Department of Electrical Engineering, Feng-Chia University

Chapter 2
Instructions: Language of the Computer
(Part 3)

王振傑 (Chen-Chieh Wang)
ccwang@mail.ee.ncku.edu.tw

Department of Electrical Engineering, Feng-Chia University

Outline

2.12 Translating and Starting a Program
2.14 Arrays versus Pointers
2.16 Real Stuff: ARM Instructions
2.17 Real Stuff: x86 Instructions
2.18 Fallacies and Pitfalls
2.19 Concluding Remarks

Software

Source Program

- Any sequence of statements and/or declarations written in some human-readable computer programming language (e.g. C++, Assembly program)
- Usually created using a text editor (ASCII file)

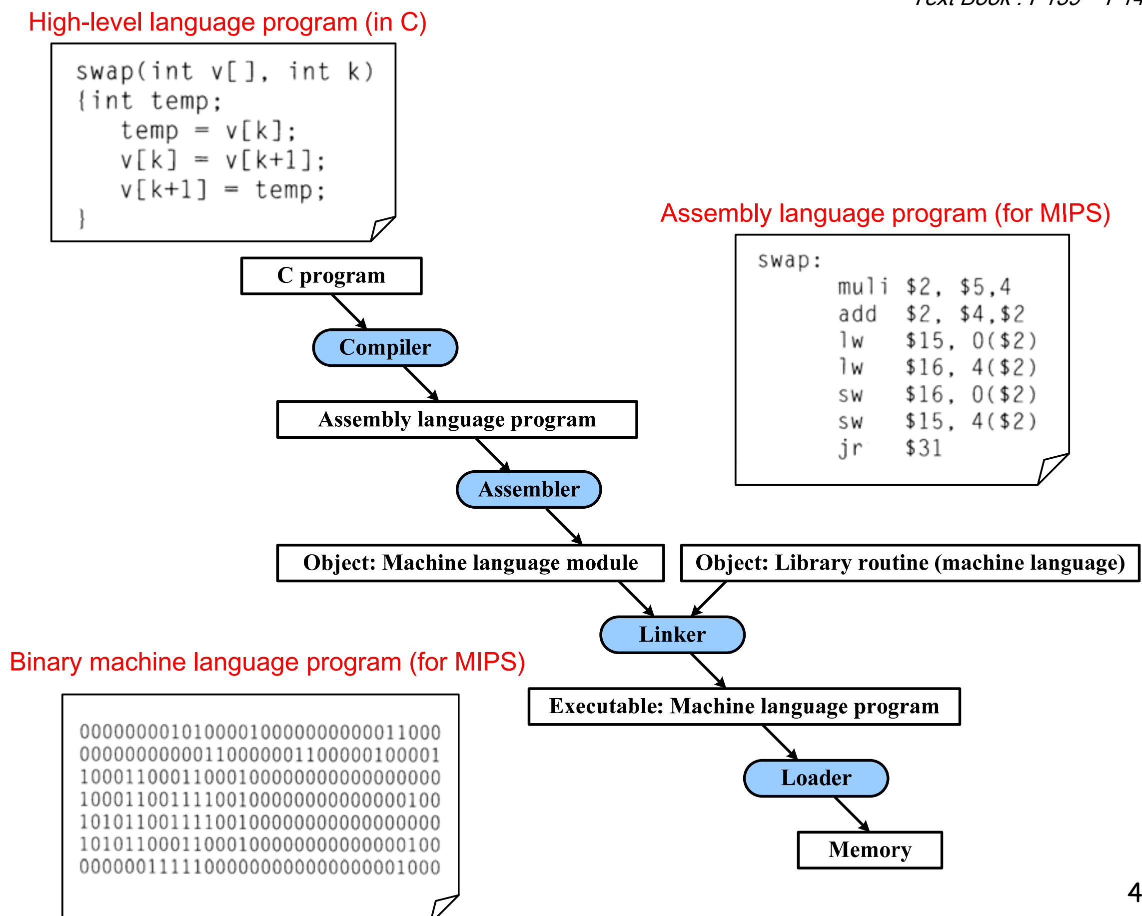
Object Program

- Produced from a source program by compiling/assembling to intermediate machine code
- Also contain data for use by the code at runtime, relocation information, program symbols for linking and/or debugging purposes, and other debugging information

Executable Program

- Machine code directly executed by a computer's CPU

Text Book : P139 ~ P145



Department of Electrical Engineering, Feng-Chia University

Assembler Pseudo-instructions

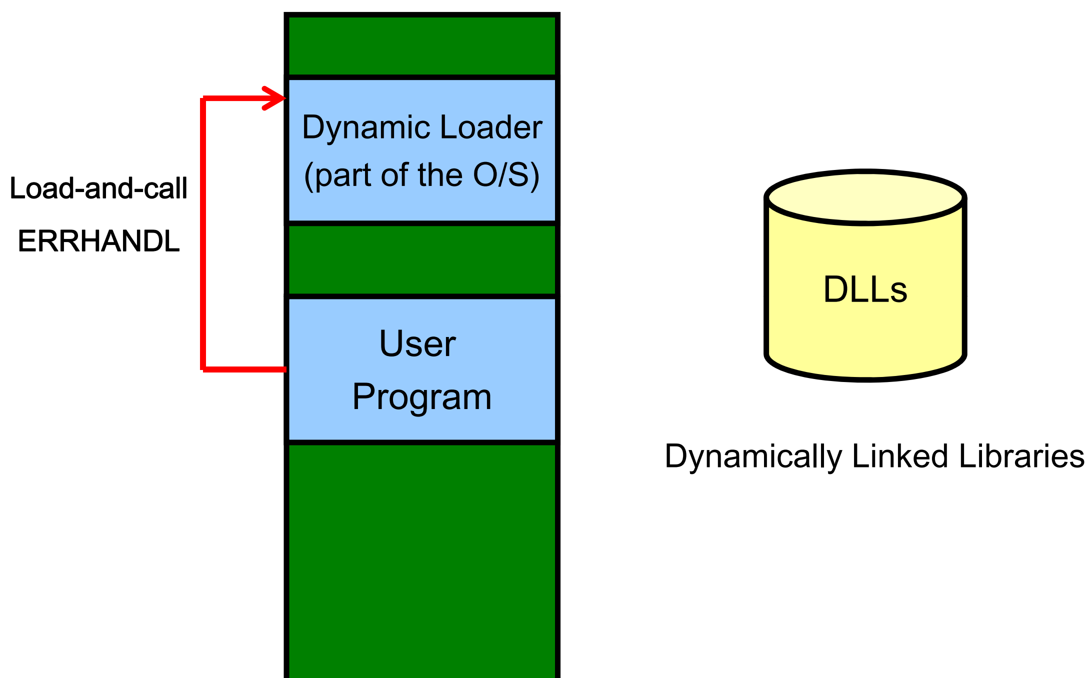
- Most assembler instructions represent machine instructions one-to-one
- Pseudo-instructions: figments of the assembler's imagination

```
move $t0, $t1      →    add $t0, $zero, $t1
blt $t0, $t1, L    →    slt $at, $t0, $t1
                        bne $at, $zero, L
```

