Unit 6 Array and Pointer

!!! * Midterm postponed to week 8 - Wed. July 1st

10 qns; closed book 6-9pm

§2.14 Arrays versus Pointers

- Array indexing involves
  - Multiplying index element by byte number
  - Adding to array base address
- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity

Example: Clearing and Array

Array

```
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
} // i->$t0, array[]->$a0, size->$a1
```

```
       move $t0,$zero   # i = 0
loop1: sll $t1,$t0,2    # $t1 = i * 4
       add $t2,$a0,$t1  # $t2=&array[i]
       sll $t4,$a1,2    # $t4 = size*4
       sw $zero, 0($t2) # array[i] = 0
       addi $t0,$t0,1   # i = i + 1
     * slt $t3,$t0,$t4  # set $t3 = 1
                        # if  (i<size)
       bne $t3,$zero,loop1 # if(i<size)
                           # goto loop1
```

$t0 is on base 1, but $t4 is base4 so cannot compare

Pointer

```
clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p < &array[size];
       p = p + 1)
    *p = 0;
} //p->$t0, array[0]->$a0, size->$a1
```

```
       move $t0,$a0     # p = & array[0]
       sll $t1,$a1,2    # $t1 = size * 4
       add $t2,$a0,$t1  # $t2 =
set to last element     #    &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 (…)
                           # goto loop2
```

Comparison of Array vs. Ptr

- Multiply can be done by left shift bit } the same
- Array version requires new member address to add to the base array address inside loop
  - Pointer version just add new member index inside the loop to get new pointer address;
  - Pointer doesn't need to add array base address to new member pointer address inside the loop

ARM & MIPS Similarities

good to know

§2.16 Real Stuff: ARM Instructions

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

	ARM v7: old v8: new	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 ← good to know	3 : R, I, J
Registers	15 × 32-bit	32 × 32-bit
Input/output	Memory mapped	Memory mapped

