

## Chapter 2

### Instructions: Language of the Computer

**Part 4: ARM ISA, x86 ISA**  
**(MIPS 와의 대비)**

# Alternative Architectures

- ARM instructions
- x86 instructions

# Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

# The ARM Instruction Set

- Used as the example in chapters 2 and 3
- Most popular 32-bit instruction set in the world ([www.arm.com](http://www.arm.com))
- 4 Billion shipped in 2008
- Large share of embedded core market
  - Applications include mobile phones, consumer electronics, network/storage equipment, cameras, printers, ...
- Typical of many modern RISC ISAs
  - See ARM Assembler instructions, their encoding and instruction cycle timings in appendixes B1,B2 and B3 (CD-ROM)

# ARM & MIPS Similarities

- 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	15 × 32-bit	31 × 32-bit
Input/output	Memory mapped	Memory mapped



	Instruction name	ARM	MIPS
Register-register	Add	add	addu, addiu
	Add (trap if overflow)	adds; swivs	add
	Subtract	sub	subu
	Subtract (trap if overflow)	subs; swivs	sub
	Multiply	mul	mult, multu
	Divide	—	div, divu
	And	and	and
	Or	orr	or
	Xor	eor	xor
	Load high part register	—	lui
	Shift left logical	lsl <sup>1</sup>	sllv, sll
	Shift right logical	lsr <sup>1</sup>	srlv, srl
	Shift right arithmetic	asr <sup>1</sup>	srav, sra
	Compare	cmp, cmn, tst, teq	slt/i,slt/iu
Data transfer	Load byte signed	ldrsb	lb
	Load byte unsigned	ldrb	lbu
	Load halfword signed	ldrsh	lh
	Load halfword unsigned	ldrh	lhu
	Load word	ldr	lw
	Store byte	strb	sb
	Store halfword	strh	sh
	Store word	str	sw
	Read, write special registers	mrs, msr	move
	Atomic Exchange	swp, swpb	ll;sc

FIGURE 2.32 ARM register-register and data transfer instructions equivalent to MIPS core. Dashes mean the operation is not available in that architecture or not synthesized in a few instructions. If there are several choices of instructions equivalent to the MIPS core, they are separated by commas. ARM includes shifts as part of every data operation instruction, so the shifts with superscript 1 are just a variation of a move instruction, such as lsr<sup>1</sup>. Note that ARM has no divide instruction.

Addressing mode	ARM	MIPS
Register operand	X	X
Immediate operand	X	X
Register + offset (displacement or based)	X	X
Register + register (indexed)	X	—
Register + scaled register (scaled)	X	—
Register + offset and update register	X	—
Register + register and update register	X	—
Autoincrement, autodecrement	X	—
PC-relative data	X	—

