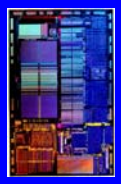
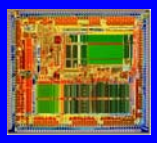
Instruction Set Architecture (Contd)

Lecture 3 (Chapter 2)
EEC170 FQ 2005



Intel 80486
1989
1200K Transistors
Size: 163mm²
50MHz



MIPS R3000
1988
110K Transistors
72mm²
20MHz
4watts

source: Prof. Kent Wilken

Review: MIPS Architecture

- Good example of *RISC* processor: Reduced Instruction-Set Computer
  - RISC really a misnomer: architecture goal is speed not small instruction set. However, certain simplicities result in short clock cycle
  - Alternate RISC definition: Relegate the Interesting Stuff to the Compiler
    - Avoid runtime overhead if complexity can be resolved at compile time
    - Old school view: software is hard, keep compiler simple
      RISC view: complier writers are sharp, have good modern development tools
- MIPS company spun off from Hennessy's MIPS processor project at Stanford
  - MIPS: Microprocessor without Interlocking Pipeline Stages
    - Designed for efficient pipelining (see Chapter 6)

Review: MIPS General Architecture Characteristics

- 32-bit integer registers ⇨ 32-bit architecture
  - 32-bit integer arithmetic
    - Higher precision must be done in software
  - Memory addresses up to 2^{32} = 4GBytes
  - MIPS-64 extends architecture to 64 bits
- 32-bit fixed-length instructions (for simplicity)
  - More complex operations done as a sequence of instructions
- Very few instruction formats (for simplicity)
- General Principle: Small/simple ⇨ Fast

Review: MIPS Arithmetic Operations

- Uses only 3 *specifier* arithmetic instructions:
  - Operation result, source1, source2
    E.g.,: ADD a, b, c ⇨ a = b + c
- Other architectures (e.g., x86) use 2 specifier instructions
  - One specifier is used as both a destination and source:
  - Operation destination/source1, source2
    E.g.,: ADD a, b ⇨ a = a + b

Review: 3 Specifier Instructions

- 3 specifier is the most general form, reduces instruction count vs. two specifier, e.g.:

```
a = b + c;   ⇨  ADD a, b, c     for 3 specifier
             ⇨  move a,b        for 2 specifier
                ADD a,c
```

Why do you suppose two-specifier was ever used?

Jump Register (`jr`)

- Unconditional jump to address contained in specified register, R-type format:

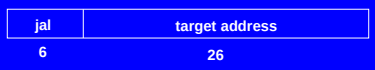
op	Rs	(not used)	jr
6	5		6

  - "Wasted" bits a non-issue: memory is cheap
- Allows branches to any of 2^{30} word addresses
  - (Bits 0-1 must be zero)
- Also used to implemented Case/Switch statements

```
lw $t1,0($t0) # load address from jump table
jr $t1
```

Subroutine Calls/Returns

- Some method needed for saving return address MIPS (other RISCs) use *Jump and Link* (`jal`)
- `jal` unconditionally jumps to specified address, Saves return address in $31 = $ra, *Link Register*
  - Return address in register is faster than memory
- J-type format



- Subroutine return achieved using `jr`:

```
jr $ra
```


Nested Subroutine Calls

- Need to handle nested subroutine calls
  - A calls B, B calls C, C calls D ...
  - Recursive subroutine calls (A conditionally calls A)
- Must avoid overwriting return address in $31 = $ra
- General solution: Store return address on stack
  - MIPS (other RISCs) does not have architected stack pointer
  - Any general purpose register could be used as stack pointer
  - Software convention uses $29 = $sp
  - Save return address, push ⇨

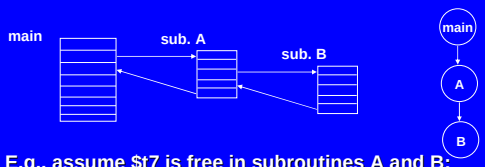
```
add $sp,$sp,-4 #expand stack by one word
sw  $ra,0($sp) #save ret. addr. on top of stack
```

  - Restore return address, pop ⇨

```
lw  $ra,0($sp) #get ret. addr. from top of stack
add $sp,$sp,4  #shrink stack by one word
```


Saving Return Address in Register

- Compiler can analyze *call graph* and register usage, possibly save return address in a free register rather than on stack (much faster!)