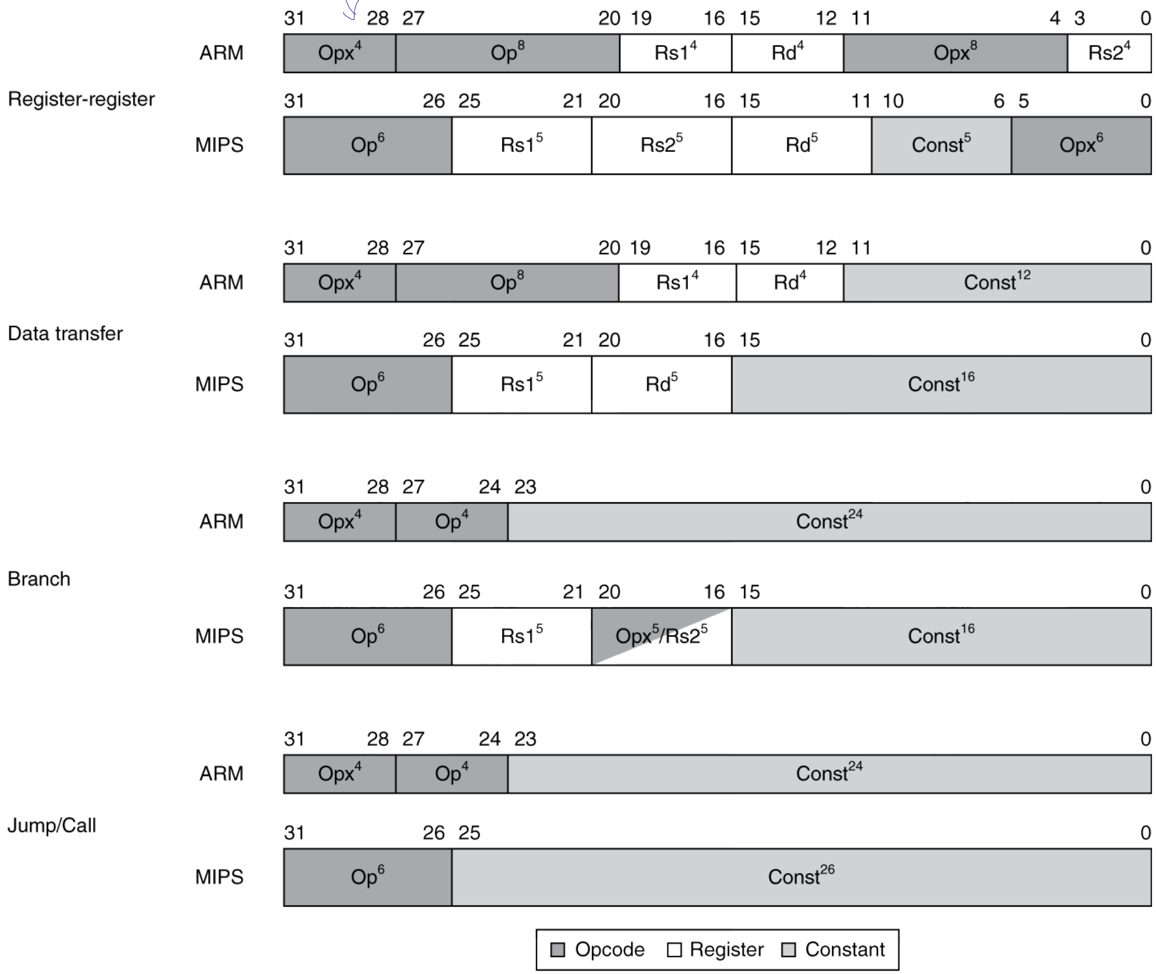
Compare and Branch in ARM

- Uses condition codes for result of an arithmetic/logical instruction
  - Negative, zero, carry, overflow
  - Compare instructions to set condition code
- Each instruction can be conditional
  - Top 4 bits of instruction word: condition value

ARM: 1st bit is condition code (for comment: no execution)
different from MIPS opcode

Instruction Encoding

conditional code



Register-register

ARM: 31–28 Opx⁴ | 27–20 Op⁸ | 19–16 Rs1⁴ | 15–12 Rd⁴ | 11–4 Opx⁸ | 3–0 Rs2⁴

MIPS: 31–26 Op⁶ | 25–21 Rs1⁵ | 20–16 Rs2⁵ | 15–11 Rd⁵ | 10–6 Const⁵ | 5–0 Opx⁶

Data transfer

ARM: 31–28 Opx⁴ | 27–20 Op⁸ | 19–16 Rs1⁴ | 15–12 Rd⁴ | 11–0 Const¹²

MIPS: 31–26 Op⁶ | 25–21 Rs1⁵ | 20–16 Rd⁵ | 15–0 Const¹⁶

Branch

ARM: 31–28 Opx⁴ | 27–24 Op⁴ | 23–0 Const²⁴

MIPS: 31–26 Op⁶ | 25–21 Rs1⁵ | 20–16 Opx⁵/Rs2⁵ | 15–0 Const¹⁶

Jump/Call

ARM: 31–28 Opx⁴ | 27–24 Op⁴ | 23–0 Const²⁴

MIPS: 31–26 Op⁶ | 25–0 Const²⁶

▨ Opcode □ Register ▨ Constant

The Intel x86 ISA

10nm = Silicon atom size.

current 25nm transistor

§2.17 Real Stuff: x86 Instructions

- 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
  - Pentium Pro (1995), Pentium II (1997)
  - Pentium III (1999) for pipeline (SSE)
    - Added SSE (Streaming SIMD Extensions) and associated registers
  - Pentium 4 (2001)
    - New microarchitecture
    - Added SSE2 instructions

The Intel x86 ISA

AMD stock ↗$15, intel $12 4 cores (cpu).

