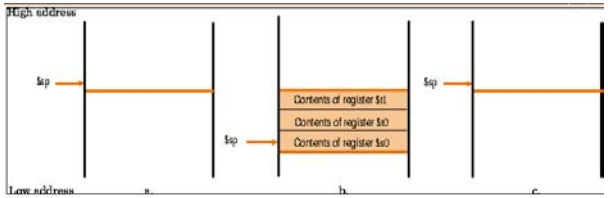
Other Issues

- support for procedures (Refer to section 3.6)
- stacks, frames, recursion
- manipulating strings and pointers
- linkers, loaders, memory layout
- interrupts and exceptions
- system calls and conventions

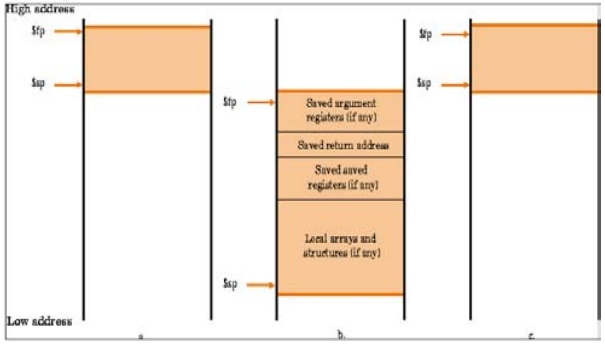
Stack Manipulation

- Register $29 is used as stack pointer
- Stack grows from high address to low address
- Stack pointer should point to the last filled address
- Once entries are removed, stack pointer should be adjusted



Frame Pointer

- Stores the last address for the last frame
- When completing a subroutine, frame address can be used as the starting stack pointer value



How about larger constants?

- We'd like to be able to load a 32 bit constant into a register
- Must use two instructions, new "load upper immediate" instruction

```
lui $t0, 1010101010101010
```

filled with zeros

1010101010101010	0000000000000000

- Then must get the lower order bits right, i.e.,

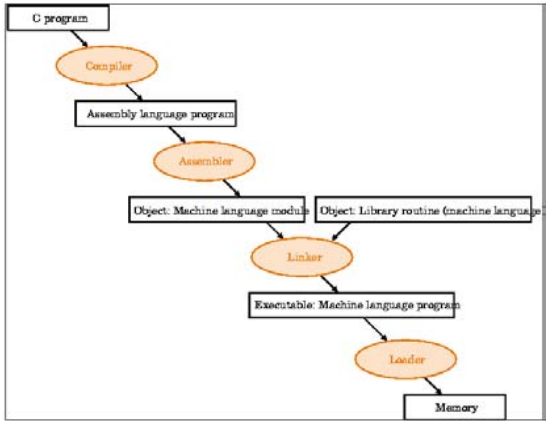
```
ori $t0, $t0, 1010101010101010
```

1010101010101010	0000000000000000
0000000000000000	1010101010101010
ori	
1010101010101010	1010101010101010

Assembly Language vs. Machine Language

- Assembly provides convenient symbolic representation
  - much easier than writing down numbers
  - e.g., destination first
- Machine language is the underlying reality
  - e.g., destination is no longer first
- Assembly can provide 'pseudoinstructions'
  - e.g., “move $t0, $t1” exists only in Assembly
  - would be implemented using “add $t0,$t1,$zero”
- When considering performance you should count real instructions

Sequence of Steps in Running a Program



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
- Sometimes referred to as “RISC vs. CISC”
  - virtually all new instruction sets since 1982 have been RISC
  - VAX: minimize code size, make assembly language easy
    instructions from 1 to 54 bytes long!
- We’ll look at PowerPC and 80x86

PowerPC

- Indexed addressing
  - example: `lw $t1,$a0+$s3 #$t1=Memory[$a0+$s3]`
  - What do we have to do in MIPS?
- Update addressing
  - update a register as part of load (for marching through arrays)
  - example: `lwu $t0,4($s3) #$t0=Memory[$s3+4];$s3=$s3+4`
  - What do we have to do in MIPS?
- Others:
  - load multiple/store multiple
  - a special counter register “`bc Loop`”
    decrement counter, if not 0 goto loop

