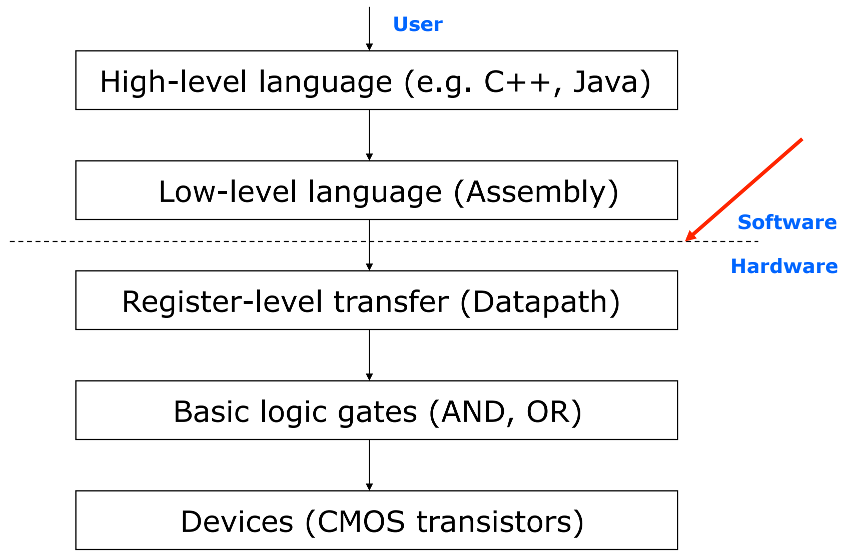
EEL-4713C Computer Architecture Instruction Set Architectures

Outline

- Instruction set architectures
- The MIPS instruction set
  - Operands and operations
  - Control flow
  - Memory addressing
  - Procedures and register conventions
  - Pseudo-instructions
- Reading:
  - Textbook, Chapter 2
  - Sections 2.1-2.8, 2.10-2.13, 2.17-2.20

Abstraction layers

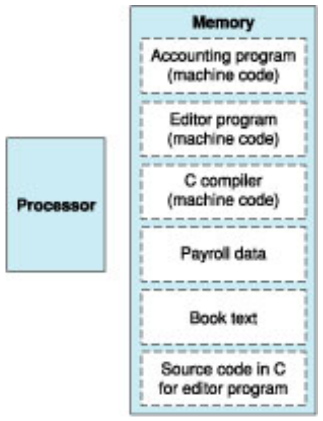

User
High-level language (e.g. C++, Java)
Low-level language (Assembly)
Software
Hardware
Register-level transfer (Datapath)
Basic logic gates (AND, OR)
Devices (CMOS transistors)

Introduction to Instruction Sets

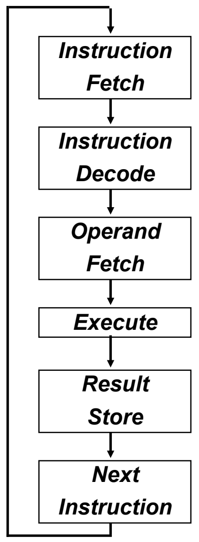
- Instructions: words of computer hardware's language
  - Instruction sets: vocabulary
  - What is available for software to program a computer
- Many sets exist; core functionality is similar
  - Support for arithmetic/logic operations, data flow and control
- We will focus on the MIPS set in class
  - Simple to learn and to implement
  - Hardware perspective will be the topic of Chapter 5
  - Current focus will be on software, more specifically instructions that result from compiling programs written in the C language

Stored-program concept

- Treat instructions as data
  - Same technology used for both


Processor
Memory
Accounting program (machine code)
Editor program (machine code)
C compiler (machine code)
Payroll data
Book text
Source code in C for editor program

Stored-program execution flow



Instruction Fetch	Obtain instruction from program storage
Instruction Decode	Determine required actions and instruction size
Operand Fetch	Locate and obtain operand data
Execute	Compute result value or status
Result Store	Deposit results in storage for later use
Next Instruction	Determine successor instruction

Basic issues and outline

- What operations are supported?
  - What operands do they use?
- How are instructions represented in memory?
- How are data elements represented in memory?
- How is memory referenced?
- How to determine the next instruction in sequence?

