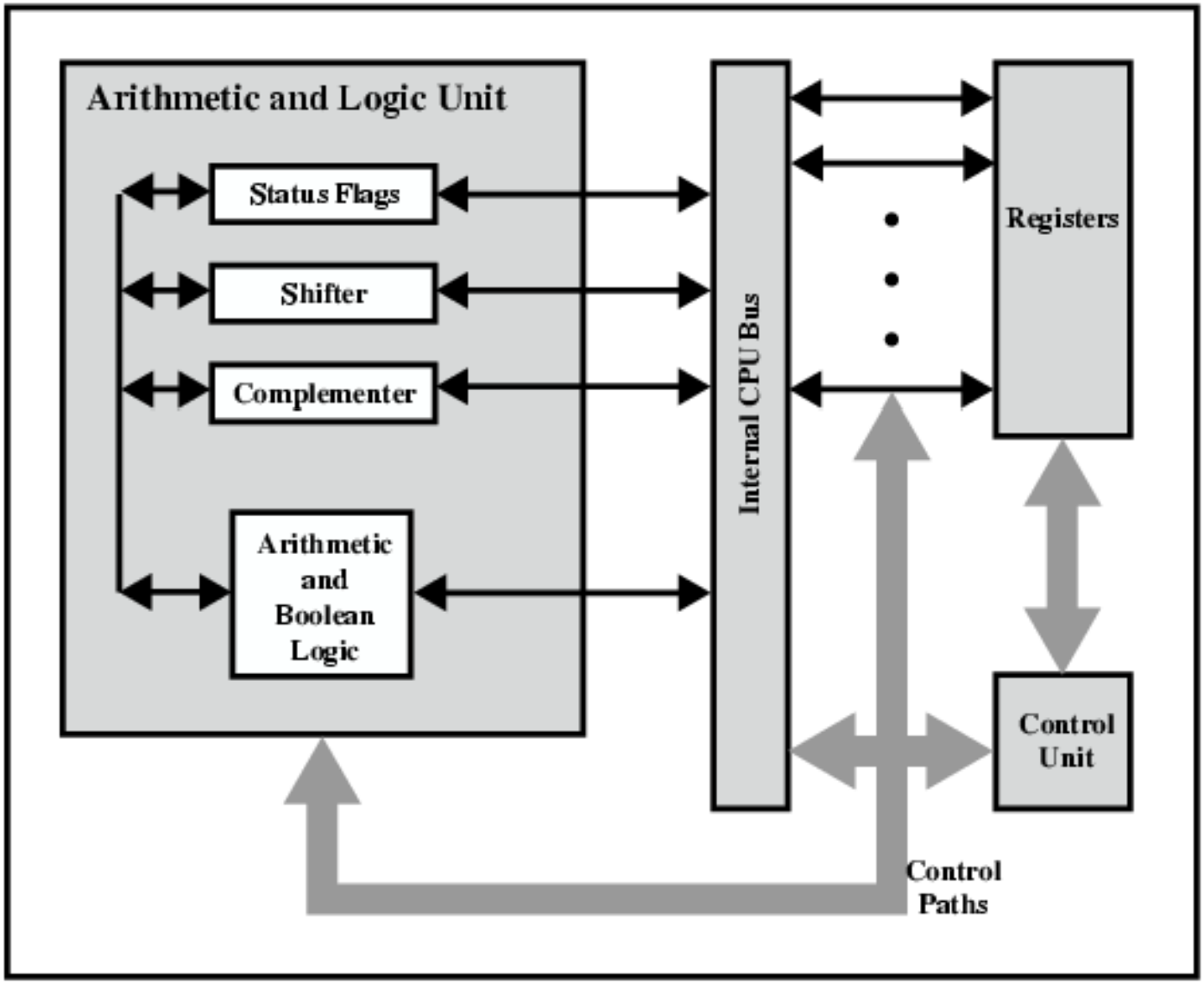
Instruction Sets:
Characteristics and Functions

CPU Internal Structure



Registers

- CPU must have some working space (temporary storage)
- Called registers
- Number and function vary between processor designs
- One of the major design decisions
- Top level of memory hierarchy

User Visible Registers

- General Purpose
- Data
- Address
- Condition Codes

How Many GP Registers?