- E.g., assume $t7 is free in subroutines A and B:
  - In A, before call to B:
    `mov $t7,$ra # save return address in $t7`
  - To return from A:
    `jr $t7 # return address is in $t7, go for it`
  - Saves 3 instructions vs. stack: one for call, two for return

Synthesizing Large Constants

- RISC fixed-length instructions do not allow large immediate constants (e.g., 32 bits)
- MIPS uses special instruction in two-instruction sequence to create constants > 16 bits (uncommon case)
  - Load Upper Immediate (`lui`):

  - Sets Rd upper 16 bits to immediate data, lower 16 bits to 0s
  - 32 bit constant:

```
lui  $t0,1234hex      # $t0 has 12340000hex
addi $t0,$t0,5678hex  # $t0 has 12345678hex
```


Register 0

- Register 0 is special purpose
- Read from $0 always produces value 0
  - used to synthesize instructions beyond base set without added complexity
  - "move $7, $20" = add $7, $0, $20
  - "load $7, immediate" = addi $7, $0, immediate
  - Pseudo Instructions
- Write to $0 does nothing
  - add $0, $12, $20 = NOP
- Various other examples of exploiting $0 as we review instruction set

Instruction Set Design

- As seen, some operations can be synthesized with short sequence, e.g.:
  - Can also create 32-bit constant w/o `lui`:

```
addi $t0,$0,1234hex    # $t0 has 00001234hex
sll  $t0,$t0,16        # $t0 has 12340000hex
addi $t0,$t0,5678hex   # $t0 has 12345678hex
```

- How to decide if specific instruction (e.g., `lui`) is worthwhile?

Experimentation/Quantitative Analysis

Evaluating Instruction Sets?

Design-time metrics:

- ° Can it be implemented, in how long, at what cost?
- ° Can it be programmed? Ease of compilation?

Static Metrics:

- ° How many bytes does the program occupy in memory?

Dynamic Metrics:

- ° How many instructions are executed?
- ° How many bytes does the processor fetch to execute the program?
- ° How many clocks are required per instruction?
- ° How "lean" a clock is practical?

Best Metric: Time to execute the program!

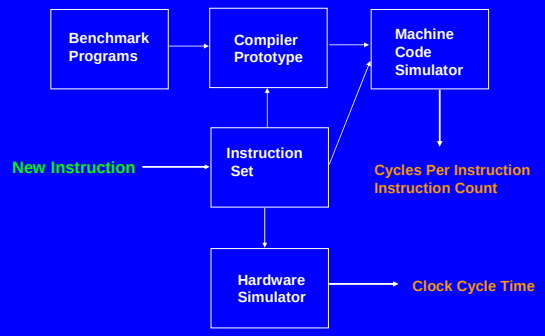
NOTE: this depends on instructions set, processor organization, and compilation techniques.

Aspects of CPU Performance

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

	instr count	CPI	clock rate
Program	X		
Compiler	X	X	
Instr. Set	X	X	
Organization		X	X
Technology			X

Architecture Development



Example Tradeoff: Auto Increment (Decrement)

- Some architectures (not most RISCs) include instruction that auto increments base/index register
  - MIPS:
```
lw   $t1,0($t0) # $t0 is index
addi $t0,$t0,4  # increment index
```
  - Auto Inc: `lw+  $t1,0($t0)  # load and inc index`
- Auto Inc (dec) may be useful for incrementing through arrays, but
  - not always increment by same value (e.g., 4)
  - instruction complicates hardware because produces two results (two register writes):
    - Value from memory
    - Updated index