What operations are supported?

- “Classic” instruction sets:
- Typical “integer” arithmetic and logic functions:
  - Addition, subtraction
  - Division, multiplication
  - AND, OR, NOT, …
- Floating-point operations
  - Add, sub, mult, div, square root, exponential, …
- More recent add-ons:
  - Multi-media, 3D operations

MIPS operations

- See MIPS reference chart (green page of textbook) for full set of operations
- Most common: addition and subtraction
- MIPS assembly: add rd, rs, rt
  - register rd holds the sum of values currently in registers rs and rt

Operands

- In MIPS, operands for arithmetic and logic operations always come from registers
  - Other sets (e.g. Intel IA-32/x86) support memory operands
- Registers: fast memory within the processor datapath
  - Goal is to be accessible within a clock cycle
  - How many?
    - » Smaller is faster – typically only a few registers are available
    - » MIPS: 32 registers
  - How wide?
    - » 32-bit and 64-bit now common
    - » Evolved from 4-bit, 8-bit, 16-bit
    - » MIPS: both 32-bit and 64-bit. We will only study 32-bit.

Example

f = (g+h) – (i+j);

```
add $t0,$s1,$s2 # $t0 holds g+h
add $t1,$s3,$s4 # $t1 holds i+j
sub $s0,$t0,$t1 # $s0 holds f
```

(assume f=$s0, g=$s1, h=$s2, i=$s3, j=$s4)

Operands (cont)

- Operands need to be transferred from registers to memory (and vice versa)
- Data transfer instructions:
  - Load: transfer from memory to register
  - Store: transfer from register to memory
  - What to transfer?
    - » 32-bit integer? 8-bit ASCII character?
    - » MIPS: 32-bit, 16-bit and 8-bit
  - From where in memory?
    - » MIPS: 32-bit address needs to be provided
    - » addressing modes
  - Which register?
    - » MIPS: one out of 32 registers needs to be provided

Example

A[12] = h + A[8];

```
lw  $t0,32($s3) # $t0: A[8] (32=8*4bytes)
add $t0,$s2,$t0 # $t0 = h+A[8]
sw  $t0,48($s3) # A[12] holds final result
```

(assume A is an array of 32-bit/4-Byte integer elements.
Base address of array A is $s3. h=$s2)

Immediate operands

- Constants are commonly used in programming
  - E.g. 0 (false), 1 (true)
- Immediate operands:
  - Which instructions need immediate operands?
    - » MIPS: some of arithmetic/logic (e.g. add)
    - » Loads and stores
    - » Jumps (will see later)
  - Width of immediate operand?
    - » In practice, most constants are small
    - » MIPS: pack 16-bit immediate in instruction code
- Example: addi $s3, $s3, 4

Instruction representations

- Stored program: instructions are in memory
- Must be represented with some binary encoding
- Assembly language
  - mnemonics used to facilitate people to "read the code"
  - E.g. MIPS add $t0,$s1,$s2
- Machine language
  - Binary representation of instructions
  - E.g. MIPS 00000010001100100100000000100000
- Instruction format
  - Form of representation of an instruction
  - E.g. MIPS 00000010001100100100000000100000
    - » Red: "add" code; brown: "$s2"

MIPS instruction encoding fields

op	rs	rt	rd	shamt	funct

- op (6 bits): basic operation; "opcode"
- rs (5 bits): first register source operand
- rt (5 bits): second register source operand
- rd (5 bits): register destination
- shamt (5 bits): shift amount for binary shift instructions
- funct (6 bits): function code; select which variant of the "op" field is used. "function code"
- "R-type"
  - Two other types: I-type, J-type; will see later

Logical operations

- Bit-wise operations; packing and unpacking of bits into words
- MIPS:
  - Shift left/right
    - » E.g. sll $s1,$s2,10
  - Bit-wise AND, OR, NOT, NOR
    - » E.g. and $s1,$s2,$s3
  - Immediate AND, OR
    - » E.g. andi $s1,$s2,100
- What does andi $s1,$s1,0 do?

