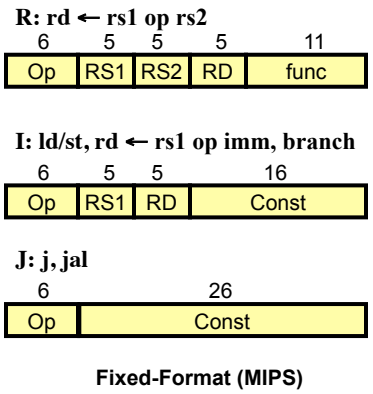
Lecture 7: Instruction Set Architectures IV

- Previously
  - Memory, Registers, ALU, & Control
  - MIPS instructions
- Today
  - Turn in Homework #2 and collect graded #1
  - Take QUIZ 4 before 11:59pm today over P&H 2.16-19
  - Quiz 3 grades: 100% - 9; 75%- 12; 50% - 9; 25% - 15; 0% - 8
  - ISA IV
    - A few more words on other ISAs
    - What about the Compiler?
    - Thinking like a compiler
      - MIPS assembly connected to high-level programming
      - Procedure calls and activations
    - The end of ISA

Instruction Formats: Can not be perfectly uniform

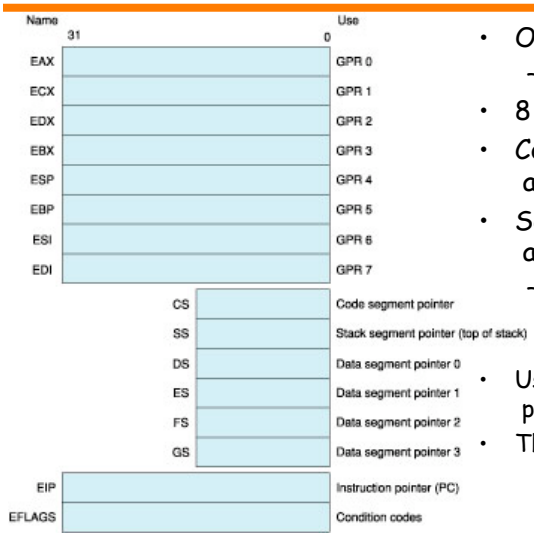
- MIPS uses a fixed size with three formats
- R instructions
  - 32 bit ALU operations
  - Addressing arithmetic
- I instructions
  - load/store
- J instructions
  - Jump, Jump Load/Link



Fixed-Format (MIPS)

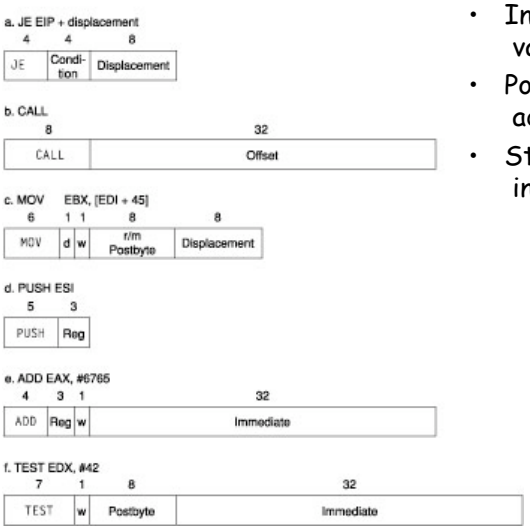
A Bit More on Other ISAs

Intel x86 Registers



- Originally 16 bits each
  - 386 extended to 32 bits
- 8 general purpose regs
- Condition code architecture
- Segment registers encode address prefix
  - Certain accesses use certain segment registers by default
- Used stack for for floating point
- Then added SSE floating point

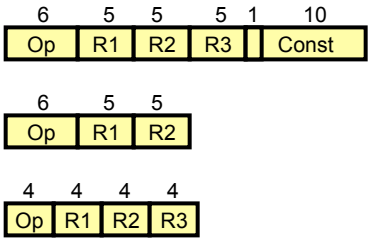
x86 Example Instruction Formats



- Instruction length varies a lot
- Postbyte encodes the addressing mode
- Stack access instructions

Compromise: A Few Good Formats

- Gives much better code density than fixed-format
  - important for embedded processors
- Simple to decode
- Examples:
  - ARM Thumb, MIPS 16
- Another approach
  - On-the fly instruction decompression (IBM CodePack)