80x86

- 1978: The Intel 8086 is announced (16 bit architecture)
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits, +instructions
- 1985: The 80386 extends to 32 bits, new addressing modes
- 1989-1995: The 80486, Pentium, Pentium Pro add a few instructions (mostly designed for higher performance)
- 1997: MMX is added

"This history illustrates the impact of the "golden handcuffs" of compatibility

"adding new features as someone might add clothing to a packed bag"

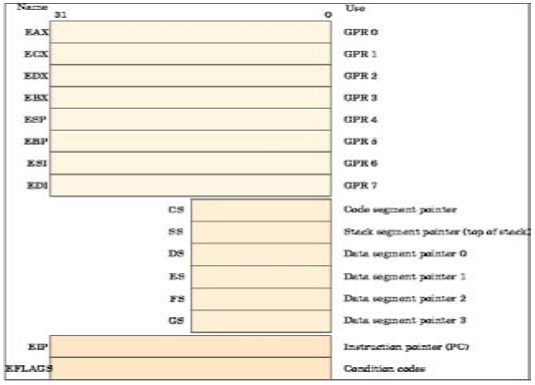
"an architecture that is difficult to explain and impossible to love"

A dominant architecture: 80x86

- 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"

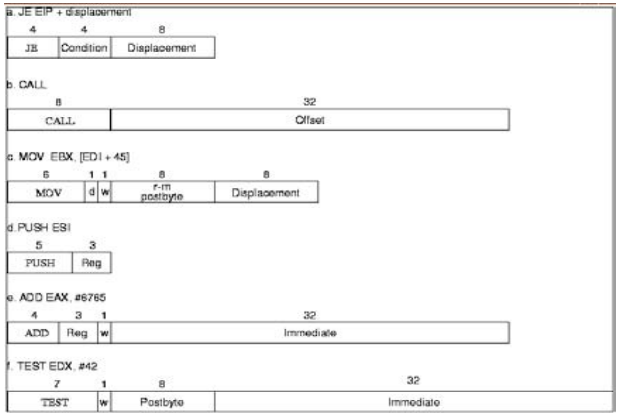
Registers in 80x86 Architecture



Examples of non-arithmetic instructions

Instruction	Function
JE name	If equal (CC) EIP = name; EIP−128 ≤ name < EIP+128
JMP name	(EIP = NAME);
CALL name	SP = SP − 4; M[SP] = EIP + 5; EIP = name;
MOVW EBX,[EDI + 45]	EBX = M [EDI + 45]
PUSH ESI	SP = SP − 4; M[SP] = ESI
POP EDI	EDI = M[SP]; SP = SP + 4
ADD EAX,#6765	EAX = EAX + 6765
TEST EDX,#42	Set condition codes (flags) with EDX & 42
MOVSL	M[EDI] = M[ESI]; EDI = EDI + 4; ESI = ESI + 4

Instruction Encoding

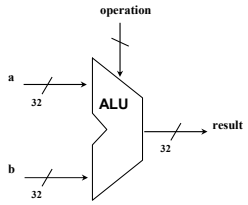


Summary

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

Arithmetic

- Where we've been:
  - Performance (seconds, cycles, instructions)
  - Abstractions:
    - Instruction Set Architecture
    - Assembly Language and Machine Language
- What's up ahead:
  - Implementing the Architecture



Numbers

- Bits are just bits (no inherent meaning)
  — conventions define relationship between bits and numbers
- Binary numbers (base 2)
  0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...
  decimal: $0...2^n-1$
- Of course it gets more complicated:
  numbers are finite (overflow)
  fractions and real numbers
  negative numbers
  e.g., no MIPS subi instruction; addi can add a negative number)
- How do we represent negative numbers?
  i.e., which bit patterns will represent which numbers?