Example Tradeoff: Memory Operations

- Some architectures (not RISC) allow arithmetic operands from memory
  - e.g., `ADDM $s0,$s1,offset($t0)`
- Problems:
  - Limited offset field (e.g., 11 bits)
  - requires 3 operations, hence very slow:
    1. compute address (r_base + offset)
    2. load data (memory is slow)
    3. ALU operation
  - significantly complicates pipeline design
    ⇨ slower clock cycle

Example Tradeoff: Dedicated Loop Counter

- A special loop-counter register is automatically decremented based on branch opcode
  - Branch if Loop-Counter = 0

Bcond	addr offset

- \+ Allows larger branch offset (lower instruction count, but small)
- ++ does not require separate increment instruction (lower instruction count)
- \- requires more work within branch instruction, may increase clock cycle time

Example Tradeoff: Variable-Length Instructions

+ Allows more compact code by using as many bits as needed
  - Move R1,R2

opcode	Rs	Rd

  - jump label

opcode	offset

opcode	offset

- Must decode the instruction before knowing how long, where operator specifiers can be found ⇨ higher CPI

Announcement: HW #1

- Chapter 2: problems 4, 5, 6, 9, 15, 28, 30, 31, 37, 38, 54, 55
  - Some are quick and easy questions.
- Due on Friday 10/21 at 5pm
- EEC 170 homework box

Compiler Optimization

Chapter 2.11-2.12
EEC170 FQ 2005

Example: HLL MIPS Assembly

C Code:

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

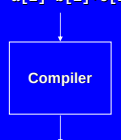
MIPS Code:

```
swap:  addi  $sp, $sp, -12     # make room on stack for 3 regs
       sw    $t0, 0($sp)       # save $t0 on stack
       sw    $s0,4($sp)        # save $s0 on stack
       sw    $s1, 8($sp)       # save $s1 on stack
       muli  $t0, $a0, 4       # $t0 = k * 4
       add   $t0,$a1, $t0      # $t0 = v + k*4 = the addr. of v[k]
       lw    $s0, 0($t0)       # $s0 = temp = v[k]
       lw    $s1, 4($t0)       # $s1 = v[k+1] = next element of v
       sw    $s1, 0($t0)       # v[k] = v[k+1]
       sw    $s0, 4($t0)       # v[k+1] = temp
       lw    $t0, 0($sp)       # restore $t0 from stack
       lw    $s0, 4($sp)       # restore $s0 from stack
       lw    $s1, 8($sp)       # restore $s1 from stack
       addi  $sp, $sp, 12      # restore stack pointer
       jr    $ra               # return to calling routine
```


Compiling

- Translating a high level language program to a machine language program

```
for (i=0; i<j; i++)
   a[i]=b[i]+c[i];
```

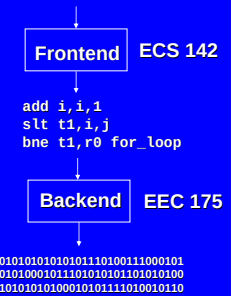



```
01001010101010101110100111000101
11001010001011101010101101010100
11111010101010100010010010101010
00001010111010101010001101010111
```


Compiling (cont.)

- Compilers have two major parts:
  - Frontend interacts with programmer/language
  - Backend interacts with architecture/processor
  - Frontend/Backend communicate using a generic language (*intermediate code*, *intermediate representation*)

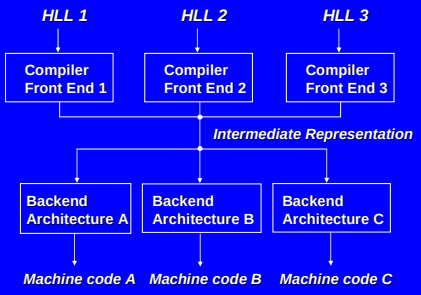
```
for (i=0; i<j; i++)
   a[i]=b[i]+c[i];
```




```
01001010101010101110100111000101
11001010001011101010101101010100
01010101010100010101111010010110
```

