CS429: Computer Organization and Architecture
Instruction Set Architecture III

Warren Hunt, Jr. and Bill Young
Department of Computer Sciences
University of Texas at Austin

Last updated: September 19, 2014 at 09:52

Controlling Program Execution

- We can now generate programs that execute linear sequences of instructions
  - Access registers and storage
  - Perform computations
- But what about loops, conditions, etc.?
- Need ISA support for:
  - comparing and testing data values
  - directing program control
    - jump to some instruction that isn't just the next one in sequence
    - Do so based on some condition that has been tested.

Condition Codes

Single bit registers

- CF: carry flag
- ZF: zero flag
- SF: sign flag
- OF: overflow flag

Implicitly set by arithmetic operations

E.g., `addl` Src, Dest

C analog: `t = a + b;`

- CF set if carry out from most significant bit; used to detect overflow in unsigned computations.
- ZF set if `t == 0`
- SF set if `t < 0`
- OF set if two's complement overflow:

```
(a>0 && b>0 && t<0) || (a<0 && b<0 && t >=0)
```

- Condition codes not set by `leal` instruction.

Setting Condition Codes

Explicitly set by Compare instruction

`cmpl` Src2, Src1

- `cmpl b, a` is like computing `a - b` without setting destination.
- CF set if carry out from most significant bit; used for unsigned computations.
- ZF set if `a == b`
- SF set if `(a-b) < 0`
- OF set if two's complement overflow:

  `(a>0 && b>0 && (a-b)<0) || (a<0 && b<0 && (a-b)>=0)`

Setting Condition Codes

Explicitly set by Test instruction

`testl` Src2, Src1

- Sets condition codes based on value of (Src1 & Src2).
- Often useful to have one of the operands by a mask.
- `testl b, a` is like computing `a&b`, without setting a destination.
- ZF set if `a == b`
- SF set if `(a-b) < 0`

Reading Condition Codes

SetX Instructions: Set single byte based on combinations of condition codes.

SetX	Condition	Description
sete	ZF	Equal / Zero
setne	~ZF	Not equal / not zero
sets	SF	Negative
setns	~SF	Nonnegative
setg	~(SF^OF)&~ZF	Greater (signed)
setge	~(SF^OF)	Greater or equal (signed)
setl	(SF^OF)	Less (signed)
setle	(SF^OF)|ZF	Less or equal (signed)
seta	~CF&~ZF	Above (unsigned)
setb	CF	Below (unsigned)

Reading Condition Codes

SetX instructions

- Set single byte based on combinations of conditions codes.
- One of 8 addressable byte registers.
  - embedded within first 4 integer registers;
  - does not alter remaining 3 bytes;
  - typically use `movzbl` to finish the job.

%eax	%ah	%al
%ecx	%ch	%cl
%edx	%dh	%dl
%ebx	%bh	%bl
%esi		
%edi		
%esp		
%ebp		

Reading Condition Codes

```
int gt (int x, int y)
{
    return x > y;
}
```

This might be compiled into the following:

```
movl    12(%ebp), %eax     # eax = y
cmpl    %eax, 8(%ebp)      # compare x : y
                           # note inverted order
setg    %al                # al = x > y
movzbl %al, %eax           # zero rest of %eax
```

Jumping

jX Instructions: Jump to different parts of the code depending on condition codes.

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not equal / not zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~(SF^OF)&~ZF	Greater (signed)
jge	~(SF^OF)	Greater or equal (signed)
jl	(SF^OF)	Less (signed)
jle	(SF^OF) | ZF	Less or equal (signed)
ja	~CF&~ZF	Above (unsigned)
jb	CF	Below (unsigned)

Conditional Branch Example

```c
int max(int x, int y)
{
    if (x > y)
        return x;
    else
        return y;
}
```

Conditional flow of control is handled at the assembler level with jumps and labels.