- Between 8 - 32
- Fewer = more memory references
- More does not reduce memory references and takes up processor space
- Large enough to hold full address
- Large enough to hold full word
- Often possible to combine two data registers
  - C programming
  - double int a;
  - long int a;

Control & Status Registers

- Program Counter
- Instruction Decoding Register
- Memory Address Register
- Memory Buffer Register

Condition Code Registers

- Sets of individual bits
  - —e.g. result of last operation was zero
- Can be read (implicitly) by programs
  - —e.g. Jump if zero
- Can not (usually) be set by programs
- Needs for conditional instructions

Program Status Word

- A set of bits
- Includes Condition Codes
- Sign of last result
- Zero
- Carry
- Equal
- Overflow
- Interrupt enable/disable
- Supervisor

Function of Control Unit

- For each operation a unique code is provided
  - —e.g. ADD, MOVE
- A hardware segment accepts the code and issues the control signals

What is an Instruction Set?

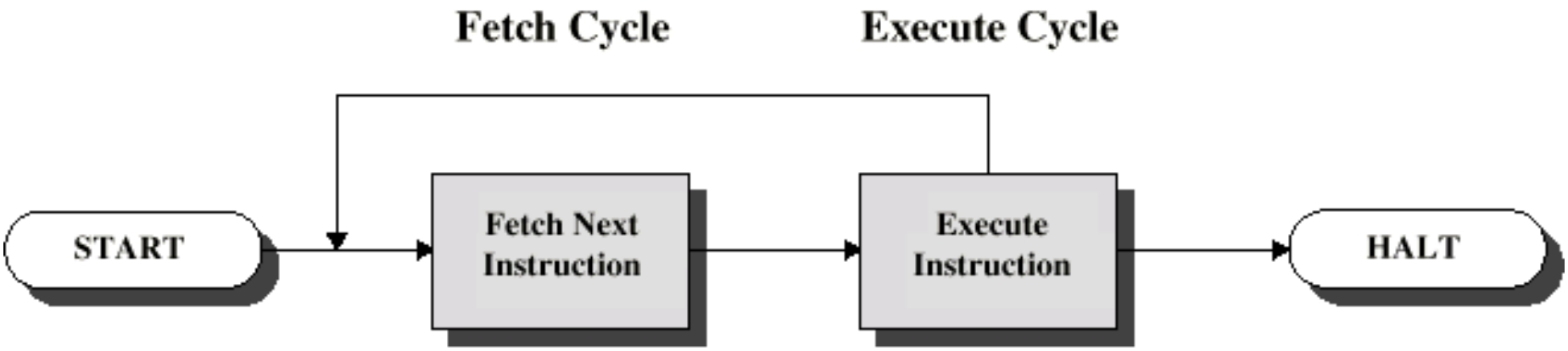
- The complete collection of instructions that are understood by a CPU
- Machine Code
  - Binary
- Usually represented by assembly codes

Elements of an Instruction

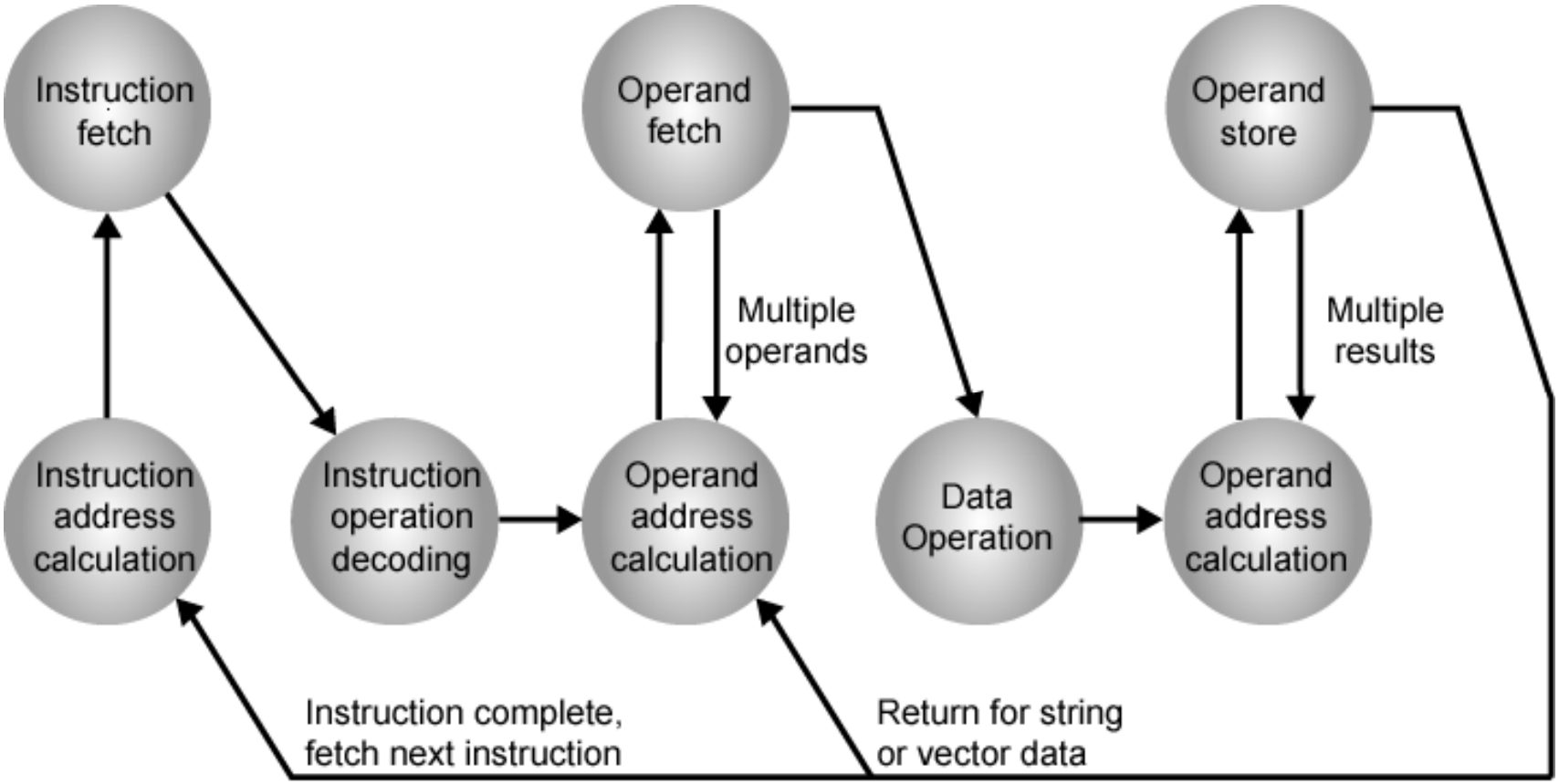
- Operation code (Op code)
  - —Do this
- Source Operand reference
  - —To this
- Result Operand reference
  - —Put the answer here
- Next Instruction Reference
  - —When you have done that, do this...