Possible Representations

Sign Magnitude:	One's Complement	Two's Complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

- Issues: balance, number of zeros, ease of operations
- Which one is best? Why?

MIPS

- 32 bit signed numbers:

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = +1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = +2_{ten}$
...
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = +2{,}147{,}483{,}646_{ten}$ *maxint*
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2{,}147{,}483{,}648_{ten}$ *minint*
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -2{,}147{,}483{,}646_{ten}$
...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1_{ten}$

Two's Complement Operations

- Negating a two's complement number: invert all bits and add 1
  - remember: "negate" and "invert" are quite different!
- Converting n bit numbers into numbers with more than n bits:
  - MIPS 16 bit immediate gets converted to 32 bits for arithmetic
  - copy the most significant bit (the sign bit) into the other bits

```
0010  -> 0000 0010
1010  -> 1111 1010
```

  - "sign extension" (lbu vs. lb)

Addition & Subtraction

- Just like in grade school (carry/borrow 1s)

```
  0111        0111        0110
+ 0110      - 0110      - 0101
```

- Two's complement operations easy
  - subtraction using addition of negative numbers

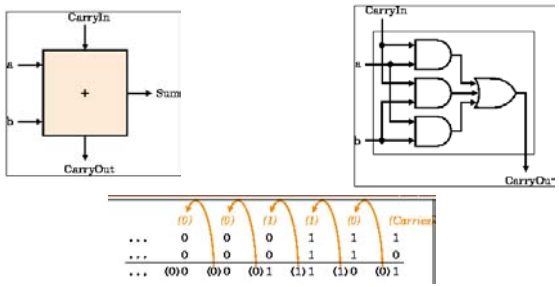
```
  0111
+ 1010
```

- Overflow (result too large for finite computer word):
  - e.g., adding two n-bit numbers does not yield an n-bit number

```
  0111
+ 0001      note that overflow term is somewhat misleading,
  1000      it does not mean a carry "overflowed"
```

One-Bit Adder

- Takes three input bits and generates two output bits
- Multiple bits can be cascaded



Adder Boolean Algebra

A	B	CI	CO	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

C = A.B + A.CI+ B.CI

S = A.B.CI + A'.B'.CI+A'.B.CI'+A.B'.CI'

Detecting Overflow

- No overflow when adding a positive and a negative number
- No overflow when signs are the same for subtraction
- Overflow occurs when the value affects the sign:
  - overflow when adding two positives yields a negative
  - or, adding two negatives gives a positive
  - or, subtract a negative from a positive and get a negative
  - or, subtract a positive from a negative and get a positive
- Consider the operations A + B, and A – B
  - Can overflow occur if B is 0 ?
  - Can overflow occur if A is 0 ?

Effects of Overflow

- An exception (interrupt) occurs
  - Control jumps to predefined address for exception
  - Interrupted address is saved for possible resumption
- Details based on software system / language
  - example: flight control vs. homework assignment
- Don't always want to detect overflow
  — new MIPS instructions: `addu, addiu, subu`

  note: `addiu` *still sign-extends!*
  note: `sltu, sltiu` *for unsigned comparisons*

Review: Boolean Algebra & Gates

- **Problem: Consider a logic function with three inputs: A, B, and C.**

  Output D is true if at least one input is true
  Output E is true if exactly two inputs are true
  Output F is true only if all three inputs are true

- **Show the truth table for these three functions.**
- **Show the Boolean equations for these three functions.**
- **Show an implementation consisting of inverters, AND, and OR gates.**

Real Design

A	B	C	D	E	F	
0	0	0	0	0	0	
0	0	1	1	0	0	
0	1	0	1	0	0	D = A + B + C
0	1	1	1	1	0	
1	0	0	1	0	0	E = A'.B.C + A.B'.C + A.B.C'
1	0	1	1	1	0	
1	1	0	1	1	0	F = A.B.C
1	1	1	1	0	1	