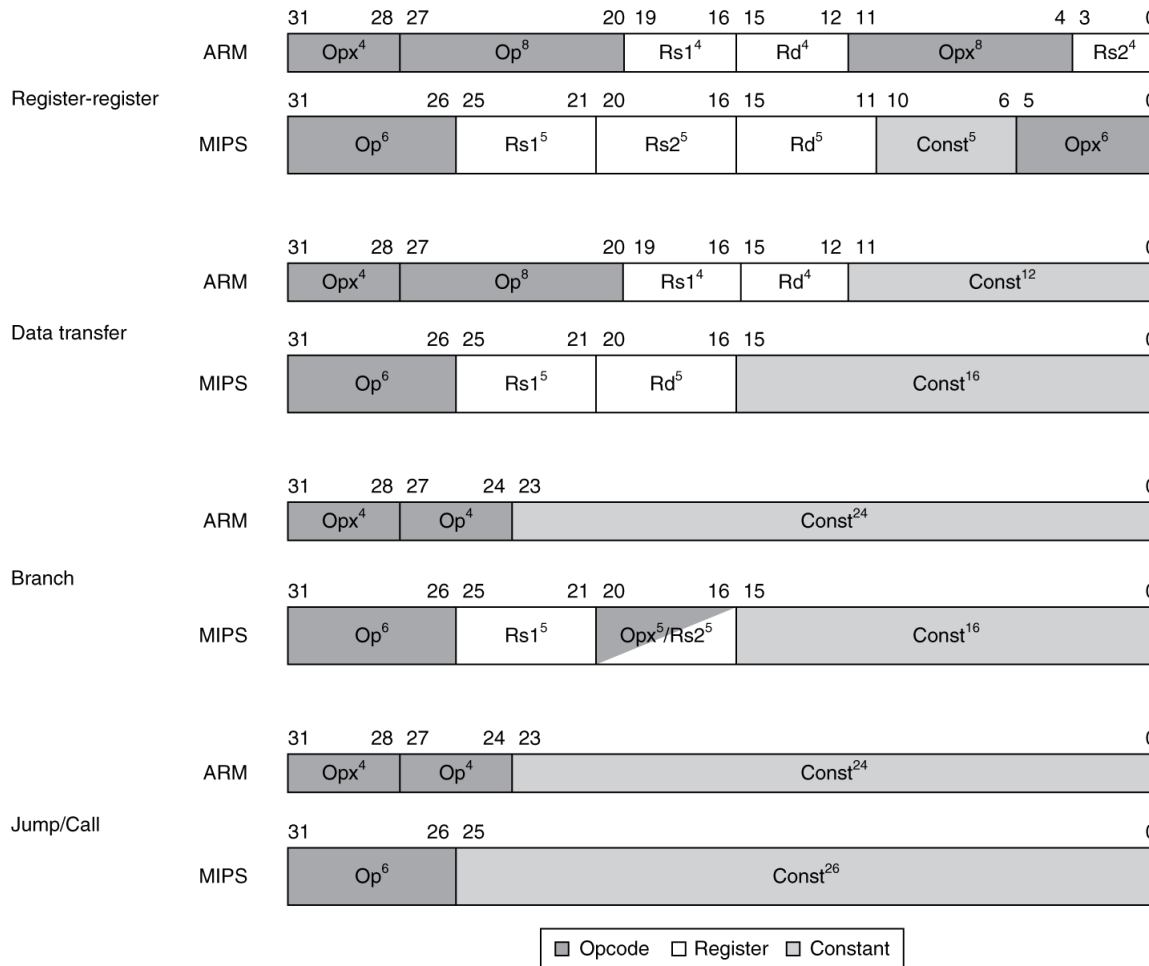
FIGURE 2.33 Summary of data addressing modes. ARM has separate register indirect and register 1 offset addressing modes, rather than just putting 0 in the offset of the latter mode. To get greater addressing range, ARM shifts the offset left 1 or 2 bits if the data size is halfword or word.

# 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



# Instruction Encoding



❖ 여기서  
slides 38 & 39를  
미리 보는 것이  
 좋음

# Branch Instruction format

Condition	12	address
4 bits	4 bits	24 bits

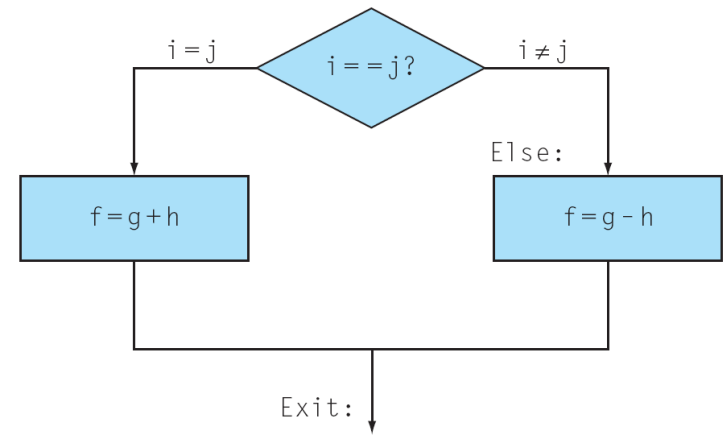
- Encoding of options for Condition field