- $at (register 1): assembler temporary

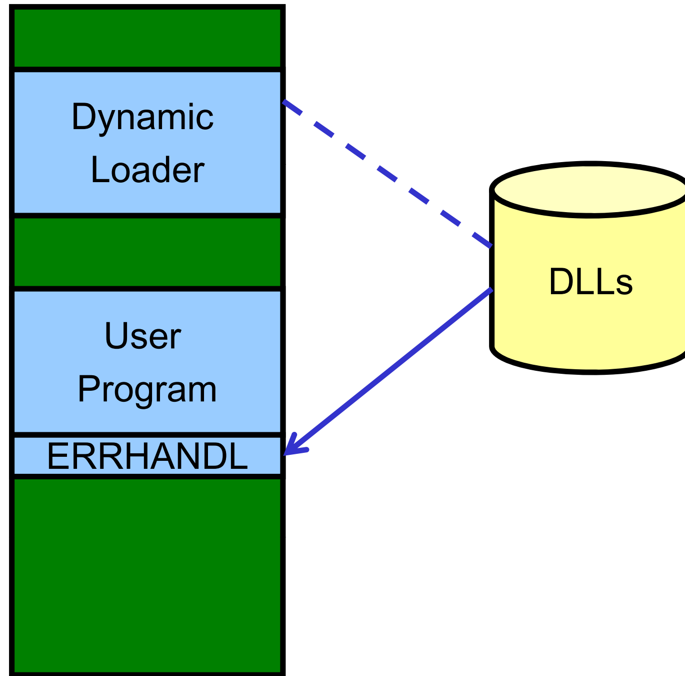
Department of Electrical Engineering, Feng-Chia University

Dynamic linking



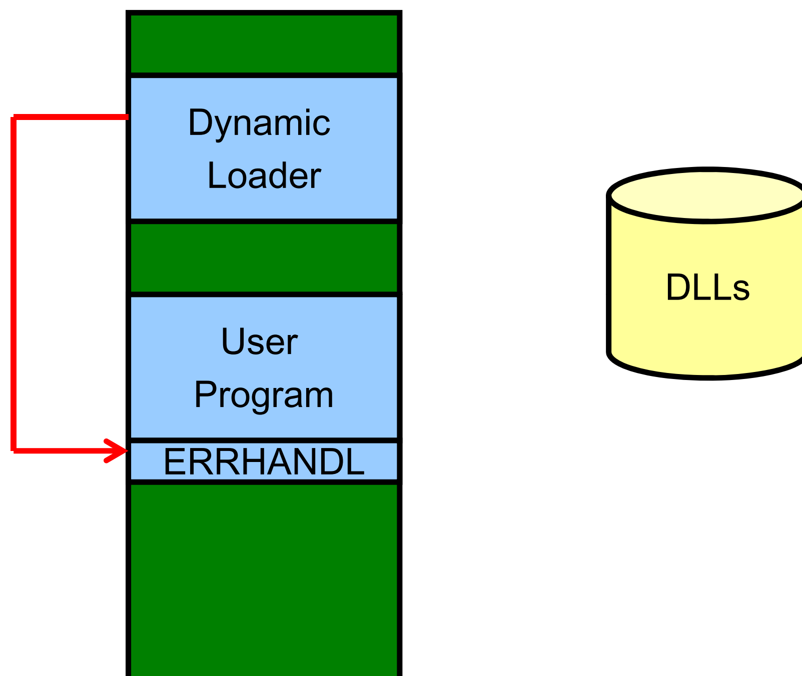
Department of Electrical Engineering, Feng-Chia University

Dynamic linking



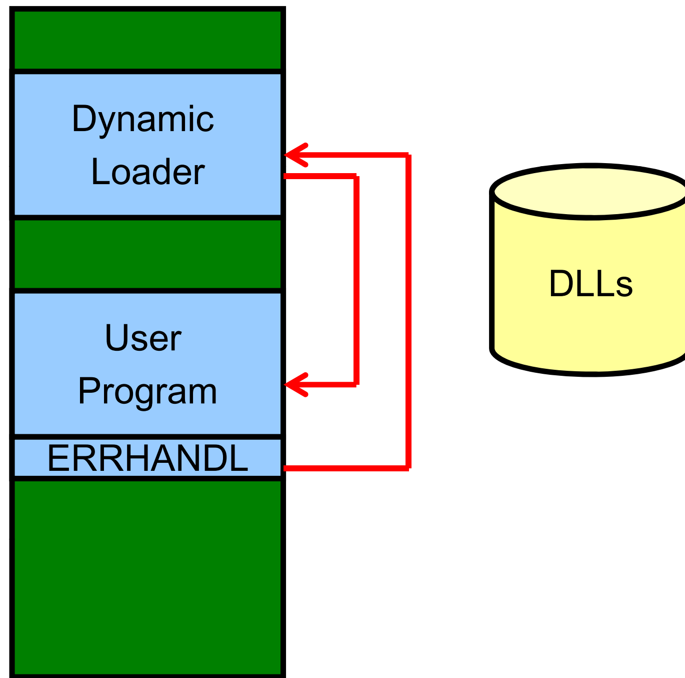
Department of Electrical Engineering, Feng-Chia University