Instruction Cycle

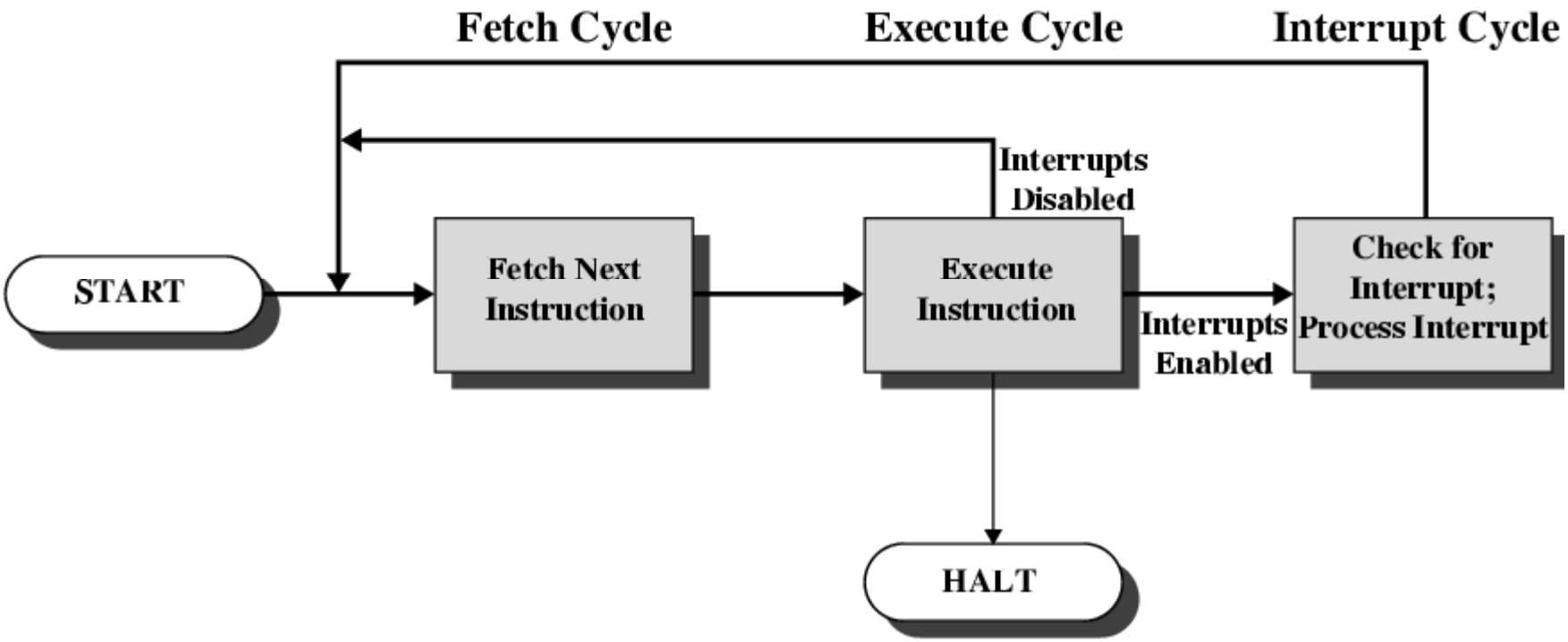
- Two steps:
  - Fetch
  - Execute



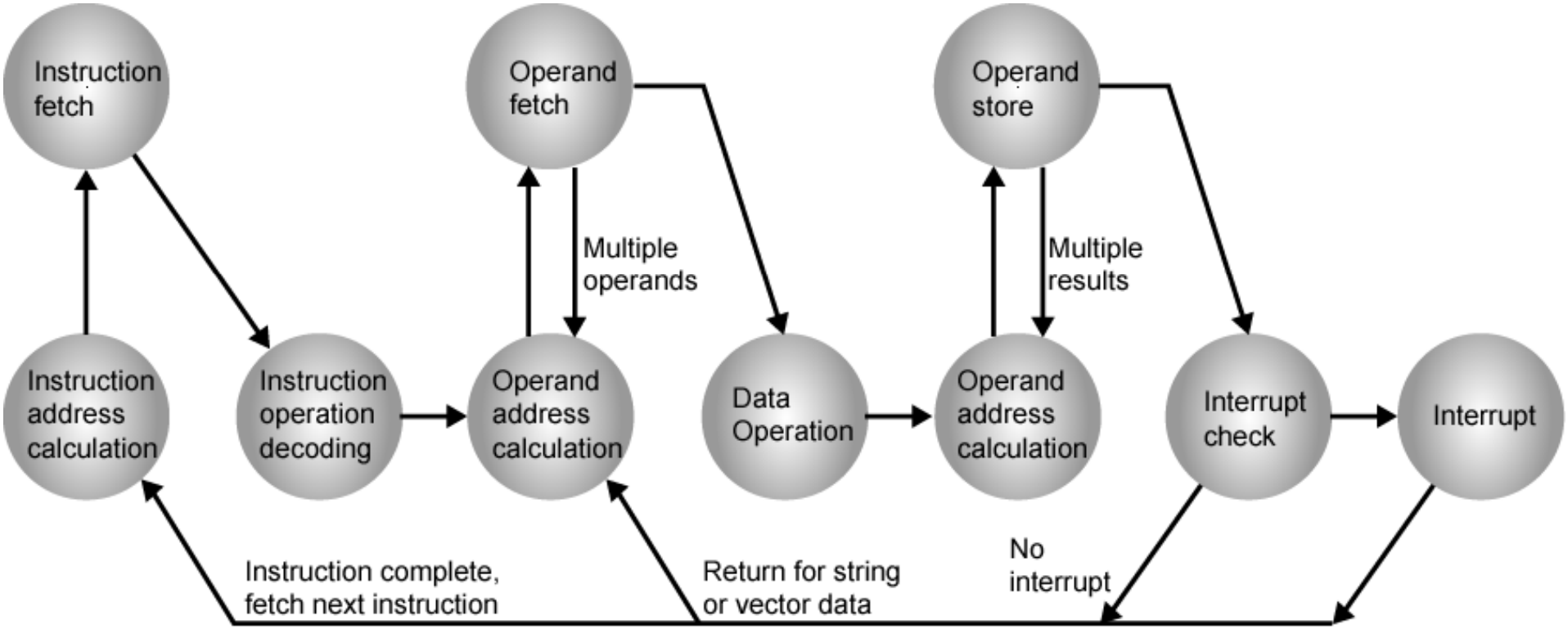
Instruction Cycle State Diagram



Instruction Cycle with Interrupts



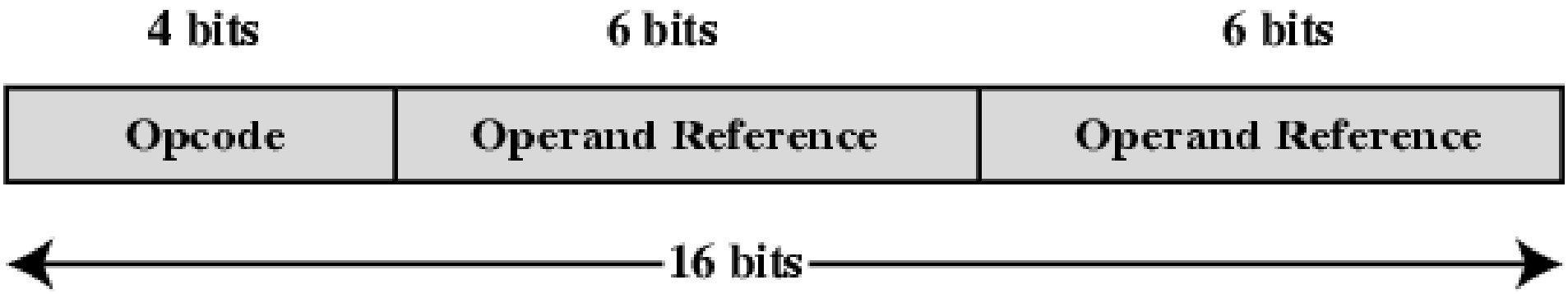
Instruction Cycle (with Interrupts) - State Diagram