Value	Meaning	Value	Meaning
0	EQ (Equal)	8	HI (unsigned Higher)
1	NE (Not Equal)	9	LS (unsigned Lower or Same)
2	HS (unsigned Higher or Same)	10	GE (signed Greater than or Equal)
3	LO (unsigned Lower)	11	LT (signed Less Than)
4	MI (MInus, <0)	12	GT (signed Greater Than)
5	PL - (PLus, >=0)	13	LE (signed Less Than or Equal)
6	VS (oVerflow Set, overflow)	14	AL (Always)
7	VC (oVerflow Clear, no overflow)	15	NV (reserved)

# Conditional Execution

- ARM code for executing *if* statement
- Code on right does not use branch. This can help performance of pipelined computers (Chapter 4)

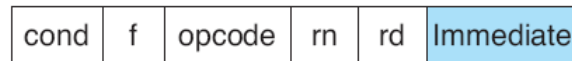
```
CMP r3, r4          ; reg r3 and r4
                    ; contain i,j
                    BNE Else      ; go to Else if i
<> j
                    ADD r0,r1,r2   ; f = g + h
                    B Exit        ; go to Exit
Else: SUB r0,r1,r2   ; f = g - h
Exit:
```



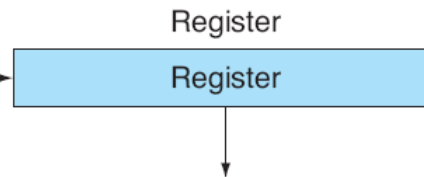
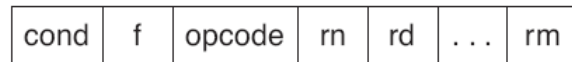
```
CMP r3,r4
ADDEQ r0,r1,r2 ; f = g+h
SUBNE r0,r1,r2 ; f = g-h
```

# Addressing Mode Summary (1-3 of 12)

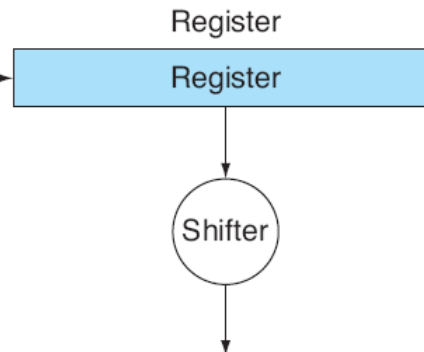
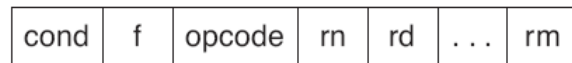
1. Immediate: ADD r2, r0, #5