Dynamic linking



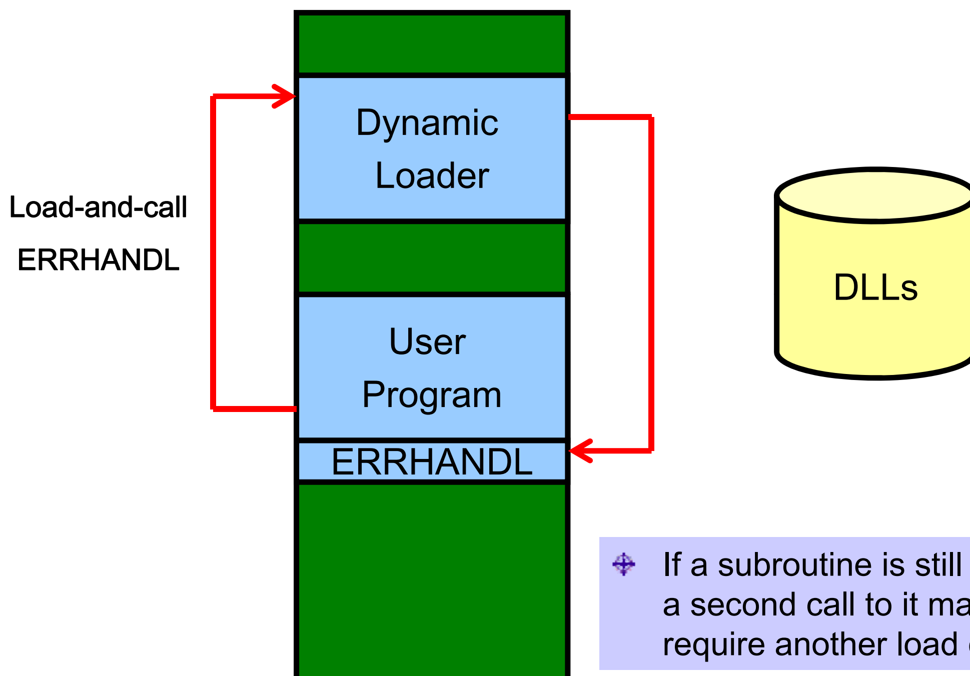
Department of Electrical Engineering, Feng-Chia University

Dynamic linking



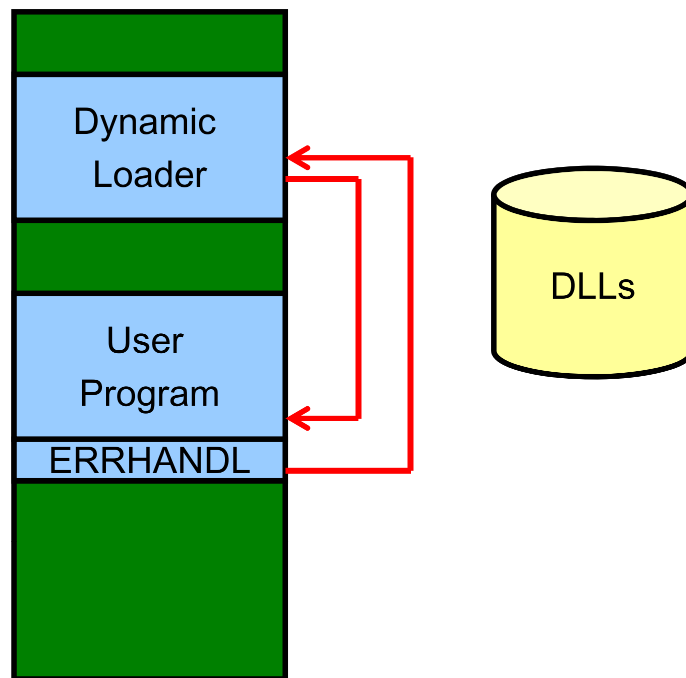
Department of Electrical Engineering, Feng-Chia University

Dynamic linking



- If a subroutine is still in memory, a second call to it may not require another load operation.

Dynamic linking



Outline

2.12 Translating and Starting a Program
2.14 Arrays versus Pointers
2.16 Real Stuff: ARM Instructions
2.17 Real Stuff: x86 Instructions
2.18 Fallacies and Pitfalls
2.19 Concluding Remarks

Text Book : P157 ~ P160

Arrays vs. Pointers

- C Code

Array	Pointer

clear1(int array[], int size)
{
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
	clear2(int *array, int size)
	{
	  int *p;
	  for (p = &array[0]; p < &array[size]; p = p + 1)
	    *p = 0;
	}

- MIPS Code

Array	Pointer

move $t0,$zero # i = 0
loop1: sll $t1,$t0,2 # $t1 = i * 4
add $t2,$a0,$t1 # $t2 = &array[i]
sw $zero, 0($t2) # array[i] = 0
addi $t0,$t0,1 # i = i + 1
slt $t3,$t0,$a1 # $t3 = (i < size)
bne $t3,$zero,loop1 # if () go to loop1
	move $t0,$a0 # p = & array[0]
	sll $t1,$a1,2 # $t1 = size * 4
	add $t2,$a0,$t1 # $t2 = &array[size]
	loop2: sw $zero,0($t0) # Memory[p] = 0
	addi $t0,$t0,4 # p = p + 4
	slt $t3,$t0,$t2 # $t3=(p<&array[size])
	bne $t3,$zero,loop2 # if () go to loop2

Outline

2.12 Translating and Starting a Program
2.14 Arrays versus Pointers
2.16 Real Stuff: ARM Instructions
2.17 Real Stuff: x86 Instructions
2.18 Fallacies and Pitfalls
2.19 Concluding Remarks

ARM & MIPS Similarities

Department of Electrical Engineering, Feng-Chia University

- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

	ARM	MIPS
Date announced	1985	1985
Instruction size	32 bits	32 bits
Address space	32-bit flat	32-bit flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Registers	37 × 32-bit	35 × 32-bit
Input/output	Memory mapped	Memory mapped

