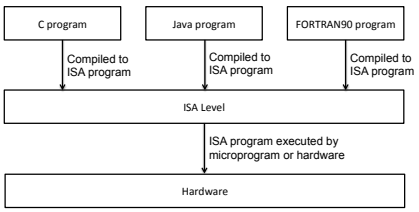
Computer Architecture

Instruction Set Principles

Madhu Mutyam
PACE Laboratory
Department of Computer Science and Engineering
Indian Institute of Technology Madras

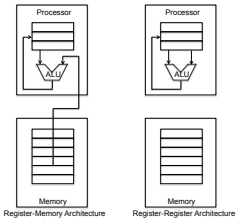
The Role of ISA

- Specifies the functionality of a processor
- Provides the interface between the compilers and hardware



Classification of ISA

- General-purpose register architecture



- Register-memory architecture
  - Memory can be accessed as part of any instruction
  - 2-operand ALU instructions
  - No separate load instruction to access data from memory
  - CPI vary based on operand location
  - Example: 80x86
- Register-register (or load-store) architecture
  - Memory is accessed through load/store instructions
  - 3-operand ALU instructions
  - Same type of ALU instructions take similar number of cycles
  - Instruction count is higher than register-memory architecture
  - Example: ARM

Memory Addressing

- 80x86 and ARM use byte addressing
- Provide access for bytes, half-words (2 bytes), words (4 bytes), and double words (8 bytes)
- Two important issues: *Endianness* and *Alignment*

Little Endian and Big Endian

- Little Endian: The least significant byte is stored in the smallest address
- Big Endian: The most significant byte is stored in the smallest address
- 80x86 – Little Endian; Motorola 6800 – Big Endian; ARM – switchable endianness

```
struct{
int       a;    // 0x11121314
int       b;
double c;    // 0x21222324252627
char*  d;    //0x31323334
char   e[7]; //'A', 'B', 'C', 'D', 'E', 'F', 'G'
short  f;    //0x5152
}
```

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

Little Endian Address Mapping

								Byte address
07	06	05	04	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
1F	1E	51 1D	52 1C	1B	'G' 1A	'F' 19	'E' 18	18

- Endianness does not affect the ordering of data items within a structure[1]
- Creates problem when exchanging data among computers with different ordering
- Registers do not care about endianness

[1]Example is taken from William Stallings book on Computer Organization & Architecture.

Alignment

- An access to an object of size s bytes at byte address A is aligned if A mod $s = 0$
- Example showing the addresses at which an access is aligned (indicated with "Y") or misaligned (indicated with "N")

Width of object	0	1	2	3	4	5	6	7
1 byte	Y	Y	Y	Y	Y	Y	Y	Y
2 bytes	Y		Y		Y		Y	
2 bytes		N		N		N		N
4 bytes	Y				Y			
4 bytes		N				N		
4 bytes			N				N	
4 bytes				N				N
8 bytes	Y							
8 bytes		N						
8 bytes			N					
8 bytes				N				
8 bytes					N			
8 bytes						N		
8 bytes							N	
8 bytes								N

- Misaligned operations may take multiple aligned memory references
- Supporting multi-byte accesses requires an alignment to align bytes, half-words, and words in 64-bit registers
- 80x86 does not require alignment while ARM requires it

Structure Member Alignment and Padding

```
typedef struct structA
{
   char c;
   short int s;

} structA_t;
```

Total Size = 4 Bytes

```
typedef struct structB
{
   short int s;
   char c;
   int i;
} structB_t;
```

Total Size = 8 Bytes

```
typedef struct structC
{
   char c;
   double d;
   int s;
} structC_t;
```

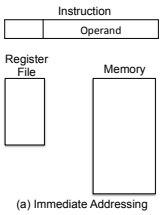
Total Size = 24 Bytes

```
typedef struct structD
{
   double d;
   int s;
   char c;
} structD_t;
```

Total Size = 16 Bytes

Addressing Modes

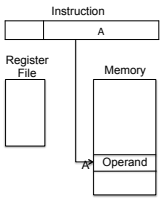
- Specify the way in which operands of an instruction are stored



(a) Immediate Addressing

ADD R1, Operand

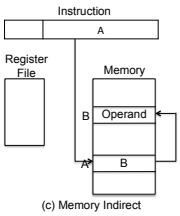
- Used for constants
- No memory reference
- Limited operand magnitude



(b) Direct Addressing

ADD R1, A

- Used for accessing static data
- Only one memory reference
- Limited address space