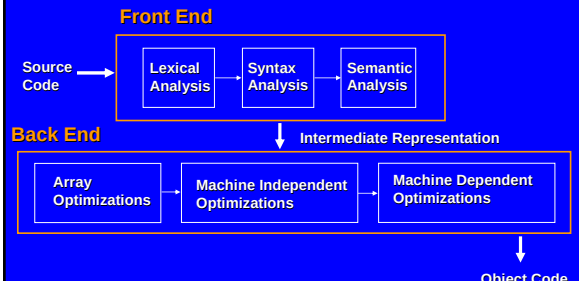
Compiler Retargeting

- Frontend/backend models allow easily retargeting of HLLs to different architectures



Compiler phases

- Compilers decompose the translation problem into smaller steps to manage the large complexity



Lexical Analysis

- Translates character string to string of *tokens*
  - Tokens are the basic lexical units in the HLL, e.g.:
    - A keyword: *Then*
    - A constant: *17*
    - An identifier: *newdata*
    - An operator: >=
    - Punctuation: ;
    - etc.
  - Token type and value are identified

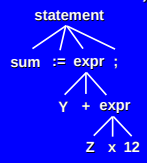
    IF (New > MaxNum) ... => token string:

    (Keyword, IF)
    (Identifier, "New")
    (Operator, >)
    (Identifier, "MaxNum")

Syntax Analysis (Parsing)

- Groups tokens into syntactic structure, in the form of a syntax tree (parse tree) to identify statements

sum := Y + Z x 12 ;



(Recall high school English)

Semantic Analysis

- Performs static error checking
  - Uses symbol table to check that variables are defined, declared
  - Checks Operand/Result variables for type compatibility
- Generates intermediate representation used by compiler backend
  - May resemble 'generic' assembly code, assumes unlimited registers

    Statement: S = A + B x C Becomes:

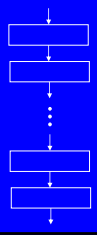
```
move R101,A
move R102,B
move R103,C
mult R104,R102,R103
add R105, R101,R104
move S,R105
```


Compiler Optimization Basics

- Usually optimizations only occur in the backend, frontend produces simple translation
- Backend attempts to improve code quality, some combination of code speed, size and possibly power consumption

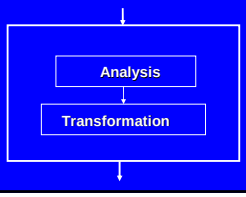
General Optimization Framework

- Backend is a series of optimization phases
  - Each phase lowers the intermediate representation toward the machine code and/or tries to improve the code quality



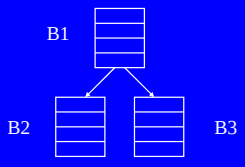
General Optimization Phase Structure

- Each optimization phase is generally structured as an analysis algorithm followed by a code-improving transformation algorithm
  - Analysis is specific to the transformation



Control Flow Analysis

- Compiler divides program into basic blocks (BB)
  - Straight-line code with no branch in except at start, no branch out except at end
  - Optimizations within BB are local, across BB are global



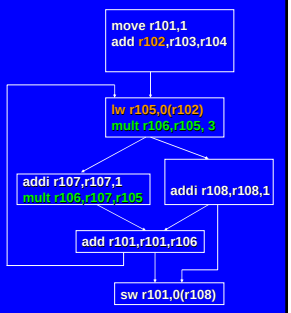
Instruction Scheduling

- The compiler can reorder (schedule) instructions to minimize delays from pipelining (see Chapter 6)
  - New schedule must produce same program output

```
LW    r101, 0(r102)                LW    r101, 0(r102)
<stall> <stall>                    LW    r107, 4(r102)
ADD   r104, r103,r101              <stall>
LW    r105, 0(r101)                LW    r105, 0(r101)
<stall> <stall>          ⇨         ADD   r104, r103,r101
SUB   r106, r105, r104             SW    r107, 0(r104)
LW    r107, 4(r102)                SUB   r106, r105, r104
<stall>
SW    r107, 0(r104)
```