2. Register: ADD r2, r0, r1

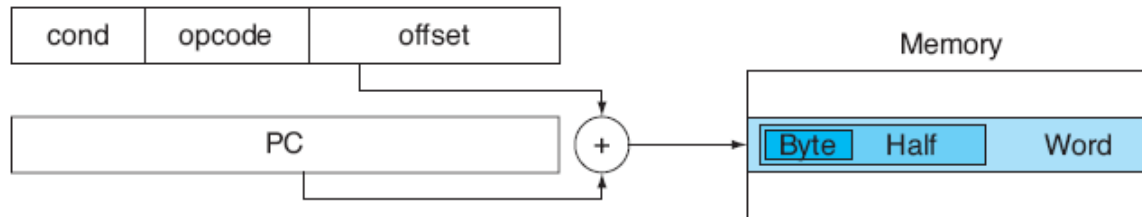


3. Scaled register: ADD r2, r0, r1, LSL #2

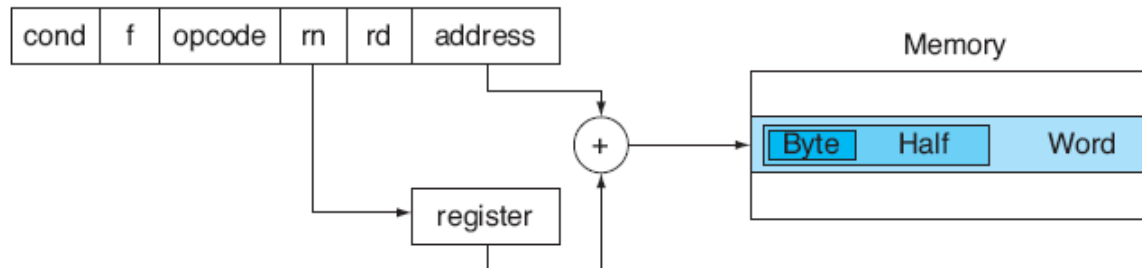


# Addressing Mode Summary (4-6 of 12)

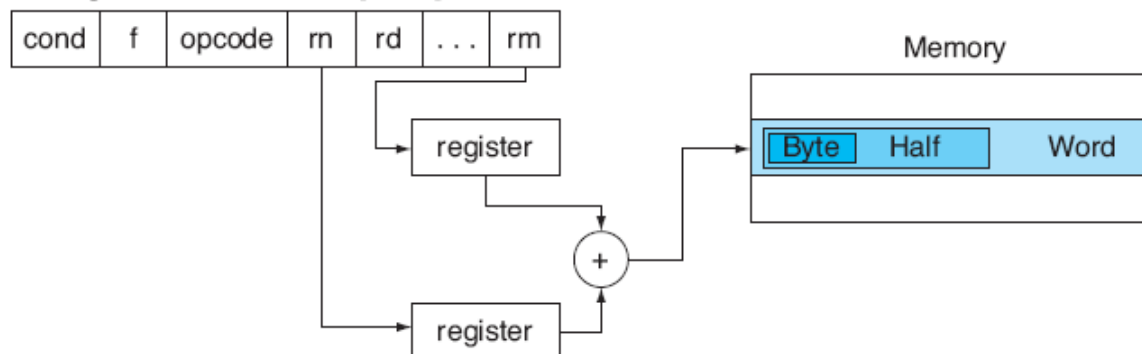
4. PC-relative: BEQ 1000



5. Immediate offset: LDR r2, [r0, #8]

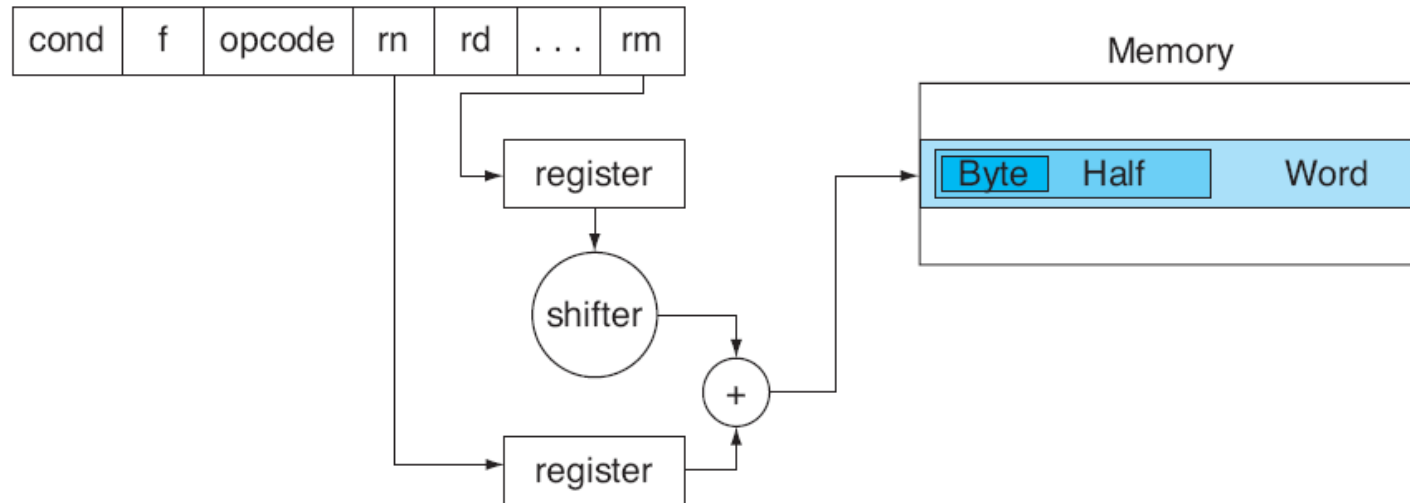


6. Register offset: LDR r2, [r0, r1]

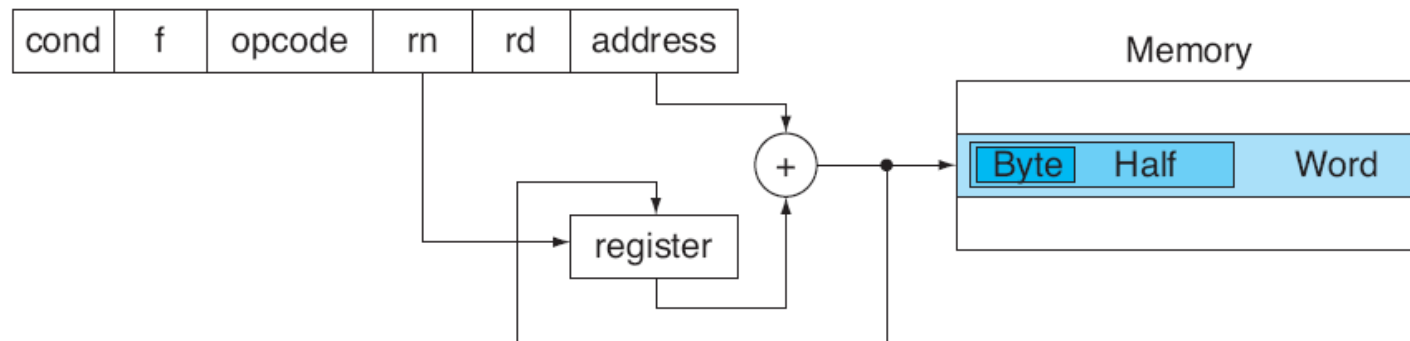


# Addressing Mode Summary (7-8 of 12)

7. Scaled register offset: LDR r2, [r0, r1, LSL #2]