Instruction Representation

- In machine code each instruction has a unique bit pattern
- For human consumption (well, programmers anyway) a symbolic representation is used
  - e.g. ADD, SUB, LOAD
- Operands can also be represented in this way
  - ADD A,B

Simple Instruction Format



Instruction Types

- Data processing
- Data storage (main memory)
- Data movement (I/O)
- Program flow control

Number of Addresses (a)

- 3 addresses
  - —Operand 1, Operand 2, Result
  - —a = b + c;
  - —May be a forth - next instruction (usually implicit)
  - —Not common
  - —Needs very long words to hold everything

Number of Addresses (b)

- 2 addresses
  - One address doubles as operand and result
  - a = a + b
  - Reduces length of instruction
  - Requires some extra work
    - Temporary storage to hold some results

Number of Addresses (c)

- 1 address
  - —Implicit second address
  - —Usually a register (accumulator)
  - —Common on early machines

Number of Addresses (d)

Instruction		Comment
SUB	Y, A, B	Y ← A − B
MPY	T, D, E	T ← D × E
ADD	T, T, C	T ← T + C
DIV	Y, Y, T	Y ← Y ÷ T

(a) Three-address instructions

Instruction		Comment
MOVE	Y, A	Y ← A
SUB	Y, B	Y ← Y − B
MOVE	T, D	T ← D
MPY	T, E	T ← T × E
ADD	T, C	T ← T + C
DIV	Y, T	Y ← Y ÷ T

(b) Two-address instructions

Instruction		Comment
LOAD	D	AC ← D
MPY	E	AC ← AC × E
ADD	C	AC ← AC + C
STOR	Y	Y ← AC
LOAD	A	AC ← A
SUB	B	AC ← AC − B
DIV	Y	AC ← AC ÷ Y
STOR	Y	Y ← AC

(c) One-address instructions

Figure 10.3 Programs to Execute $Y = \frac{A - B}{C + (D \times E)}$

Number of Addresses (e)

- 0 (zero) addresses
  - —All addresses implicit
  - —Uses a stack
  - —e.g. push a
  - — push b
  - — add
  - — pop c

  - —c = a + b

How Many Addresses

- More addresses
  - More complex instructions
  - More registers
    - Inter-register operations are quicker
  - Fewer instructions per program
- Fewer addresses
  - Less complex instructions
  - More instructions per program
  - Faster fetch/execution of instructions

Design Decisions (1)

- Operation repertoire
  - How many ops?
  - What can they do?
  - How complex are they?
- Data types
- Instruction formats
  - Length of op code field
  - Number of addresses

Design Decisions (2)

- Registers
  - Number of CPU registers available
  - Which operations can be performed on which registers?
- Addressing modes

Types of Operand

- Addresses
- Numbers
  - —Integer/floating point
- Characters
  - —ASCII etc.
- Logical Data
  - —Bits or flags

Specific Data Types

- General - arbitrary binary contents
- Integer - single binary value
- Ordinal - unsigned integer
- Unpacked BCD - One digit per byte
- Packed BCD - 2 BCD digits per byte
- Near Pointer - 32 bit offset within segment
- Bit field
- Byte String
- Floating Point

Types of Operation

- Data Transfer
- Arithmetic
- Logical
- Conversion
- I/O
- System Control
- Transfer of Control

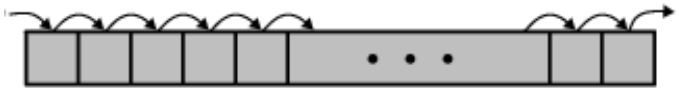
Data Transfer

- Specify
  - Source
  - Destination
  - Amount of data
- May be different instructions for different movements
  - e.g. IBM 370
- Or one instruction and different addresses
  - e.g. VAX

Arithmetic

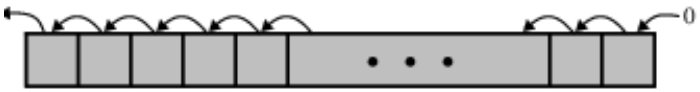
- Add, Subtract, Multiply, Divide
- Signed Integer
- Floating point
- May include
  - Increment (a++)
  - Decrement (a--)
  - Negate (-a)