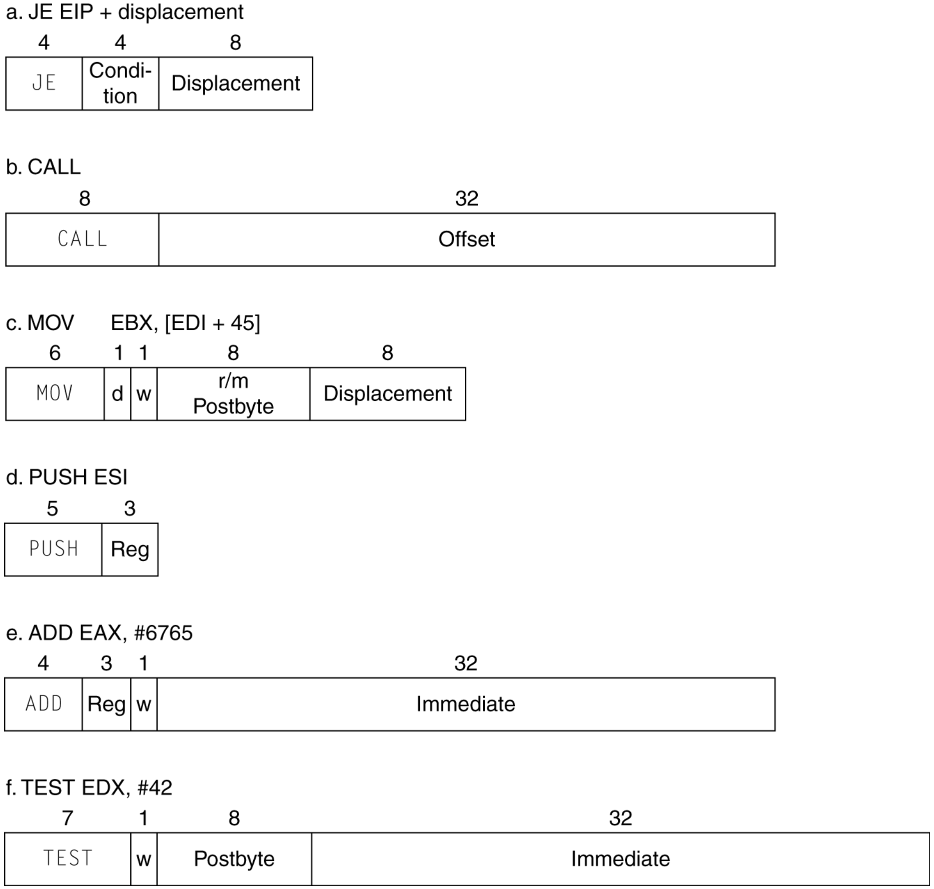
- And further…
  - AMD64 (2003): extended architecture to 64 bits
  - EM64T – Extended Memory 64 Technology (2004)
    - AMD64 adopted by Intel (with refinements)
    - Added SSE3 instructions
  - Intel Core (2006)
    - Added SSE4 instructions, virtual machine support
  - AMD64 (announced 2007): SSE5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

Basic x86 Addressing Modes

- Two operands per instruction

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

x86 Instruction Formats Samples



a. JE EIP + displacement

4	4	8
JE	Condition	Displacement

b. CALL

8	32
CALL	Offset

c. MOV EBX, [EDI + 45]

6	1	1	8	8
MOV	d	w	r/m Postbyte	Displacement

d. PUSH ESI

5	3
PUSH	Reg

e. ADD EAX, #6765

4	3	1	32
ADD	Reg	w	Immediate

f. TEST EDX, #42

7	1	8	32
TEST	w	Postbyte	Immediate

Implementing IA-32

- Complex instruction set makes implementation difficult
  - Hardware translates instructions to simpler microoperations
  - Similar to RISC
  - Market share makes this economically viable
- Comparable performance to RISC

ARM v8 Instructions

(good to know)