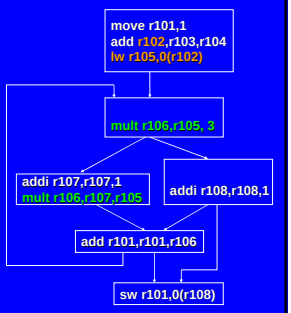

Loop Invariant Code Motion

- If instruction J's operands don't change during loop execution, J produces same result each loop iteration
- If J is also only instruction to modify its destination register, can move J outside the loop, only executed once per loop entry



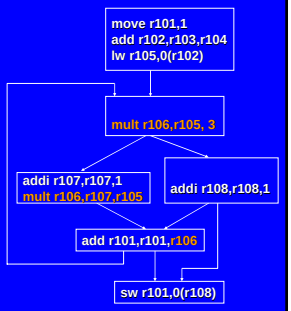
Loop Invariant Code Motion

- If instruction J's operands don't change during loop execution, J produces same result each loop iteration
- If J is also only instruction to modify its destination register, can move J outside the loop, only executed once



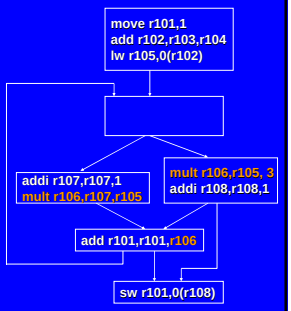
Partial Deadcode Elimination

- If instruction's result is not used along a path, the instruction can be pushed down the path it is used
  - reduces instruction's execution count



Partial Deadcode Elimination

- If instruction's result is not used along a path, the instruction can be pushed down the path it is used
  - reduces instruction's execution count



Common Sub-expression

- Eliminate recomputed sub-expressions by storing expression result in a register:

HLL:

A = B + C + D + E ...
F = G + H + D + E

IR:

$T_1 = B + C$
$T_2 = D + E$
$A = T_1 + T_2$
...
$T_3 = G + H$
$T_4 = D + E$
$F = T_3 + T_4$

⇨

$T_1 = B + C$
$T_2 = D + E$
$A = T_1 + T_2$
...
$T_3 = G + H$
$T_4 = D + E$
$F = T_3 + T_2$

Eliminating Jumps in Loops

- Jumps can generally be eliminated from inside loops (e.g., For & While) by duplicating test outside loop
  - Source:

```
WHILE (i < 14)
  <body>
```

  - Naive compile:

```
        move r101, i
test:   slti r102,r101,14
        beq r102,r0,out_of_loop
        <body>
        j test
```

  - Optimized compile:

```
        move r101,i
        slti r102,r101,14
        beq r102,r0,out_of_loop
loop:   <body>
        slti r102,r101,14
        bne r102,r0,loop
```

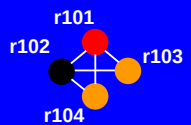

Register Allocation

- Until toward end of optimization phases compiler assumes unlimited number of *symbolic registers*
- Register allocation assigns symbolic registers to real registers
- Traditional allocation algorithms use *graph coloring*

Graph Coloring

1. Identify live range of all variables:

```
r101 = ...
r102 = ...
r103 = ...
... = r103
r104 = ...
... = ... r104
... = ... r102
... = ... r101
```

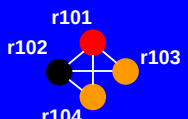


2. Construct an *interference graph*
   Symbolic are nodes, connection edge if both symbolic registers live at some point
3. Color nodes so neighbors have different colors: colors correspond to registers