ARM introduction

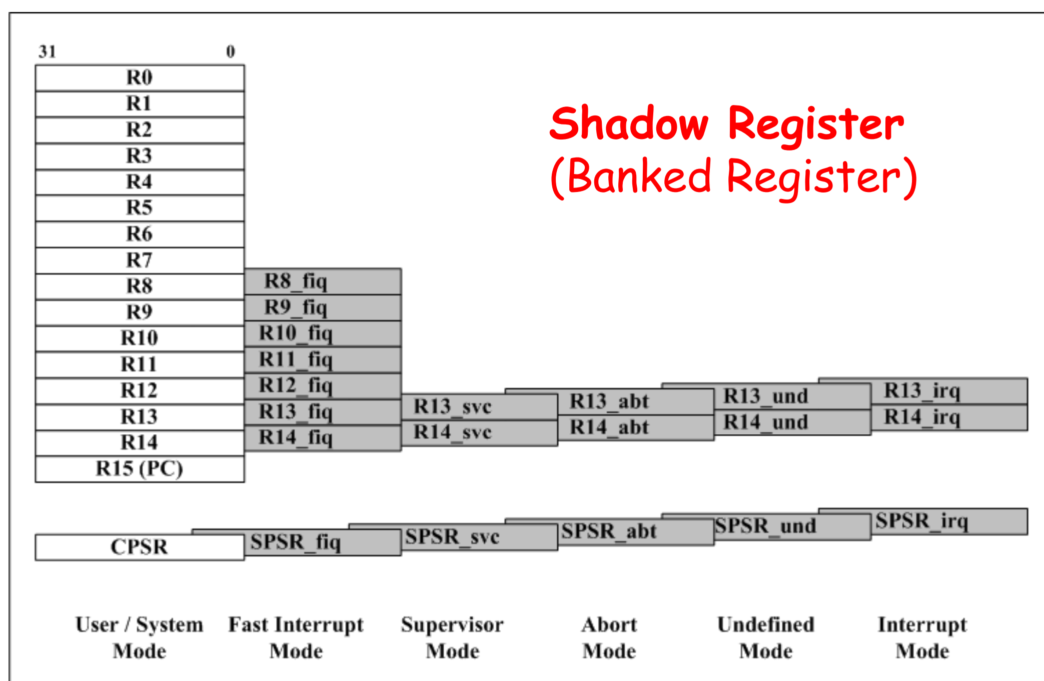
Department of Electrical Engineering, Feng-Chia University

- A 32-bit RISC architecture.
  - A large uniform register file
  - Many instructions execute in a single cycle
  - A load/Store architecture
  - Simple addressing mode, with all load/store addresses being determined form register contents, not directly on memory contents.
  - Uniform and fixed-length instruction fields, to simplify instruction decode. (32-bit length and 3-address format)
- Other features
  - Control over both the ALU and Shifter in every data-processing instruction to maximize the use of an ALU and Shifter.
  - Auto-increment and auto-decrement addressing modes to optimize program loops.
  - Load and Store Multiple instructions to maximize data throughput.
  - Conditional execution of all instructions to maximize execution throughput.

ARM register briefs

- ARM has 31 general-purpose 32-bit registers. Only 16 of them are visible, R0 to R15.
- ARM has 6 Program status registers (PSR).
- The 16 registers are User mode register. Only exception can change User mode to other processor mode.
- R14 is Link register used for holding the address of next to a Branch and link.
- R15 is program counter (PC).
- PC points to instruction that is two instruction being executed (In EXE).
- R13 is generally used as a Stack Pointer (SP). This is defined by the Software.

Register File



Department of Electrical Engineering, Feng-Chia University

Text Book : P163

Compare and Branch in ARM

- Uses condition codes for result of an arithmetic/logical instruction
  - Negative, zero, carry, overflow
  - Compare instructions to set condition codes without keeping the result
- Each instruction can be conditional
  - Top 4 bits of instruction word: condition value
  - Can avoid branches over single instructions

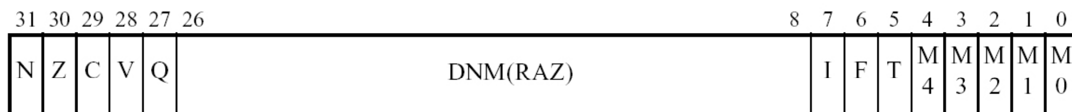
Department of Electrical Engineering, Feng-Chia University

Condition Codes

Opcode [31:28]	Mnemonic extension	Meaning	Condition flag state
0000	EQ	Equal	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/unsigned higher or same	C set
0011	CC/LO	Carry clear/unsigned lower	C clear
0100	MI	Minus/negative	N set
0101	PL	Plus/positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N set and V set, or N clear and V clear (N == V)
1011	LT	Signed less than	N set and V clear, or N clear and V set (N != V)
1100	GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0,N == V)
1101	LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
1110	AL	Always (unconditional)	-
1111	(NV)	See *Condition code 0b1111* on page A3-5	-

Department of Electrical Engineering, Feng-Chia University

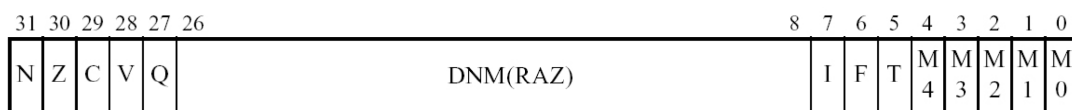
Program status registers



- 4 condition code flags
  - (N, Z, C, V) flags：Negative, Zero, Carry, oVerflow
- 1 sticky overflow flag
  - Q bit：DSP instruction overflow bit.
  - In E variants of ARM architecture 5 and above.
- 2 interrupt disable bits
  - I bit：disable normal interrupt (IRQ)
  - F bit：disable fast interrupt (FIQ)
- 1 bit which encodes whether ARM or Thumb instructions are being executed.
  - T bit

Department of Electrical Engineering, Feng-Chia University

Program status registers (cont.)