Decision-making: control flow

- A microprocessor fetches an instruction from memory address pointed by a register (PC)
- The PC is *implicitly incremented* to point to the next memory address in sequence after an instruction is fetched
- Software requires more than this:
  - Comparisons; if-then-else
  - Loops; while, for
- Instructions are required to change the value of PC from the implicit next-instruction
  - Conditional branches
  - Unconditional branches

MIPS control flow

- Conditional branches:
  - beq $s0,$s1,L1
    - » Go to statement labeled L1 if $s0 equal to $s1
  - bne $s0,$s1,L1
    - » Go to statement labeled L1 if $s0 not equal to $s1
- Unconditional branches:
  - J L2
    - » Go to statement labeled L2

Example: if/then/else

if (i==j) f = g+h; else f=g-h;

```
    bne $s3,$s4,Else # go to else if i!=j
    add $s0,$s1,$s2 # f=g+h
    j Exit
Else: sub $s0,$s1,$s2
Exit:
```

($s3=i, $s4=j, $s1=g, $s2=h, $s0=f)

Example: while loop

While (save[i]==k) i=i+1;

```
Loop: sll $t1,$s3,2   # $t1 holds 4*i
      add $t1,$t1,$s6 # $t1:addr of save[i]
      lw  $t0,0($t1)  # $t0: save[i]
      bne $t0,$s5,Exit # not equal? end
      addi $s3,$s3,1  # increment I
      j Loop         # loop back
Exit:
```

($s3=i, $s5=k, $s6 base address of save[])

MIPS control flow

- <u>**Important note:**</u>
  - MIPS register $zero is not an ordinary register
    - » It has a fixed value of zero
  - A special case to facilitate dealing with the zero value, which is commonly used in practice
- E.g. MIPS does not have a branch-if-less-than
  - Can construct it using set-less-than (slt) and register $zero:
  - E.g.: branch if $s3 less than $s2
    - » slt $t0,$s3,$s2   # $t0=1 if $s3<$s2
    - » bne $t0,$zero,target   # branch if $t0 not equal to zero

MIPS control flow: supporting procedures

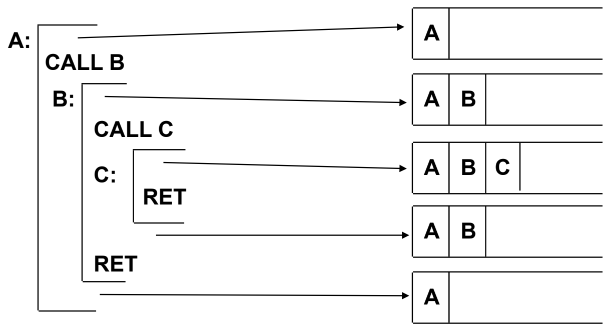
- Instruction "jump-and-link" (jal JumpAddr)
  - Jump to 26-bit immediate address JumpAddr
    - » Used when calling a subroutine
  - Set R31 ($ra) to PC+4
    - » Save return address (next instruction after procedure call) in a specific register
- Instruction "jump register" (jr $rx)
  - Jump to address stored in address $rx
  - jr $ra: return from subroutine

Support for procedures

- Handling arguments and return values
  - $a0-$a3: registers used to pass parameters to subroutine
  - $v0-$v1: registers used to return values
  - Software convention – these are general-purpose registers
- How to deal with registers that procedure body needs to use, but caller does not expect to be modified?
  - E.g. in nested/recursive subroutines
- Memory "stacks"
  - Placeholder for register values that need to be preserved during procedure call

Procedure calls and stacks

Stacking of Subroutine Calls & Returns and Environments:



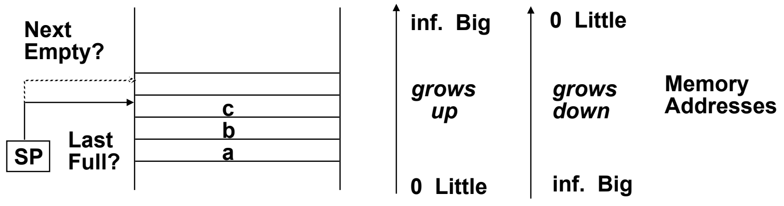
Some machines provide a memory stack as part of the architecture (e.g., VAX)