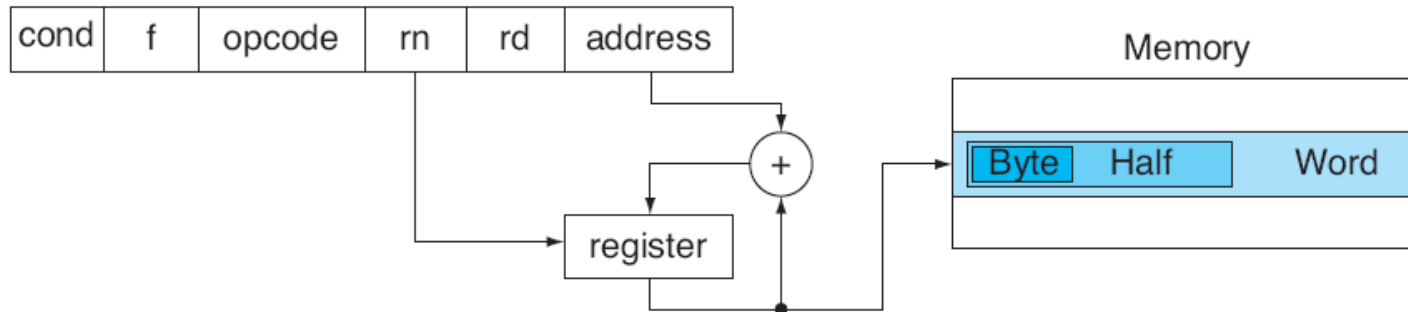


8. Immediate offset pre-indexed: LDR r2, [r0, #4]!

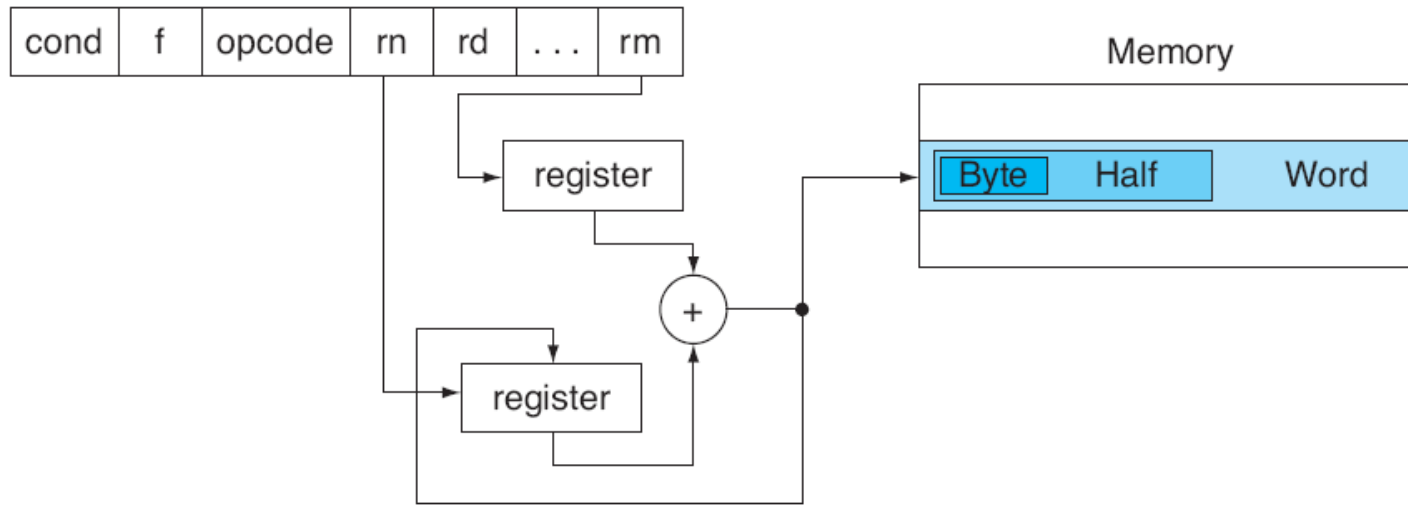


# Addressing Mode Summary (9-10 of 12)

9. Immediate offset post-indexed: LDR r2, [r0], #4

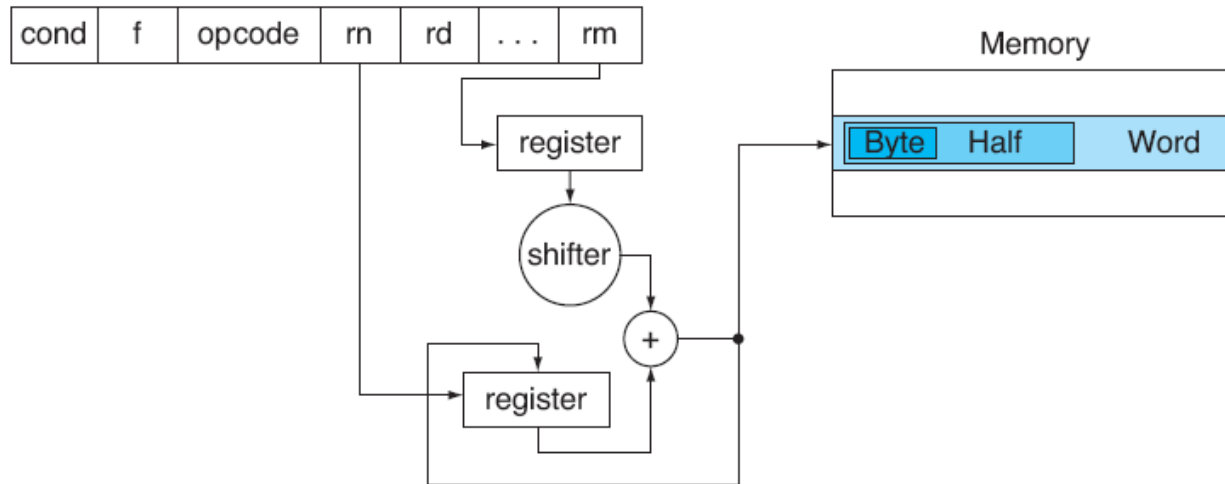


10. Register offset pre-indexed: LDR r2, [r0, r1]!

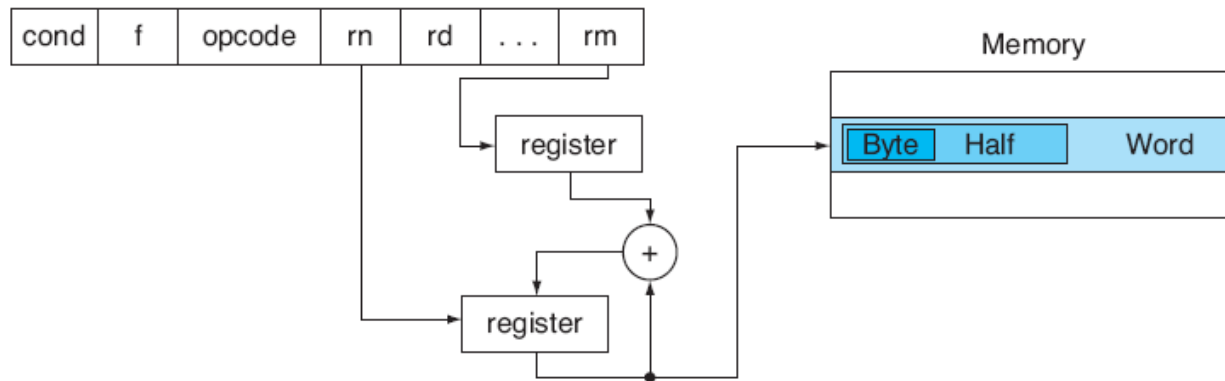


# Addressing Mode Summary (11-12)

11. Scaled register offset pre-indexed: LDR r2, [r0, r1, LSL #2]!



12. Register offset post-indexed: LDR r2, [r0], r1





# ARM v8 Instructions

- In moving to 64-bit, ARM did a complete overhaul
- ARM v8 resembles MIPS
  - Changes from v7:
    - No conditional execution field
    - Immediate field is 12-bit constant
    - Dropped load/store multiple
    - PC is no longer a GPR
    - GPR set expanded to 32
    - Addressing modes work for all word sizes
    - Divide instruction
    - Branch if equal/branch if not equal instructions

# 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



# 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

# 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

# Basic x86 Registers

Name	31	0	Use
EAX			GPR 0
ECX			GPR 1
EDX			GPR 2
EBX			GPR 3
ESP			GPR 4
EBP			GPR 5
ESI			GPR 6
EDI			GPR 7
	CS		Code segment pointer
	SS		Stack segment pointer (top of stack)
	DS		Data segment pointer 0
	ES		Data segment pointer 1
	FS		Data segment pointer 2
	GS		Data segment pointer 3
EIP			Instruction pointer (PC)
EFLAGS			Condition codes