- 5 bits that encode the current processor mode.
  - M[4:0] are the mode bits.

M[4:0]	Mode	Accessible registers
0b10000	User	PC, R14 to R0, CPSR
0b10001	FIQ	PC, R14_fiq to R8_fiq, R7 to R0, CPSR, SPSR_fiq
0b10010	IRQ	PC, R14_irq, R13_irq, R12 to R0, CPSR, SPSR_irq
0b10011	Supervisor	PC, R14_svc, R13_svc, R12 to R0, CPSR, SPSR_svc
0b10111	Abort	PC, R14_abt, R13_abt, R12 to R0, CPSR, SPSR_abt
0b11011	Undefined	PC, R14_und, R13_und, R12 to R0, CPSR, SPSR_und
0b11111	System	PC, R14 to R0, CPSR (ARM architecture v4 and above)

Exceptions

- Vector address

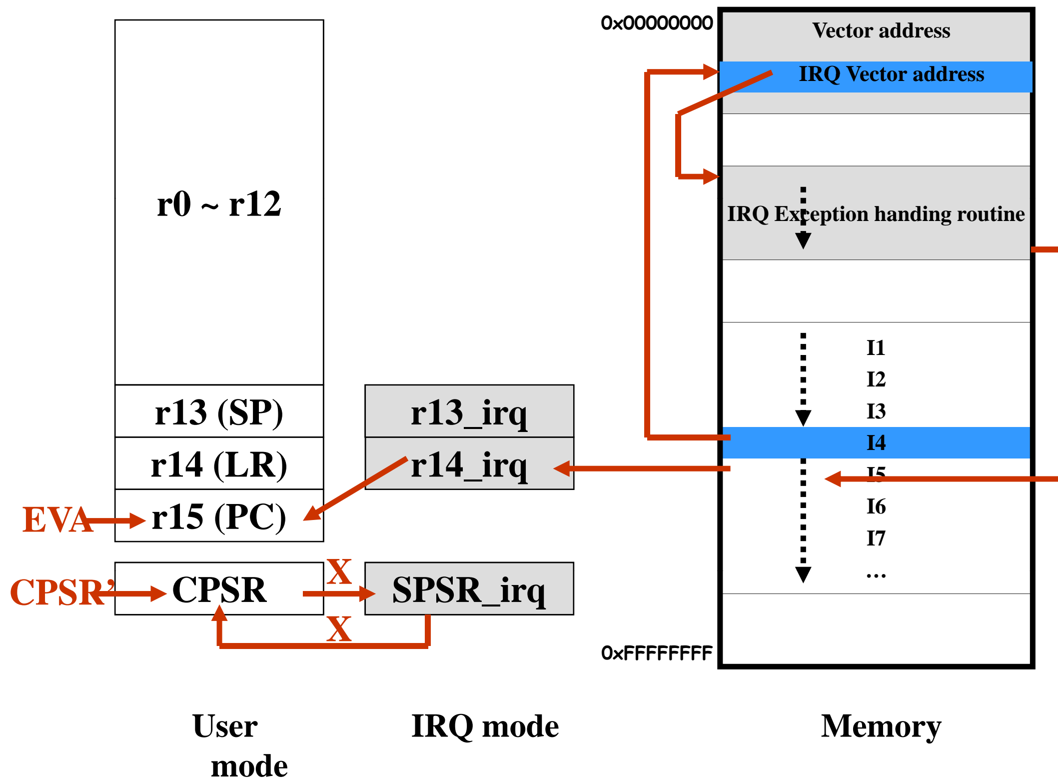
Exception type	Mode	Normal address	High vector address
Reset	Supervisor	0x00000000	0xFFFF0000
Undefined instructions	Undefined	0x00000004	0xFFFF0004
Software interrupt (SWI)	Supervisor	0x00000008	0xFFFF0008
Prefetch Abort (instruction fetch memory abort)	Abort	0x0000000C	0xFFFF000C
Data Abort (data access memory abort)	Abort	0x00000010	0xFFFF0010
IRQ (interrupt)	IRQ	0x00000018	0xFFFF0018
FIQ (fast interrupt)	FIQ	0x0000001C	0xFFFF001C

Exception process

- When an exception occurs, the banked versions of R14 and the SPSR for the exception mode are used to save state as follows:

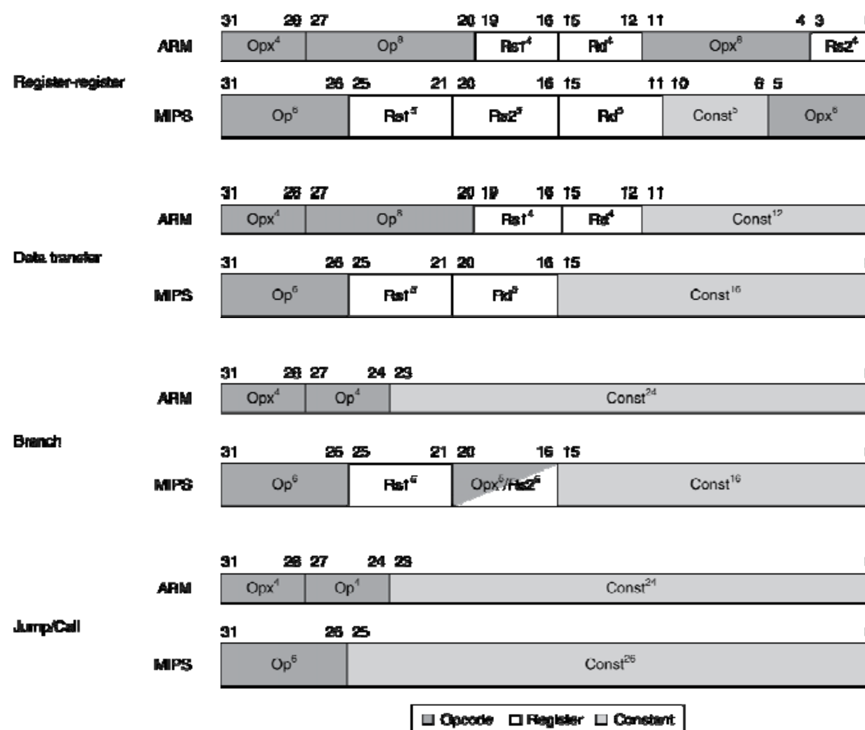
```
R14_<exception_mode> = return link
SPSR_<exception_mode> = CPSR
CPSR[4:0] = exception mode number
CPSR[5] = 0                                  /* Execute in ARM state */
If <exception_mode> == Reset or FIQ then
      CPSR[6] = 1                            /* Disable fast interrupts */
/* else CPSR[6] is unchanged */
CPSR[7] = 1                                  /* Disable normal interrupts */
PC = exception vector address
```