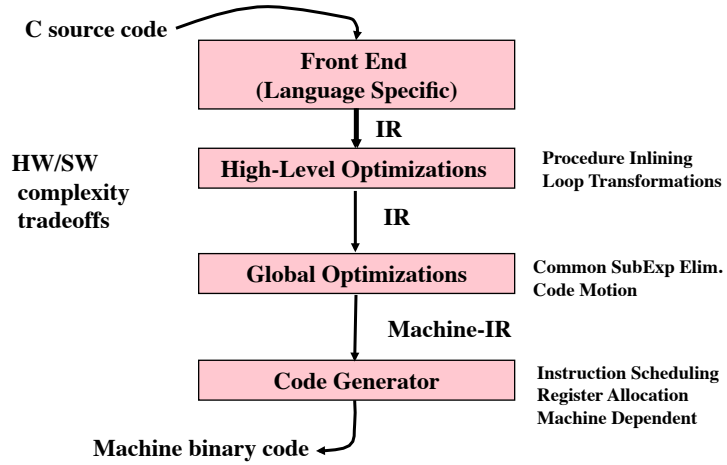
CISC vs RISC

- What is a RISC?
  (Reduced Instruction Set Computer)
  - no firm definition
  - generally includes
    - general registers
    - fixed 3-address instruction format
    - strict load-store architecture
    - operations on registers
    - simple addressing modes
    - simple instructions
  - Examples
    - DEC Alpha
    - MIPS
  - Advantages
    - good compiler target
    - easy to implement/pipeline
- CISC (Complex Instruction-Set Computer)
  - CISC ≡ ¬RISC
  - may include
    - variable length instructions
    - memory-register instructions
    - complex addressing modes
    - complex instructions
      - CALLP, EDIT, …
  - Examples
    - DEC VAX, IBM 370, x86
  - Advantages
    - better code density
    - legacy software

Compilers

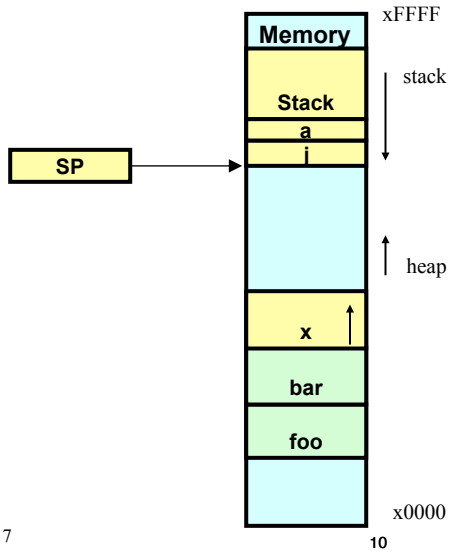
Performance = application +
compiler +
hardware

Role of the Optimizing Compiler



High Level Program in C

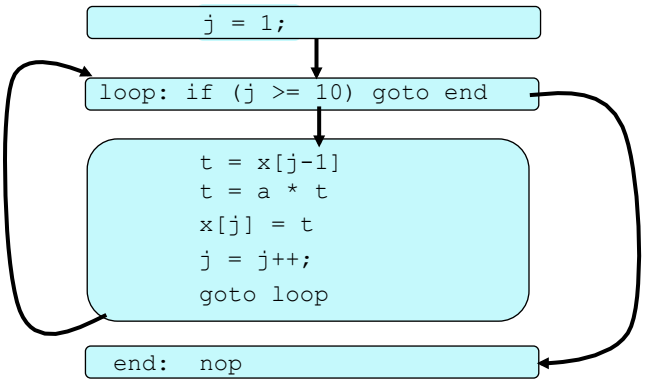
```
int x [10] ;
void foo(int a) {
  int j ;
  for(j=1; j<10; j++)
    x[j] = a * x [j-1];
  bar(a);
}
```



How do we translate high-level code to assembly?

```
for(j=1; j<10; j++)
  x [j] = a * x [j-1];
```

Lower representation closer to assembly



Control Flow Graph

control flow edges

basic blocks

Now lower it further or thinking like a compiler

```
       j = 1;
loop: if (j >= 10) goto end
       t = value (&x + 4 * (j - 1))   //t = x[j-1]
       t = a * t
       store at &x + (4 * j) value t // x [j] = t
       j = j++
       goto loop
end: nop
```