§2.18 Real Stuff: ARM v8 (64-bit) Instructions

- In moving to 64-bit, ARM did a complete overhaul
- ARM v8 resembles MIPS
  - Changes from v7:
    - No conditional execution field (ie, No condition code)
    - Immediate field is 12-bit constant
    - Dropped load/store multiple
    - Addressing modes work for all word sizes
    - Divide instruction
    - Branch if equal/branch if not equal instructions

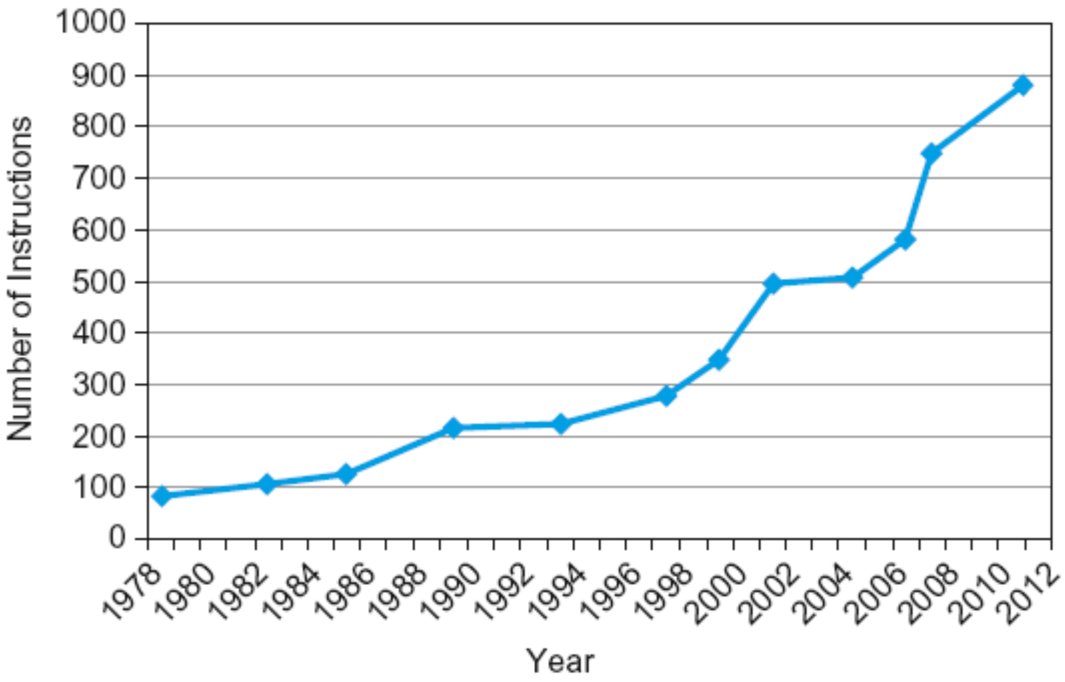
Fallacies

- Powerful instruction ⇒ higher performance (not true)
  - 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 (not true)
  - But modern compilers are better at dealing with modern processors
  - More lines of code ⇒ more errors and less productivity

1 command
→ 1 move -

Fallacies

- Backward compatibility ⇒ instruction set doesn't change → wrong, they do change
  - But they do create more instructions



x86 instruction set

Pitfalls

- Sequential words are in sequential addresses ?
  - Increment by 4, not by 1!
- What are needed to make sure the subroutine procedures to return the values and control back to the calling main program ?
  - Main program uses $a0 to $a3 to pass arguments to the calling procedure
  - Subroutine uses $v0 to $v1 to pass the values back to the main routine
  - Subroutine uses $ra to return control back to the calling main program

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

Concluding Remarks

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

Unit 6 Homework

1. Is it true that processor with more instructions mean higher performance ? Give examples to support your point.
2. What are the differences between array and pointer MIPS Assembly codes ?
3. What are the algorithms used in the MIPS instructions to make sure the main program can call the subroutine and the subroutine can return the values and control back to the main program ?