Example : IRQ



Text Book : P164

Instruction Encoding



Department of Electrical Engineering, Feng-Chia University

Thumb Architecture Extension

- The Thumb instruction set is a re-encoded subset of the ARM instruction set and the instructions operate on restricted view of the ARM registers. (R0~R7, R13, R14, R15)
- Thumb is designed to increase the performance of ARM implementations that use a 16-bit or narrower memory data bus and to allow better code density than ARM
- Every Thumb instruction is encode in 16 bits.
- Most Thumb instructions are executed unconditionally.
- Many Thumb data processing instructions use 2-address format. (the destination register is the same as one of the source registers)

Department of Electrical Engineering, Feng-Chia University

Example : ARM vs. Thumb

Simple C routine

```
if (x>=0)
    return x;
else
    return -x;
```

The equivalent ARM assembly

```
labs    CMP     r0,#0      ;Compare r0 to zero
        RSBLT   r0,r0,#0   ;If r0<0 (less than=LT) then do r0= 0-r0
        MOV     pc,lr      ;Move Link Register to PC (Return)
```

The equivalent Thumb assembly

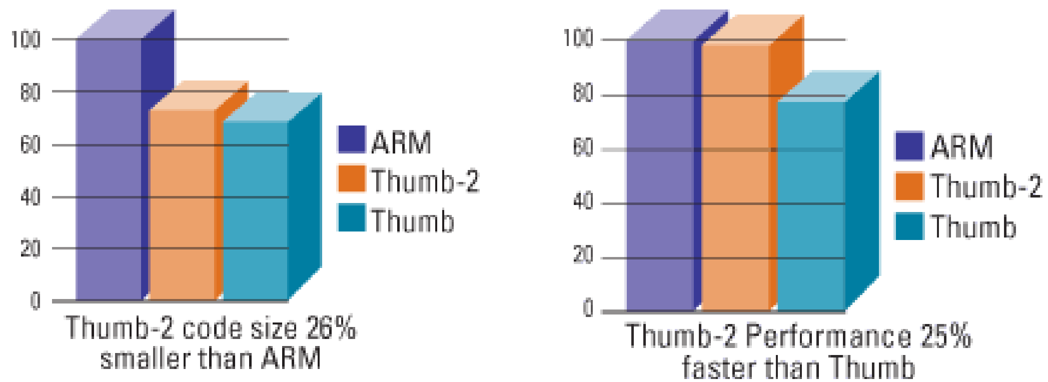
```
        CODE16 ;Directive specifying 16-bit (Thumb) instructions
labs    CMP     r0,#0      ;Compare r0 to zero
        BGE     return     ;Jump to Return if greater or
                           ;equal to zero
        NEG     r0,r0      ;If not, negate r0
return  MOV     pc,lr      ;Move Link register to PC (Return)
```

Code	Instructions	Size (Bytes)	Normalised
ARM	3	12	1.0
Thumb	4	8	0.67

Department of Electrical Engineering, Feng-Chia University

Thumb-2

- Improved code density with performance and power efficiency.



Department of Electrical Engineering, Feng-Chia University

Outline

2.12 Translating and Starting a Program
2.14 Arrays versus Pointers
2.16 Real Stuff: ARM Instructions
2.17 Real Stuff: x86 Instructions
2.18 Fallacies and Pitfalls
2.19 Concluding Remarks

Department of Electrical Engineering, Feng-Chia University

Alternative Architectures

- Design alternative:
  - provide more powerful operations
  - goal is to reduce number of instructions executed
  - danger is a slower cycle time and/or a higher CPI

– *"The path toward operation complexity is thus fraught with peril. To avoid these problems, designers have moved toward simpler instructions"*

- Let's look (briefly) at IA-32 (x86)

Department of Electrical Engineering, Feng-Chia University

The Intel x86 ISA

- Evolution with backward compatibility
  - 8080 (1974): 8-bit microprocessor
    - Accumulator, plus 3 index-register pairs
  - 8086 (1978): 16-bit extension to 8080
    - Complex instruction set (CISC)
  - 8087 (1980): floating-point coprocessor
    - Adds FP instructions and register stack
  - 80286 (1982): 24-bit addresses, MMU
    - Segmented memory mapping and protection
  - 80386 (1985): 32-bit extension (now IA-32)
    - Additional addressing modes and operations
    - Paged memory mapping as well as segments

Department of Electrical Engineering, Feng-Chia University

The Intel x86 ISA

- Further evolution…
  - i486 (1989): pipelined, on-chip caches and FPU
    - Compatible competitors: AMD, Cyrix, …
  - Pentium (1993): superscalar, 64-bit datapath
    - Later versions added MMX (Multi-Media eXtension) instructions
    - The infamous FDIV bug
  - Pentium Pro (1995), Pentium II (1997)
    - New microarchitecture (see Colwell, *The Pentium Chronicles*)
  - Pentium III (1999)
    - Added SSE (Streaming SIMD Extensions) and associated registers
  - Pentium 4 (2001)
    - New microarchitecture
    - Added SSE2 instructions