Shift and Rotate Operations



(a) Logical right shift

SHR

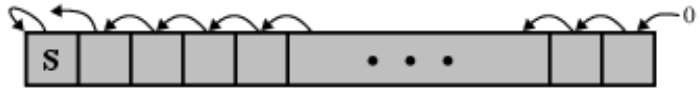


(b) Logical left shift

SHL

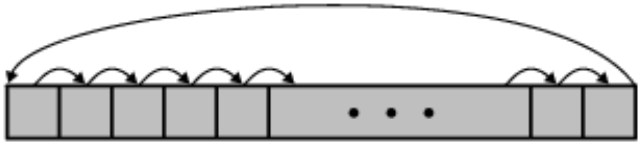
(c) Arithmetic right shift

ASR



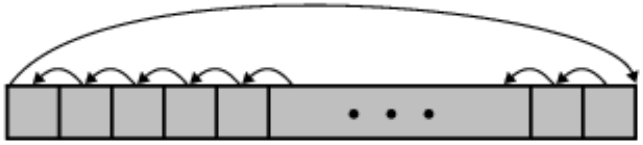
(d) Arithmetic left shift

ASL



(e) Right rotate

ROR



(f) Left rotate

ROL

Logical

- Bitwise operations
- AND, OR, NOT

Input/Output

- May be specific instructions
- May be done using data movement instructions (memory mapped)
- May be done by a separate controller (DMA)