Sometimes stacks are implemented via software convention (e.g., MIPS)

Memory Stacks

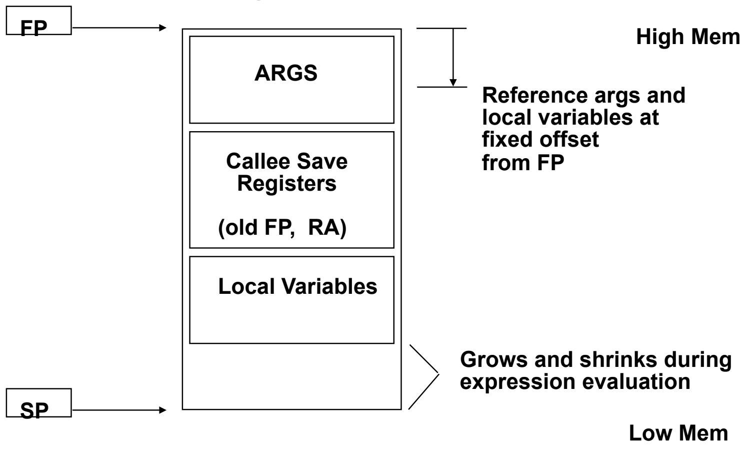
Useful for stacked environments/subroutine call & return even if operand stack not part of architecture

Stacks that Grow Up vs. Stacks that Grow Down:



Little --> Big/Last Full		Little --> Big/Next Empty	
POP:	Read from Mem(SP) Decrement SP	POP:	Decrement SP Read from Mem(SP)
PUSH:	Increment SP Write to Mem(SP)	PUSH:	Write to Mem(SP) Increment SP

Call-Return Linkage: Stack Frames



SP may change during the procedure; FP provides a stable reference to local variables, arguments

MIPS: Software conventions for Registers

0	zero	constant 0	16	s0	callee saves
1	at	reserved for assembler	. . .		(caller can clobber)
2	v0	expression evaluation &	23	s7	
3	v1	function results	24	t8	temporary (cont'd)
4	a0	arguments	25	t9	
5	a1		26	k0	reserved for OS kernel
6	a2		27	k1	
7	a3		28	gp	Pointer to global area
8	t0	temporary: caller saves	29	sp	Stack pointer
. . .		(callee can clobber)	30	fp	frame pointer
15	t7		31	ra	Return Address (HW)

See Figure 2.18.

Example in C: swap

```
swap(int v[], int k)
{
   int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

swap: MIPS

Using saved registers ($a0=v[], $a1=k):

```
swap:
addi     $sp,$sp,-12 ; room for 3 words
sw       $s0,8($sp)
sw       $s1,4($sp)
sw       $s2,0($sp)
sll      $s1, $a1,2         ; multiply k by 4
addu     $s1, $a0,$s1       ; address of v[k]
lw       $s0, 0($s1)        ; load v[k]
lw       $s2, 4($s1)        ; load v[k+1]
sw       $s2, 0($s1)        ; store v[k+1] into v[k]
sw       $s0, 4($s1)        ; store old v[k] into v[k+1]
lw       $s0,8($sp)
lw       $s1,4($sp)
lw       $s2,0($sp)
addi     $sp,$sp,12 ; restore stack pointer
jr $ra              ; return to caller
```

```
swap(int v[], int k)
{
   int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