Thinking like a compiler: Set register to one

```
      addi $t0, $zero, 1 // $t0 = j = 1;
loop: slti $t1, $t0, 10    // if $t0 < 10 $t=1 (true)
      beq  $t1, $zero, end // if ($t1==0 (false) goto end
      subi $t2, $t0, 1   // $t2 = j - 1
      sll  $t2, $t2, 2   // $t2 = 4 * (j-1)
      lw   $t3, 0(&x)    // $t3 = &x
      add  $t4, $t3, $t2 // $t4 = addr x[j-1]
      lw   $t5, 0($t4)   // $t5 = val at x[j-1]
      lw   $t6, -4($sp)  // $t6 = a
      mul  $t5, $t6, $t5 // $t5 = a * x[j-1]
      sll  $t7, $t0, 2   // $t7 = j * 4
      add  $t2, $t3, $t7 // $t2 = addr x[j]
      sw   $t5, 0($t2)   // x[j] = $t4
      addi $t0, $t0, 1   // j++
      j loop
end: nop
```


Thinking like a compiler: Test and branch for end of loop

```
      addi $t0, $zero, 1 // $t0 = j = 1;
loop: slti $t1, $t0, 10    // if $t0 < 10 $t=1 (true)
      beq  $t1, $zero, end // if ($t1==0 (false) goto end
      subi $t2, $t0, 1   // $t2 = j - 1
      sll  $t2, $t2, 2   // $t2 = 4 * (j-1)
      lw   $t3, 0(&x)    // $t3 = &x
      add  $t4, $t3, $t2 // $t4 = addr x[j-1]
      lw   $t5, 0($t4)   // $t5 = val at x[j-1]
      lw   $t6, -4($sp)  // $t6 = a
      mul  $t5, $t6, $t5 // $t5 = a * x[j-1]
      sll  $t7, $t0, 2   // $t7 = j * 4
      add  $t2, $t3, $t7 // $t2 = addr x[j]
      sw   $t5, 0($t2)   // x[j] = $t4
      addi $t0, $t0, 1   // j++
      j loop
end: nop
```

Thinking like a compiler: Values versus Addresses

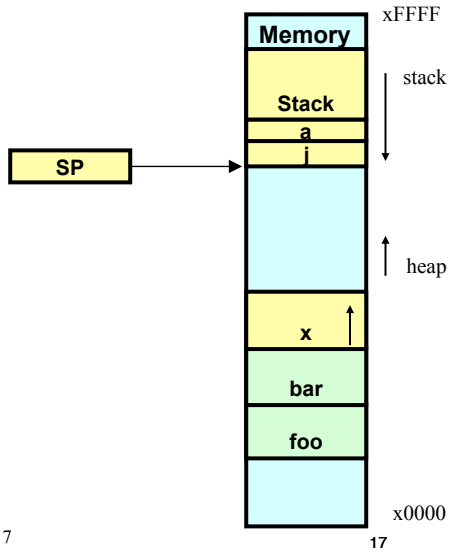
```
       addi $t0, $zero, 1 // $t0 = j = 1;
 loop: slti $t1, $t0, 10    // if $t0 < 10 $t=1 (true)
       beq  $t1, $zero, end // if ($t1==0 (false) goto end
       subi $t2, $t0, 1   // $t2 = j - 1
       sll  $t2, $t2, 2   // $t2 = 4 * (j-1)
       lw   $t3, 0(&x)    // $t3 = &x
       add  $t4, $t3, $t2 // $t4 = addr x[j-1]
       lw   $t5, 0($t4)   // $t5 = val at x[j-1]
       lw   $t6, -4($sp)  // $t6 = a
       mul  $t5, $t6, $t5 // $t5 = a * x[j-1]
       sll  $t7, $t0, 2   // $t7 = j * 4
       add  $t2, $t3, $t7 // $t2 = addr x[j]
       sw   $t5, 0($t2)   // x[j] = $t4
       addi $t0, $t0, 1   // j++
       j loop
 end:  nop
```