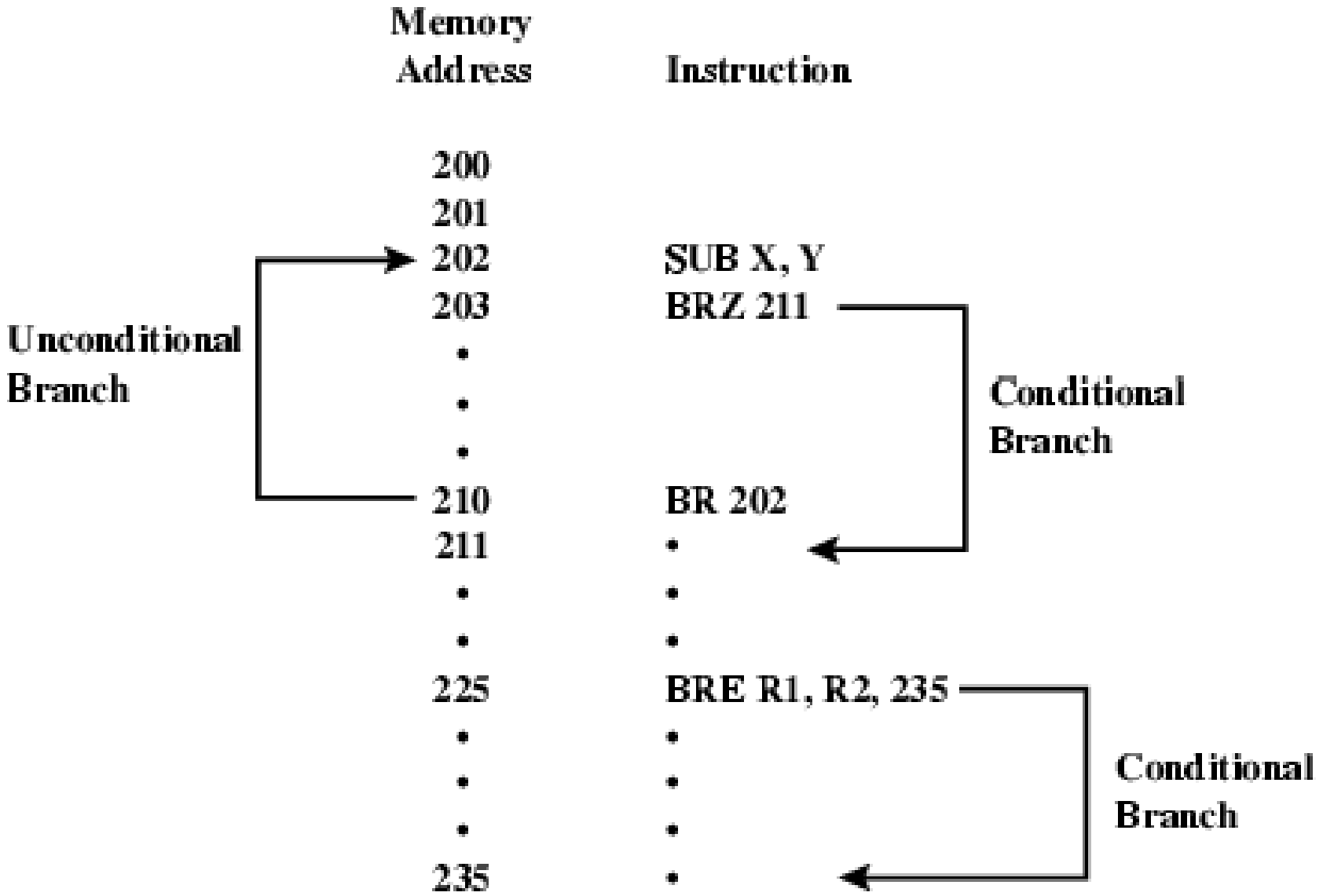
Systems Control

- Privileged instructions
- CPU needs to be in specific state
- For operating systems use

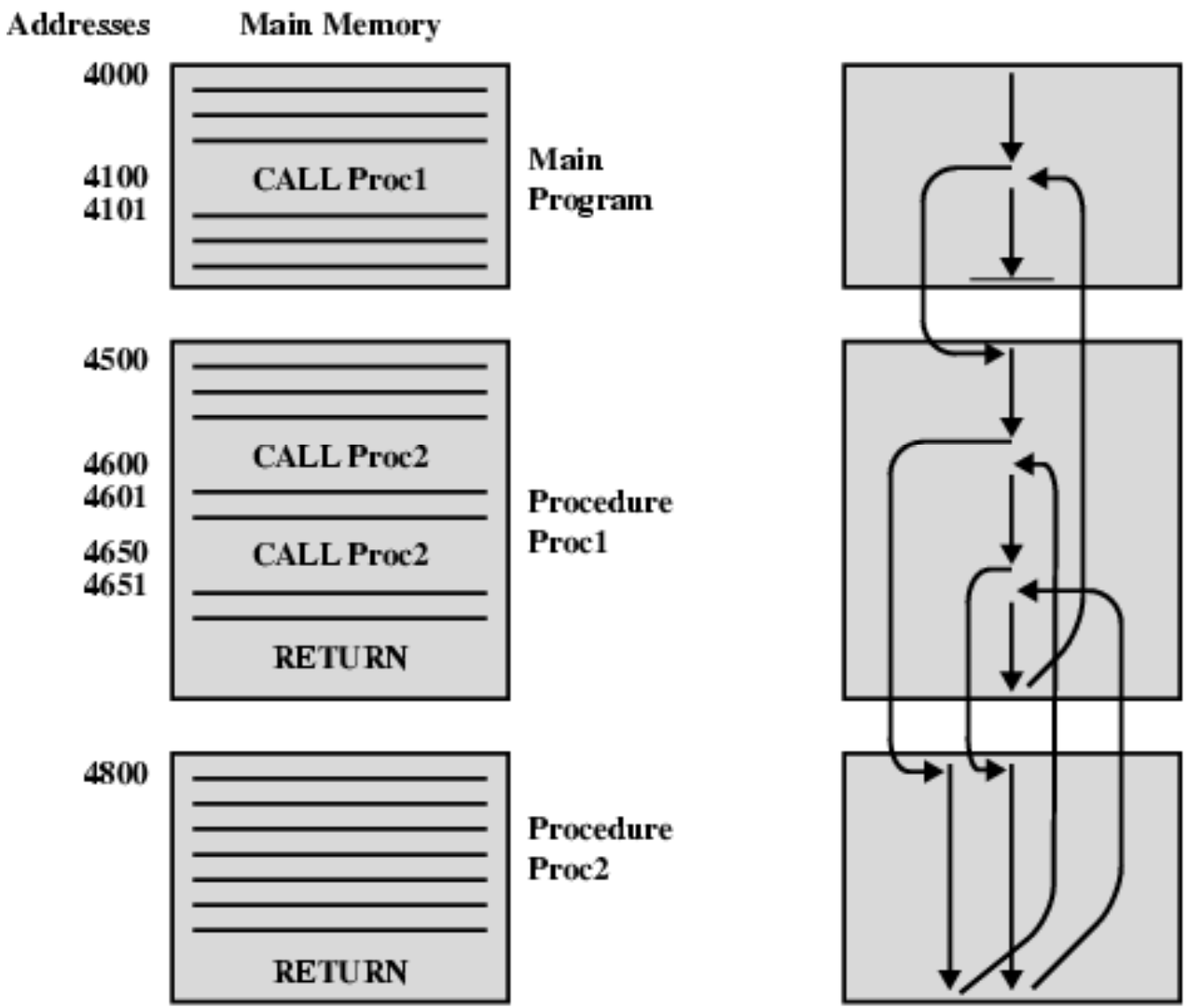
Transfer of Control

- Branch
  - e.g. branch to x if result is zero
- Skip
  - e.g. increment and skip if zero
  - ISZ Register1
  - Branch xxxx
  - ADD A
- Subroutine call
  - c.f. interrupt call

Branch Instruction



Nested Procedure Calls



Byte Order

- What order do we read numbers that occupy more than one byte
- e.g. (numbers in hex to make it easy to read)
- 12345678 can be stored in 4x8bit locations as follows

Byte Order (example)

Address	Value (1)	Value(2)
184	12	78
185	34	56
186	56	34
187	78	12

- i.e. read top down or bottom up?