swap: MIPS

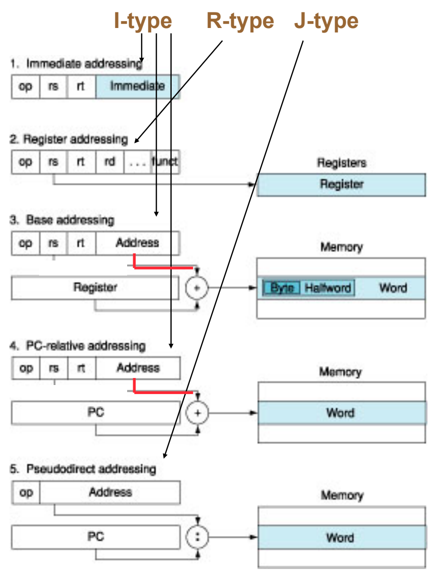
Using temporaries ($a0=v[], $a1=k)

```
swap:
sll      $t1, $a1,2         ; multiply k by 4
addu     $t1, $a0,$t1       ; address of v[k]
lw       $t0, 0($t1)        ; load v[k]
lw       $t2, 4($t1)        ; load v[k+1]
sw       $t2, 0($t1)        ; store v[k+1] into v[k]
sw       $t0, 4($t1)        ; store old v[k] into v[k+1]
jr $ra              ; return to caller
```

```
swap(int v[], int k)
{
   int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

MIPS Addressing modes

- Common modes that compilers generate are supported:
  - Immediate
    - » 16 bits, in inst
  - Register
    - » 32-bit register contents
  - Base
    - » Register + constant offset; 8-, 16- or 32-bit data in memory
  - PC-relative
    - » PC+constant offset
  - Pseudo-direct
    - » 26-bit immediate, shifted left 2x and concatenated to the 4 MSB bits of the PC



MIPS Addressing: 32-bit constants

- All MIPS instructions are 32-bit long
  - Reason: simpler, faster hardware design: instruction fetch, decode, cache
- However, often 32-bit immediates are needed
  - For constants and addresses
- Loading a 32-bit constant to register takes 2 operations
  - Load upper (a.k.a. most-significant, MSB) 16 bits (“lui” instruction)
    - » Also fills lower 16 bits with zeroes
    - » lui $s0,0x40 results in $s0=0x4000
  - Load lower 16 bits (“ori” instruction, or immediate)
    - » e.g. ori $s0,$s0,0x80 following lui above results in $s0=0x4080

MIPS Addressing: targets of jumps/branches

- Conditional branches:
  - 16-bit displacement relative to current PC
    - » I-type instruction, see reference chart
  - “Back” and “forth” jumps supported
    - » Signed displacement; positive and negative
  - “Short” conditional branches suffice most of the time
    - » E.g. small loops (back); if/then/else (forward)
- Jumps:
  - For “far” locations
  - 26-bit immediate, J-type instruction
  - Shifted left by two (word-aligned) -> 28 bits
  - Concatenate 4 MSB from PC -> 32 bits

Instructions for synchronization

- Multiple cores, multiple threads
- Synchronization is necessary to impose ordering
  - E.g.: a group working on a shared document
  - Two concurrent computations where there is a dependence
    - » A = (B + C) * (D + E)
    - » The additions can occur concurrently, but the multiplication waits for both
- Proper instruction set design can help support efficient synchronization primitives

Synchronization primitives

- Typically multiple cores share a single logical main memory, but each has its own register set
  - Or multiple processes in a single core
- “Locks” are basic synchronization primitives
  - Only one process “gets” a lock at a time
- Key insight: “atomic” read/write on memory location can be used to create locks
  - Goal: nothing can interpose between read/write to memory location
  - Cannot be achieved simply using regular loads and stores – why?
- Different possible approaches to supporting primitives in the ISA
  - Involving moving data between registers and memory

MIPS synchronization primitives

- Load linked (ll)
  - Load a value from memory to a register, like a regular load
  - But, in addition, hardware keeps track of the address from which it was loaded
- Store conditional (sc)
  - Store a value from register to memory succeeds *only if* no updates to load linked address
  - Register value also change: 0 if store failed, 1 if succeeded

Example

- Goal: build simple “lock”