# Basic x86 Addressing Modes

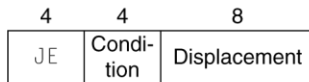
- Two operands per instruction

Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

- Memory addressing modes
  - Address in register
  - $\text{Address} = R_{\text{base}} + \text{displacement}$
  - $\text{Address} = R_{\text{base}} + 2^{\text{scale}} \times R_{\text{index}}$  (scale = 0, 1, 2, or 3)
  - $\text{Address} = R_{\text{base}} + 2^{\text{scale}} \times R_{\text{index}} + \text{displacement}$

# x86 Instruction Encoding

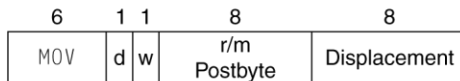
a. JE EIP + displacement



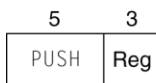
b. CALL



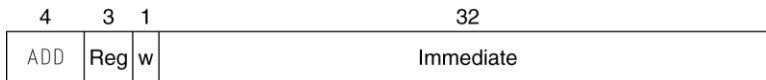
c. MOV EBX, [EDI + 45]



d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



## ■ Variable length encoding

- Postfix bytes specify addressing mode
- Prefix bytes modify operation
  - Operand length, repetition, locking, ...

# 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

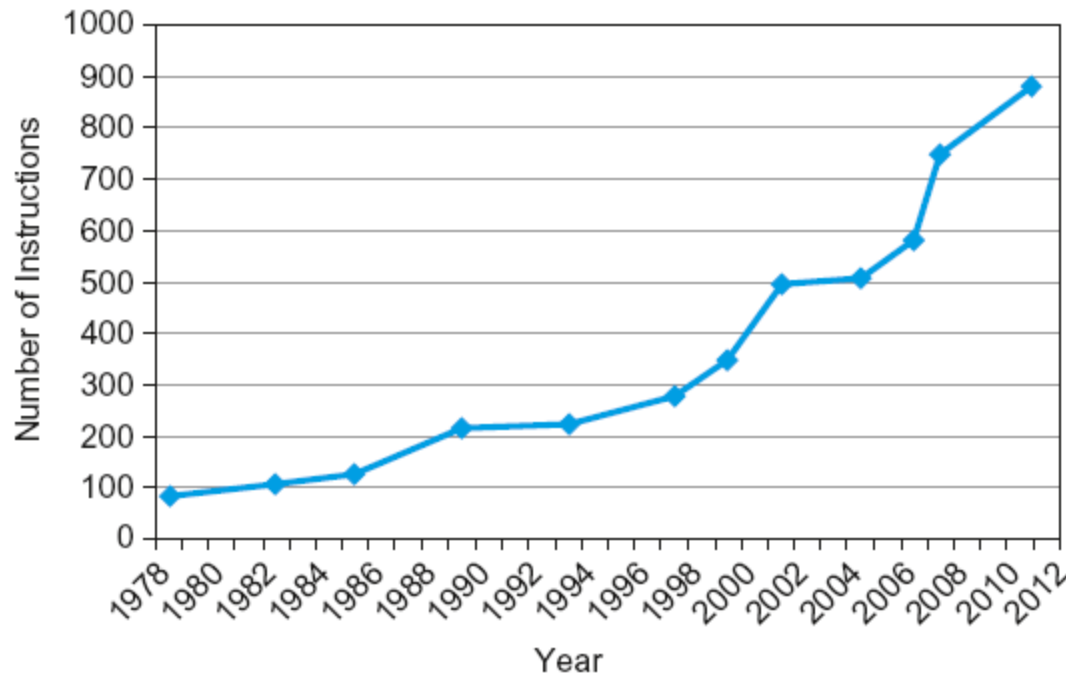


# Fallacies

- Powerful instruction  $\Rightarrow$  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  $\Rightarrow$  more errors and less productivity

# Fallacies

- Backward compatibility  $\Rightarrow$  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

# Concluding Remarks

- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises
- Layers of software/hardware
  - Compiler, assembler, hardware
- MIPS: typical of RISC ISAs
  - c.f. x86

# 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%