Department of Electrical Engineering, Feng-Chia University

The Intel x86 ISA

- And further…
  - AMD64 (2003): extended architecture to 64 bits
  - EM64T – Extended Memory 64 Technology (2004)
    - AMD64 adopted by Intel (with refinements)
    - Added SSE3 instructions
  - Intel Core (2006)
    - Added SSE4 instructions, virtual machine support
  - AMD64 (announced 2007): SSE5 instructions
    - Intel declined to follow, instead…
  - Advanced Vector Extension (announced 2008)
    - Longer SSE registers, more instructions
- If Intel didn't extend with compatibility, its competitors would!
  - Technical elegance ≠ market success

Department of Electrical Engineering, Feng-Chia University

x86 Overview

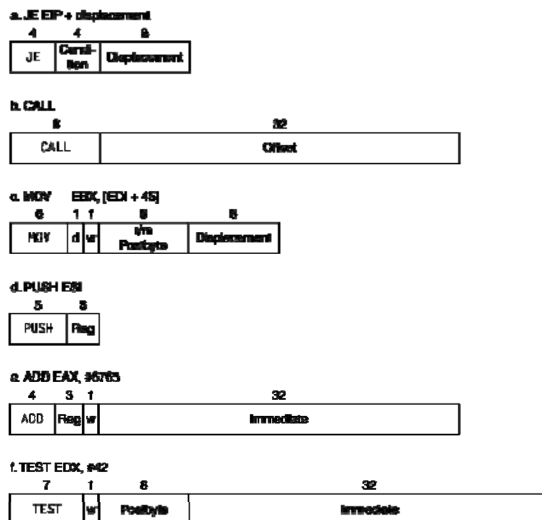
- Complexity:
  - Instructions from 1 to 17 bytes long
  - one operand must act as both a source and destination
  - one operand can come from memory
  - complex addressing modes
    e.g., “base or scaled index with 8 or 32 bit displacement”
- Saving grace:
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow

“what the 80x86 lacks in style is made up in quantity,
making it beautiful from the right perspective”

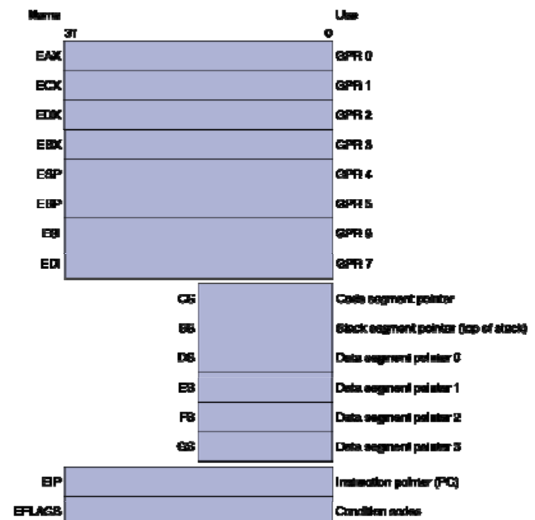
Department of Electrical Engineering, Feng-Chia University

x86 Instruction Encoding

- Variable length encoding
  - Postfix bytes specify addressing mode
  - Prefix bytes modify operation
    - Operand length, repetition, locking, …



Basic x86 Registers



Implementing IA-32

- Complex instruction set makes implementation difficult
  - Hardware translates instructions to simpler microoperations
    - Simple instructions: 1–1
    - Complex instructions: 1–many
  - Microengine similar to RISC
  - Market share makes this economically viable
- Comparable performance to RISC
  - Compilers avoid complex instructions

Outline

2.12 Translating and Starting a Program
2.14 Arrays versus Pointers
2.16 Real Stuff: ARM Instructions
2.17 Real Stuff: x86 Instructions
2.18 Fallacies and Pitfalls
2.19 Concluding Remarks

Text Book : P174

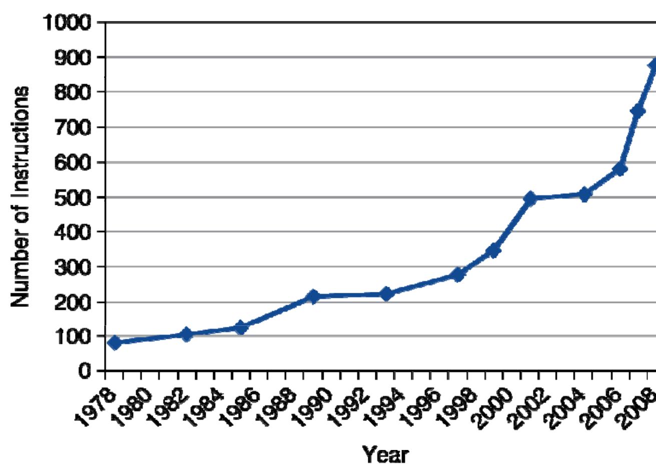
Fallacies

- Powerful instruction ⇒ higher performance
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions
- Use assembly code for high performance
  - But modern compilers are better at dealing with modern processors
  - More lines of code ⇒ more errors and less productivity

Text Book : P175

Fallacies

- Backward compatibility ⇒ instruction set doesn't change
  - But they do accrete more instructions



x86 instruction set

Pitfalls

- Sequential words are not at sequential addresses
  - Increment by 4, not by 1!
- Keeping a pointer to an automatic variable after procedure returns
  - e.g., passing pointer back via an argument
  - Pointer becomes invalid when stack popped

Outline

2.12 Translating and Starting a Program
2.14 Arrays versus Pointers
2.16 Real Stuff: ARM Instructions
2.17 Real Stuff: x86 Instructions
2.18 Fallacies and Pitfalls
2.19 Concluding Remarks

Text Book : P176 ~ P177

Concluding Remarks

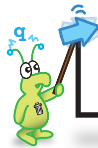
- Instruction complexity is only one variable
  - lower instruction count vs. higher CPI / lower clock rate
- Design Principles:
  - simplicity favors regularity
  - smaller is faster
  - make the common case fast
  - good design demands compromise
- Instruction set architecture
  - a very important abstraction indeed!
- Required instruction groups
  - Arithmetic and logic operations
  - Load/store
  - Control transfer

Text Book : P179

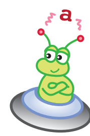
Concluding Remarks

- Measure MIPS instruction executions in benchmark programs
  - Consider making the common case fast
  - Consider compromises