Byte Order Names

- The problem is called Endian
- The system on the right has the least significant byte in the lowest address
  - This is called little-endian
- The system on the left has the least significant byte in the highest address
  - This is called big-endian

Example of C Data Structure

```
struct{
   int     a;     //0x1112_1314                    word
   int     pad;   //
   double  b;     //0x2122_2324_2526_2728          doubleword
   char*   c;     //0x3132_3334                    word
   char    d[7];  //'A','B','C','D','E','F','G'   byte array
   short   e;     //0x5152                         halfword
   int     f;     //0x6161_6364                    word
} s;
```

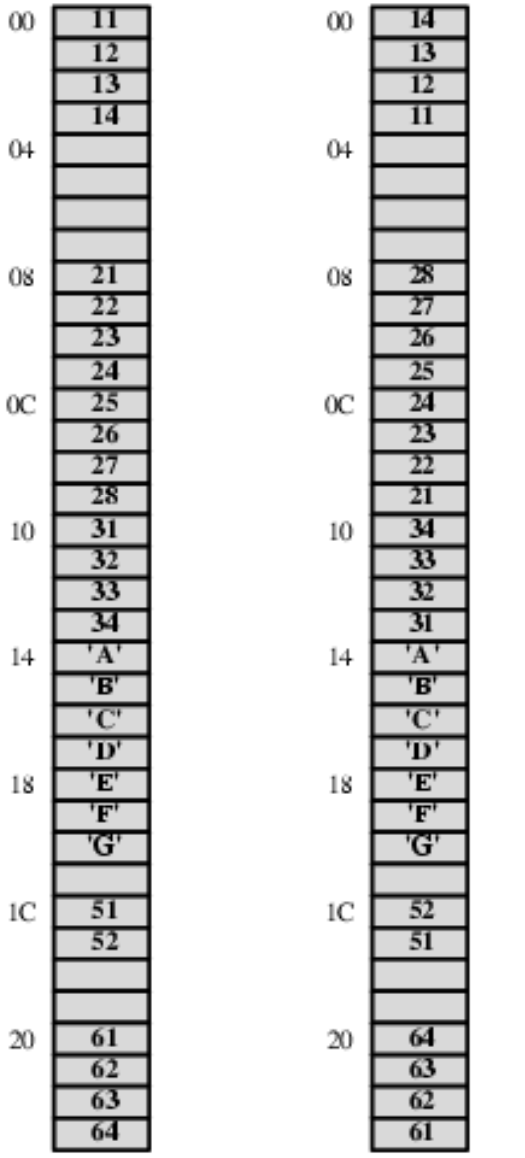
Big-endian address mapping

Byte Address								
00	**11** 00	**12** 01	**13** 02	**14** 03	04	05	06	07
08	**21** 08	**22** 09	**23** 0A	**24** 0B	**25** 0C	**26** 0D	**27** 0E	**28** 0F
10	**31** 10	**32** 11	**33** 12	**34** 13	**'A'** 14	**'B'** 15	**'C'** 16	**'D'** 17
18	**'E'** 18	**'F'** 19	**'G'** 1A	1B	**51** 1C	**52** 1D	1E	1F
20	**61** 20	**62** 21	**63** 22	**64** 23				

Little-endian address mapping

								Byte Address
				11 03	**12** 02	**13** 01	**14** 00	00
21 0F	**22** 0E	**23** 0D	**24** 0C	**25** 0B	**26** 0A	**27** 09	**28** 08	08
'D' 17	**'C'** 16	**'B'** 15	**'A'** 14	**31** 13	**32** 12	**33** 11	**34** 10	10
		51 1D	**52** 1C		**'G'** 1A	**'F'** 19	**'E'** 18	18
				61 23	**62** 22	**63** 21	**64** 20	20

Alternative View of Memory Map



(a) Big-endian (b) Little-endian

Standard…What Standard?

- Pentium (80x86), VAX are little-endian
- IBM 370, Moterola 680x0 (Mac), and most RISC are big-endian
- Internet is big-endian