Example compiler optimization: Loop invariant code motion

```
       addi $t0, $zero, 1 // $t0 = j = 1;
 loop: slti $t1, $t0, 10    // if $t0 < 10 $t=1 (true)
       beq  $t1, $zero, end // if ($t1==0 (false) goto end
       subi $t2, $t0, 1   // $t2 = j - 1
       sll  $t2, $t2, 2   // $t2 = 4 * (j-1)
       lw   $t3, 0(&x)    // $t3 = &x
       add  $t4, $t3, $t2 // $t4 = addr x[j-1]
       lw   $t5, 0($t4)   // $t5 = val at x[j-1]
       lw   $t6, -4($sp)  // $t6 = a
       mul  $t5, $t6, $t5 // $t5 = a * x[j-1]
       sll  $t7, $t0, 2   // $t7 = j * 4
       add  $t2, $t3, $t7 // $t2 = addr x[j]
       sw   $t5, 0($t2)   // x[j] = $t4
       addi $t0, $t0, 1   // j++
       j loop
 end:  nop
```

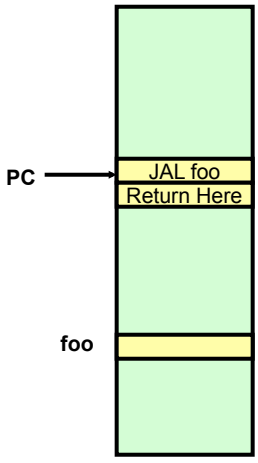
High Level Program in C

```
int x [10] ;
void foo(int a) {
  int j ;
  for(j=1; j<10; j++)
    x[j] = a * x [j-1];
  bar(a);
}
```



Support for Procedures

- Register Conventions
  - e.g., Store arguments in $a0, $a1, $a2, $a3
- Branch and Link
  - store return address in $ra and jump
    JALR Rdest: Rx ← PC + 4, PC ← Dest
- Subroutine call
  - push return address on stack and jump
- CALLP (VAX)
  - push return address
  - set up stack frame
  - save registers
  - ...



Procedure Calls on MIPS

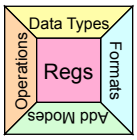
- Caller
  - Save $t0-$t9 if necessary
  - Save $fp
  - Push arguments on stack and leave in $a0-$a3
  - Set $fp to point to argument registers
  - Adjust $sp
  - Call JAL (sets $ra)
- Callee
  - Save $s0-$s7, $ra as necessary
  - Retrieve args, compute
  - Put return values in $v0,$v1
  - Jump to $ra
- Preserved State
  - Saved registers $s0-$s7
  - Stack pointer $sp
  - Return address $ra
- Not preserved
  - Temp registers $t0-$t9
  - Argument regs $a0-$a3
  - Return value regs $v0, $v1

Architect ⇒ Compiler Writer

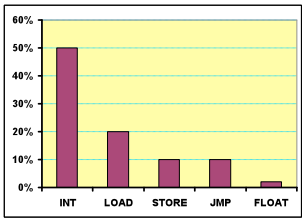
- Simplify, Simplify, Simplify
  - Feature difficult to use, it won't be used….Less is More!
- Regularity
  - Common set of formats, few special cases
- Primitive, not solutions
  - CALLS vs. Fast register moves
- Make performance tradeoffs simple
- Ultimately, the ISA will *not* be perfect

Principles of Instruction Set Design

- Keep it simple (KISS)
  - complexity
    - increases logic area
    - increases pipe stages
    - increases development time
  - evolution tends to make kludges
- Orthogonality (modularity)
  - simple rules, few exceptions
  - all ops on all registers



- Frequency
  - make the common case fast
    - some instructions are more important



- Regularity
  - principle of *least surprise*
  - performance should be easy to predict

Moving On

- That's it for ISAs in this course
- Other interesting ISA topics
  - ISAs that express concurrency explicitly
  - ISAs for signal processing and multimedia
  - ISAs for graphics
  - Vector processing ISAs

Summary

- Principals of ISA Design
- MIPS example
- Be one with the compiler!
- That's it for ISAs in this course
- Other interesting ISA topics
  - ISAs that express concurrency explicitly
  - ISAs for signal processing and multimedia
  - ISAs for graphics
  - Vector processing ISAs

- Next Time
  - Homework #3 is due 2/16
  - Reading: P&H 3