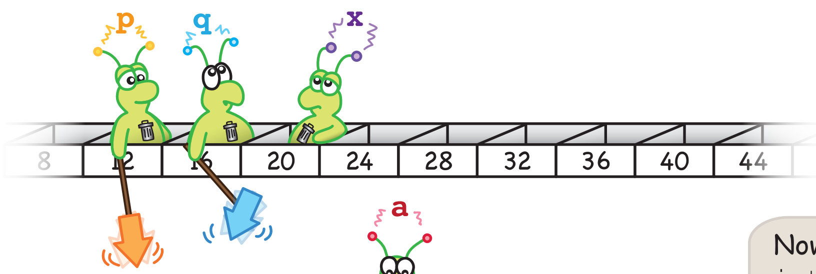
Instruction class	MIPS examples	SPEC2006 Int	SPEC2006 FP
Arithmetic	add, sub, addi	16%	48%
Data transfer	lw, sw, lb, lbu, lh, lhu, sb, lui	35%	36%
Logical	and, or, nor, andi, ori, sll, srl	12%	4%
Cond. Branch	beq, bne, slt, slti, sltiu	34%	8%
Jump	j, jr, jal	2%	0%



Pointers and Arrays



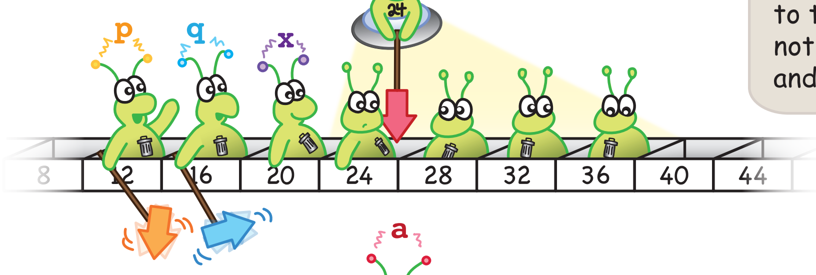
Imagine memory as long block of boxes that store data. Each box is labeled with an **address**. A **pointer** is simply a variable that holds a particular address. An **array** is a group of contiguous boxes that can be accessed by their index values. Array and pointer variables are mostly the same; we're going to highlight one of the ways they are different.




```
int *p, *q, x;
```

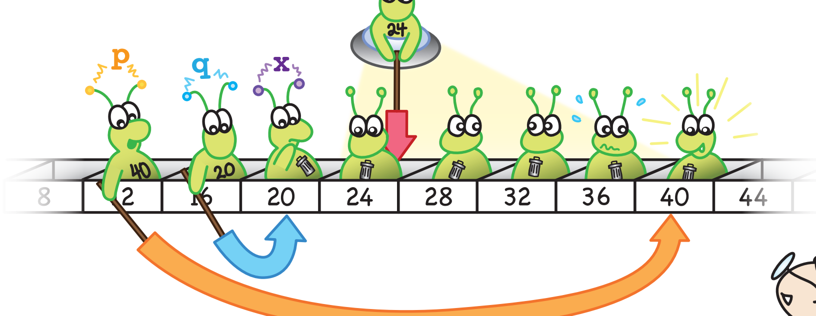
Here, we declare p and q as pointers that will hold the addresses of `int` variables, and x as an ordinary `int` variable.

Now we define an array that can store 4 `int` values. a is now a variable that points to the first index of this array. However, notice that unlike the pointer variables p and q, a does NOT live in memory.

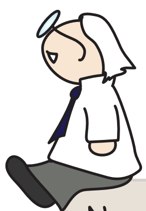



```
int a[4];
```

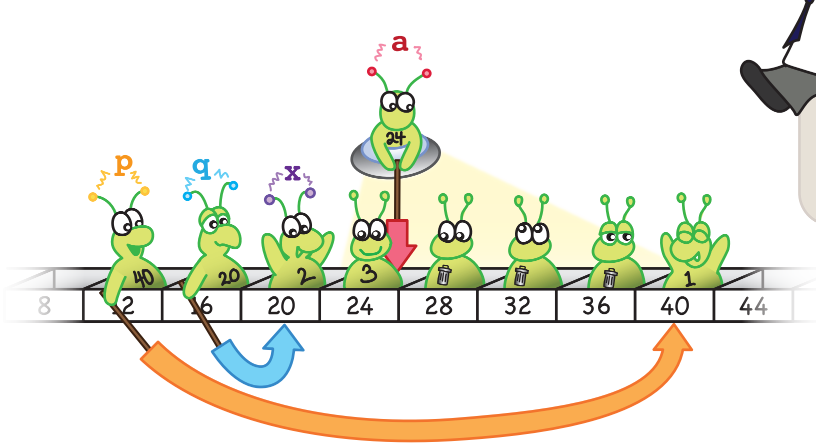
In the illustrations above, none of these variables have been assigned values yet, so they contain "garbage" -- whatever had been stored into these blocks of memory beforehand. We change that with the code below.




```
p = (int*) malloc(sizeof(int));
q = &x;
```



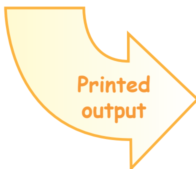
Now p contains, or **points** to, the address of a dynamically allocated memory space that can store one `int` value, and q points to the address of the variable x.




```
*p = 1;
*q = 2;
*a = 3;
```

When we **dereference** these pointers, we simply look inside the addresses that they point to. In this way, we can access the data stored there and even change those values. Here, we store the values of `1`, `2`, and `3` into the boxes that the addresses p, q, and a point to, respectively.

```
printf("*p:%u, p:%u, &p:%u\n", *p, p, &p);
printf("*q:%u, q:%u, &q:%u\n", *q, q, &q);
printf("*a:%u, a:%u, &a:%u\n", *a, a, &a);
```




```
*p:1, p:40, &p:12
*q:2, q:20, &q:16
*a:3, a:24, &a:24
```

One last thing before you go.... We said that the variable of an array doesn't live in memory. But, the system prints the address of the first index of the array as the address of a.