Graph Coloring

- Intermediate code is rewritten with assigned real register numbers

```
r1 = ...
r2 = ...
r3 = ...
... = r3
r3 = ...
... = ... r3
... = ... r2
... = ... r1
```



Register Spilling

- Program may have more *live* variables in a region than architecture has registers
- Can *spill* a register to memory to make room in register file for another variable:

```
store $4, 40($20)  #  store register 4 at location 40 in area
                   #  pointed to by $20
load  $4,32($20)   #  get new variable from location 32
. ..
load  $7,40($20)   #  restore variable that was spilled
```

- Compiler use special algorithms to keep most active variables in registers, minimize spilling

Register Spilling

- When more symbolic registers live than real registers, some symbolic register must spill to memory
  - Assume two real registers

```
r101 = ...
r102 = ...
r103 = ...
... = r103
r104 = ...
... = ... r104
... = ... r102
... = ... r101
```

3 — r101 = ... / r102 = ... / r103 = ... / ... = r103

3 — r104 = ... / ... = ... r104 / ... = ... r102 / ... = ... r101

Register Spilling

- To spill compiler inserts store/load instructions to store/restore symbolic to/from memory
  - Live ranges now shorter, graph may be colorable

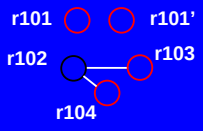
```
r101 = ...
r102 = ...
r103 = ...
... = r103
r104 = ...
... = ... r104
... = ... r102
... = ... r101'
```

2 — r101 = ... / r102 = ... / r103 = ... / ... = r103

2 — r104 = ... / ... = ... r104 / ... = ... r102

`sw r101,48($sp)`

`lw r101',48($sp)`



Loop Unrolling

- Can increase loop performance in exchange code size increase
  - HLL Code:

```
FOR (j = 0, j<1000, j++)
  sum = sum + a[j];
```

  - Intermediate code for loop:

```
loop:  lw r102,0(r101)       # r101 is address of j
       add r103,r103,r102    # sum is r103, sum = sum + a[j]
       addi r101,r101,1      #j++
       slti r104,r101,1000   # (j<1000)
       bne r104,r0,loop
```

  - Three instructions of loop overhead per iteration

Unrolled Loop

- Replicate the loop body to reduce per iternation loop overhead
  - HLL code

```
FOR (j = 0; j<1000; j=j+2)
  sum = sum + a[j];
  sum = sum + a[j+1];
```

  - Intermediate code

```
loop:  lw r102,0(r101)       # load a[j]
       add r103,r103,r102    # sum a[j]
       lw r105,4(r101)       # load a[j+1]
       add r103,r103,r105    # sum a[j+1]
       addi r101,r101,2      #j=j+2
       slti r104,r101,1000   # (j<1000)
       bne r104,r0,loop
```

  - Loop Overhead is now 1.5 instructions per iteration

Subroutine In-Line Expansion

- Insert the body of a subroutine rather than a call
  - Eliminates overhead of call/return
  - Simplifies program structure, allowing better instruction scheduling, register allocation
    - Original code:
      Load
      Add
      Call A
      <next>
    - Optimized Code
      Load
      Add
      <body of A>
      <next>

EEC 175

- Complete course on compiler optimization
- Explores the software side of the hardware/software interface
- Project course/design elective
- Spring Quarter, 5 units

EEC175 Topics

Basic blocks
Control flow graph
Loops
Call graph
Program profiling
Instruction Scheduling
Loop-Invariant Code Motion
Procedure In-Lining
Branch Optimization
Branch Alignment
Unreachable Code Elimination
Data-Flow Analysis
Data Dependence
Aliasing
Reaching Definitions
Def-Use Chains
Bit-Vector Iterative Analysis
Live-Range Analysis
Symbolic-Register Renumbering
Symbolic-Register Interference
Available Expressions
Dead-Code Elimination
Code Hoisting/Sinking
Common Sub-Expression Elimination
Partial Deadcode Elimination
Strength Reduction
Constant Propagation