Instruction Set Architecture (ISA) - Part II

CS6133 - Computer Architecture I

Vikram Padman

NYU Polytechnic School of Engineering

vikram@poly.edu

Agenda

1. **Introduction** - Part I
2. **Classifying ISA** - Part I, II, III and IV
   - Memory Addressing - Part II
   - Type and Size of operands - Part II
3. **Activity**

Reading List

- "Computer Architecture - A Quantitative Approach" - Appendix A in Fifth Edition or Appendix B in Fourth Edition
- "Computer Organization and Design" - Chapter 2 in Fourth Edition or Third Edition
- "Digital Design and Computer Architecture" - Chapter 6

Memory Addressing

A byte (8-bits) is the smallest addressable unit allowed in a modern CPU. However, a 64-bit double word is the smallest addressable unit in modern memory subsystems.

- So how are bytes arranged in a 64-bit double word? and
- How does byte access work?

Memory Addressing

Byte Order

There are essentially two popular ways to store bytes within a 64-bit double word:

1. Big Endian byte order

Bit	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Big Endian	0								1								2								3								4								5								6								7							

2. Little Endian byte order

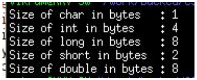
Bit	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Little Endian	7								6								5								4								3								2								1								0							

Let's consider the following C code:

```
#include<stdio.h>
int main(){
char c;
int i;
long l;
short s;
double d;

printf("Size of char in bytes   : %zu \n", sizeof(c));
printf("Size of int in bytes    : %zu \n", sizeof(i));
printf("Size of long in bytes   : %zu \n", sizeof(l));
printf("Size of short in bytes  : %zu \n", sizeof(s));
printf("Size of double in bytes : %zu \n", sizeof(d));

return 0;
}
```



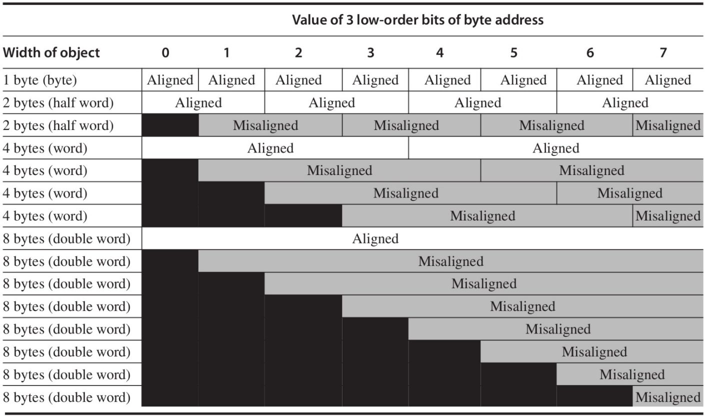
```
Size of char in bytes   : 1
Size of int in bytes    : 4
Size of long in bytes   : 8
Size of short in bytes  : 2
Size of double in bytes : 8
```

Most applications use a wide variety of data types and structures. To store and retrieve them efficiently compilers try to align data structures to word or double word boundaries.

Memory Addressing

Address Alignment

Often, data is not aligned to a double word boundary.
Misaligned data results in extra memory cycles and reduces the performance.



	Value of 3 low-order bits of byte address							
Width of object	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1 byte (byte)	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned
2 bytes (half word)	Aligned		Aligned		Aligned		Aligned	
2 bytes (half word)	■	Misaligned		Misaligned		Misaligned		Misaligned
4 bytes (word)	Aligned				Aligned			
4 bytes (word)	■	Misaligned				Misaligned		
4 bytes (word)	■	■	Misaligned				Misaligned	
4 bytes (word)	■	■	■	Misaligned				Misaligned
8 bytes (double word)	Aligned							
8 bytes (double word)	■	Misaligned						
8 bytes (double word)	■	■	Misaligned					
8 bytes (double word)	■	■	■	Misaligned				
8 bytes (double word)	■	■	■	■	Misaligned			
8 bytes (double word)	■	■	■	■	■	Misaligned		
8 bytes (double word)	■	■	■	■	■	■	Misaligned	
8 bytes (double word)	■	■	■	■	■	■	■	Misaligned