  - Value “0” indicates it is free
  - Value “1” indicates it is not available
  - E.g. if a group is collaborating on the same document, an individual may only make changes if it successfully gets lock=0
- Primitive: atomic exchange $s4 and 0($s1)
  - Attempt to acquire a lock: exchange “1” with mem location 0($s1)

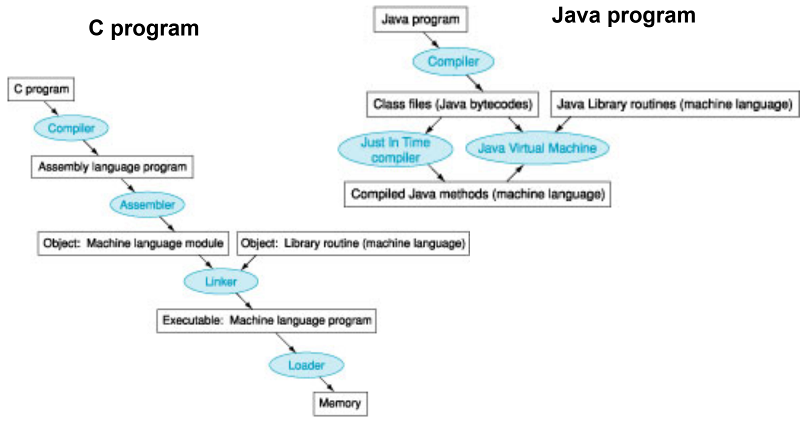
```
• Try: add $t0, $zero, $s4- $t0 gets $s4
•        ll $t1, 0($s1)          - load-linked lock addr
•        sc $t0, 0($s1)          - conditional store “1”
•        beq $t0,$zero,try       - if failed, $t0=0; retry
•        add $s4, $zero, $s1     - success: copy to $s4
```

Compiler, assembler, linker

- From high-level languages to machine executable program

C program

Java program


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory
Java program
Compiler
Class files (Java bytecodes)
Java Library routines (machine language)
Just In Time compiler
Java Virtual Machine
Compiled Java methods (machine language)

Compiler

- Translates high-level language program (source code) into assembly-level
  - E.g. MIPS assembly; Java bytecodes
- Functionality: check syntax, produce correct code, perform optimizations (speed, code size)
  - See 2.11 for more details

Assembler

- Translates assembly-level program into machine-level code
  - "Object" files (.o)
- Supports instructions of the processor's ISA, as well as "pseudo-instructions" that facilitate programming and code generation
  - Example: move $t0,$t1 a pseudo-instruction for add $t0,$zero,$t1
    - » Makes it more readable
  - Other examples: branch on less than (blt), load 32-bit immediate
    - » "unfold" pseudo-instruction into more than 1 real instruction
  - Cost: one register ($at) reserved to assembler, by convention

Linker

- Large programs can generate large object files
- Multiple developers may be working on various modules of a program concurrently
  - Sensible to partition source code across multiple files
- In addition, many commonly used functions are available in libraries
  - E.g. disk I/O, printf, network sockets, …
- Linker: takes multiple independent object files and composes an "executable file"

Loader

- Brings executable file from disk to memory for execution
  - Allocates memory for text and data
  - Copies instructions and input parameters to memory
  - Initializes registers & stack
  - Jumps to start routine (C's "main()")
- Dynamically-linked libraries
  - Link libraries to executables, "on-demand", after being loaded
  - Often the choice for functions common to many applications
  - Why?
    - » Reduce size of executable files – disk & memory space saved
    - » Many executables can share these libraries
  - .DLL in Windows, .so (shared-objects) in Linux

Miscellaneous MIPS instructions

- Break
  - A breakpoint trap occurs, transfers control to exception handler
- Syscall
  - A system trap occurs, transfers control to exception handler
- coprocessor instructions
  - Support for floating point: discussed later
- TLB instructions
  - Support for virtual memory: discussed later
- restore from exception
  - Restores previous interrupt mask & kernel/user mode bits into status register
- load word left/right
  - Supports misaligned word loads
- store word left/right
  - Supports misaligned word stores

Details of the MIPS instruction set

- Register zero always has the value zero (even if you try to write it)
- Jump and link instruction puts the return address PC+4 into the link register