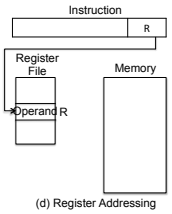


(c) Memory Indirect

ADD R1, (A)

- Used for pointer manipulations
- Large address space
- Two memory references

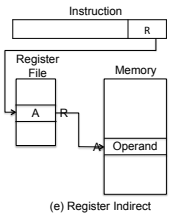
Addressing Modes



(d) Register Addressing

ADD R1, R

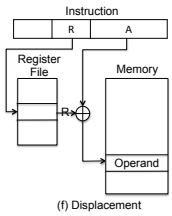
- Used for values stored in registers
- No memory reference
- Limited address space



(e) Register Indirect

ADD R1, (R)

- Used for pointer manipulations
- Large address space
- One memory reference



(f) Displacement

ADD R1, A(R)

- direct + register indirect addressing
- Flexibility
- Complex implementation

Uses of Displacement Addressing Mode

- *Relative addressing*
  - Displacement is relative to the address of the instruction
  - *PC-relative* addressing
  - Used for control-transfer instructions
- *Base-Register addressing*
  - Reference register contains a memory address
  - Address field contains a displacement from the memory address
  - Used for implementing segmentation
- *Indexed addressing*
  - Address field contains a memory address
  - Reference register contains a displacement from the memory address
  - Used for performing iterative operations

Addressing Modes Summary

- Instruction count can be reduced with some addressing modes, but CPI may increase
- 80x86 supports *register*, *immediate*, and *displacement* with *zero*, *one*, and *two registers*
- ARM supports *register*, *immediate*, *displacement* with *one* and *two registers*, *PC-relative*, *auto-increment*, and *auto-decrement*
- The addressing modes any new ISA need to have are *displacement*, *immediate*, and *register indirect*
- Choosing the displacement field sizes and the immediate field sizes are very important

Type and Size of Operands

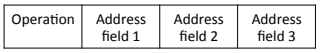
- Common operand types
  - Character (8 bits), half-word (16 bits), word (32 bits), single-precision FP (1 word), double-precision FP (2 words)
- Integers are represented as 2's complement binary number
- Characters are represented in ASCII
  - 16-bit unicode is gaining popularity
- IEEE standard 754 is used for FP numbers
  - Single-precision: Sign(1):Exponent(8):Mantissa(23)
  - Double-precision: Sign(1):Exponent(11):Mantissa(52)
  - Consider biased exponent and normalised mantissa
- BCD format can also be needed
  - Calculations that are exact in decimal can be inexact in binary

Common Instruction Operations

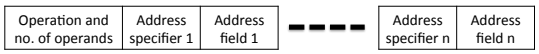
- Arithmetic/Logical: Integer ALU operations
  - ADD, SUB, DIV, AND, OR, ...
- Load/Stores: Data transfer between memory and registers
  - LOAD, STORE, MOVE
- Control: Instructions to change the flow of instruction execution
  - Conditional branches, jumps, procedure calls, procedure returns
  - PC-relative – the target address is known at compile time
  - Register indirect – the target address is not known at compile time
  - Condition codes are used to specify branch conditions
  - Procedure call places the return address in a register (ARM) or on a stack in memory (80x86)
  - BEQZ, BNEQ, JMP, CALL, RETURN, TRAP
- System: OS instructions, virtual memory management instructions
  - INT
- Floating-point: FP operations
  - FADD, FMULT, ...

Encoding An Instruction Set

- The number of registers and the number of addressing modes have an impact on the size of instructions
- *Fixed Length Encoding*:



  - The operation and the addressing mode are encoded into the opcode
  - Instruction decoding is simple
  - Example ISA: ARM
- *Variable Length Encoding*:



  - Separate address specifier is needed for each operand
  - Takes less space
  - Example ISA: 80x86

CISC Vs RISC

- Complex Instruction Set Computer (CISC) Architecture uses multi-word instructions
  - The primary goal is to complete a task in as few lines of assembly as possible
  - supporting the operations and data structures used by the high-level language
  - Supporting a large variety of memory addressing modes
  - Results in variable length instructions
  - Example ISA: x86
- Reduced Instruction Set Computer (RISC) Architecture uses one-word instructions
  - Uses processor registers extensively
  - Operands must be from registers only
  - Load-store architecture
  - Register-based addressing is used
    - Memory addressing modes are used only for loads/stores
  - Example ISA: ARM

Thank You