From "Computer Architecture – A Quantitative Approach" page A-8

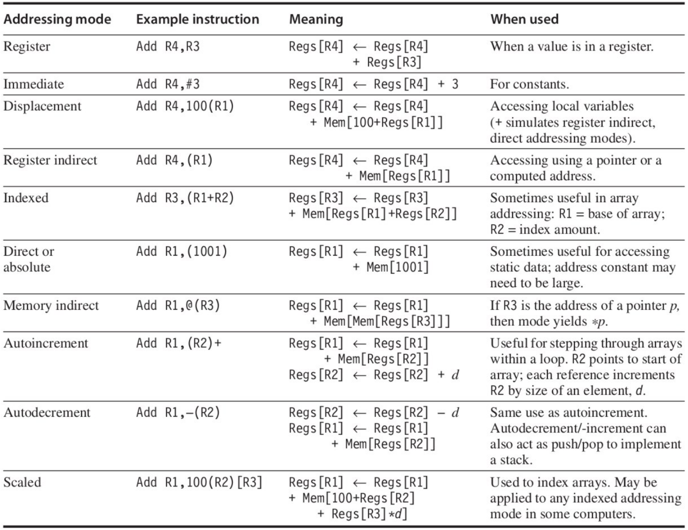
Memory Addressing
Addressing Modes

- Addressing modes states how the address of a data unit is specified
- Modern CPU's, even the low power CPU, supports a variety of addressing modes
- Addressing modes has a direct impact on an instructions length
- Addressing modes also have the ability to reduce instruction count, but could increase implementation complexity.
- Complexity usually increases the number of clock cycles required to complete an instruction

Effective Address is the actual address of the data unit. In some addressing modes the effective address is stated directly and in others it is calculated.

Memory Addressing

Addressing Modes



Addressing mode	Example instruction	Meaning	When used
Register	`Add R4,R3`	`Regs[R4] ← Regs[R4] + Regs[R3]`	When a value is in a register.
Immediate	`Add R4,#3`	`Regs[R4] ← Regs[R4] + 3`	For constants.
Displacement	`Add R4,100(R1)`	`Regs[R4] ← Regs[R4] + Mem[100+Regs[R1]]`	Accessing local variables (+ simulates register indirect, direct addressing modes).
Register indirect	`Add R4,(R1)`	`Regs[R4] ← Regs[R4] + Mem[Regs[R1]]`	Accessing using a pointer or a computed address.
Indexed	`Add R3,(R1+R2)`	`Regs[R3] ← Regs[R3] + Mem[Regs[R1]+Regs[R2]]`	Sometimes useful in array addressing: R1 = base of array; R2 = index amount.
Direct or absolute	`Add R1,(1001)`	`Regs[R1] ← Regs[R1] + Mem[1001]`	Sometimes useful for accessing static data; address constant may need to be large.
Memory indirect	`Add R1,@(R3)`	`Regs[R1] ← Regs[R1] + Mem[Mem[Regs[R3]]]`	If R3 is the address of a pointer *p*, then mode yields **p*.

Autoincrement	`Add R1,(R2)+`	`Regs[R1] ← Regs[R1] + Mem[Regs[R2]]`
		`Regs[R2] ← Regs[R2] + d`
			Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, *d*.

Autodecrement	`Add R1,–(R2)`	`Regs[R2] ← Regs[R2] – d`
		`Regs[R1] ← Regs[R1] + Mem[Regs[R2]]`
			Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack.

Scaled	`Add R1,100(R2)[R3]`	`Regs[R1] ← Regs[R1] + Mem[100+Regs[R2] + Regs[R3]*d]`	Used to index arrays. May be applied to any indexed addressing mode in some computers.

From "Computer Architecture – A Quantitative Approach" page A-9

Type & Size of Operands

- Modern CPU's support a variety of data type. A GPU, for example, have native support for large matrices and could perform complex calculation in a few clock cycles.
- The type and size of an operand is specified in the opcode or in the opcode extension.
- Commonly used data types in a general purpose CPU are:
  - Byte(8-bits), half word(16 bits), word (32 bits) and double word(64 bits)
  - IEEE floating point numbers
  - Padded and unpadded decimals

Week 5 - Activity 2

Consider the following C code:

```
int a[10], b[10], c[10], d=9, i;

for(i=0; i<10; i++)
  a[i] = (b[i] + c[i]) - (c[i] * d);
```

1. Write an equivalent assembly code for four different storage types described in the previous lecture (Section A.1 and A.2). For this part use only register and immediate addressing modes. You could invent your own assembly mnemonics (use figure A.2 for reference)
2. Rewrite (assembly code) using an addressing mode that would yield the lowest number of instructions for each storage type.
3. Let's say that a[0] is stored at memory address 0x00001000, b[0] at 0x00002003 and c[0] at 0x000040FE. Calculate the total number of memory read and write operations. Assume that memory is 64-bit wide, an integer is 32 bits and the architecture allows byte addressing.