- All instructions change all 32 bits of the destination register (including lui, lb, lh) and all read all 32 bits of sources (add, sub, and, or, …)
- Immediate arithmetic and logical instructions are extended as follows:
  - logical immediates are zero extended to 32 bits
  - arithmetic immediates are sign extended to 32 bits
- The data loaded by the instructions lb and lh are extended as follows:
  - lbu, lhu are zero extended
  - lb, lh are sign extended
- Overflow can occur in these arithmetic and logical instructions:
  - add, sub, addi
  - it cannot occur in addu, subu, addiu, and, or, xor, nor, shifts, mult, multu, div, divu

Reduced and Complex Instruction Sets

- MIPS is one example of a RISC-style architecture
  - Reduced Instruction Set Computer
  - Designed from scratch in the 80's
- Intel's "IA-32" architecture (x86) is one example of a CISC architecture
  - Complex Instruction Set
  - Has been evolving over almost 30 years

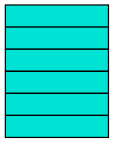
x86

- Example of a CISC ISA
  - P6 microarchitecture and subsequent implementations use RISC micro-operations
- Descended from 8086
- Most widely used general purpose processor family
  - Steadily gaining ground in high-end systems; 64-bit extensions now from AMD and Intel

Some history

- 1978: 8086 launched; 16-bit wide registers; assembly-compatible with 8-bit 8080
- 1982: 80286 extends address space to 24 bits (16MB)
- 1985: 80386 extends address space and registers to 32 bits (4GB); paging and protection for O/Ss
- 1989-95: 80486, Pentium, Pentium Pro; only 4 instructions added; RISC-like pipeline
- 1997-2001: MMX extensions (57 instructions), SSE extensions (70 instructions), SSE-2 extensions; 4 32-bit floating-point operations in a cycle
- 2003: AMD extends ISA to support 64-bit addressing, widens registers to 64-bit.
- 2004: Intel supports 64-bit, relabeled EM64T
- Ongoing: Intel, AMD extend ISA to support virtual machines (Intel VT, AMD Pacifica). Dual-core microprocessors.

x86 Registers



16-bit segment registers
CS, DS, SS, ES, FS, GS

32-bit General purpose registers
EAX, EBX, ECX, EDX,
EBP, ESI, EDI, ESP
Special uses for certain instructions
(e.g. EAX functions as accumulator,
ECX as counter for loops)

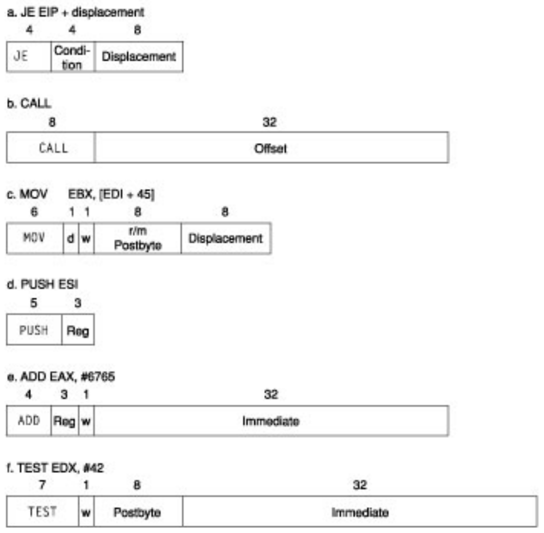
80-bit floating point stack
ST(0)-ST(7)

X86 operations

- Destination for operations can be register or memory
- Source can be register, memory or immediate

- Data movement: move, push, pop
- ALU operations
- Control flow: conditional branches, unconditional jumps, calls, returns
- String instructions: move, compare
  - MOVS: copies from string source to destination, incrementing ESI and EDI; may be repeated
  - Often slower than equivalent software loop

X86 encoding



RISC vs. CISC

- Long ago, assembly programming was very common
  - And memories were much smaller
  - CISC gives more programming power and can reduce code size
- Nowadays, most programming is done with high-level languages and compilers
  - Compilers do not use all CISC instructions
  - Simpler is better from an implementation standpoint – more on this during class
- Support for legacy codes and volume
  - Push for continued support of CISC ISAs like x86
- Compromise approach
  - Present CISC ISA to the ‘outside world’
  - Convert CISC instructions to RISC internally

Next lecture

- Introduction to the logic design process
  - Refer to slides and Appendix C, sections C.5-C.6