You can do the same in C, but it's considered bad style.

```
_max:
    pushl   %ebp
    movl    %esp, %ebp

    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     L9
    movl    %edx, %eax

L9:
    movl    %ebp, %esp
    popl    %ebp
    ret
```

Conditional Branch Example (Cont.)

```
int goto_max(int x, int y)
{
    int rval = y;
    int ok = (x <= y);
    if (ok)
        goto done;
    rval = x;
done:
    return rval;
}
```

- C allows "goto" as a means of transferring control.
- Closer to machine-level programming style.
- Generally considered bad coding style.

```
    movl    8(%ebp), %edx       # edx = x
    movl    12(%ebp), %eax      # eax = y
    cmpl    %eax, %edx          # x : y
    jle     L9                  # if <=, goto L9
    movl    %edx, %eax          # eax = x (skip if x <= y)
L9:                             # Done:
```

Do-While Loop Example

A common compilation strategy is to take a C construct and rewrite it into a semantically equivalent C version that is closer to assembly.

C Code:

```
int fact_do (int x)
{
  int result = 1;
  do {
      result *= x;
      x = x-1;
  } while (x > 1);
  return result;
}
```

Goto Version:

```
int fact_goto (int x)
{
  int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
  return result;
}
```

- Uses backward branch to continue looping.
- Only take branch when “while” condition holds.

Do-While Loop Compilation

Goto Version:

```
int fact_goto
   (int x)
{
  int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
  return result;
}
```

Registers

%edx holds `x`

%eax holds `result`

Assembly:

```
_fact_goto:
  pushl  %ebp          # setup
  movl   %esp, %ebp
  movl  $1, %eax       # eax = 1
  movl  8(%ebp),%edx   # edx = x
L11:
  imull %edx,%eax      # res *= x
  decl  %edx           # x = x-1
  cmpl  $1, %edx       # compare x:1
  jg    L11            # if > jump

  movl  %ebp, %esp     # finish
  popl  %ebp
  ret
```

General Do-While Translation

C Code:

```
do
   Body
   while (Test);
```

Goto Version:

```
loop:
    Body
    if (Test)
        goto loop;
```

- Body can be any C statement, typically is a compound statement.
- Test is an expression returning an integer.
  - If it evaluates to 0, that's interpreted as false.
  - If it evaluates to anything but 0, that's interpreted as true.

While Loop Example 1

C Code:

```
int fact_while (int x)
{
  int result = 1;
  while (x > 1) {
    result *= x;
    x = x-1;
  };
  return result
}
```

First Goto Version:

```
int fact_while_goto (int x)
{
  int result = 1;
loop:
  if (! (x > 1))
     goto done;
  result *= x;
  x = x-1;
  goto loop;
done:
  return result;
}
```

- Is this code equivalent to the do-while version?
- Must jump out of loop if the test fails.

Actual While Loop Translation

C Code:

```
int fact_while (int x)
{
  int result = 1;
  while (x > 1) {
    result *= x;
    x = x-1;
  };
  return result
}
```

Second Goto Version:

```
int fact_while_goto2 (int x)
{
  int result = 1;
  if (! (x > 1))
      goto done;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
      goto loop;
done:
  return result;
}
```

- Uses the same inner loop as do-while version.
- Guards loop entry with an extra test.

General While Translation

C Code

```
while (Test)
    Body
```

which is equivalent to:

Do-While Version

```
   if (!Test)
       goto done;
   do
       Body
       while (Test);
done:
```

which gets compiled as if it were:

Goto Version

```
   if (!Test)
       goto done;
loop:
   Body
   if (Test)
       goto loop;
done:
```

Are all three versions semantically equivalent?

For Loop Example

```
/* Compute x raised to nonnegative power p */
int ipwr_for (int x, unsigned p) {
  int result;
  for (result = 1; p != 0; p = p>>1) {
    if (p & 0x1)
      result *= x;
    x = x*x;
  }
  return result;
}
```

Algorithm

- Exploit property that $p = p_0 + 2p_1 + 4p_2 + \ldots + 2^{n-1}p_{n-1}$
- Gives $x^p = z_0 \cdot z_1^2 \cdot (z_2^2)^2 \cdot \cdots \cdot (\ldots((z_{n-1}^2)^2)\ldots)^2$
  - $z_i = 1$ when $p_i = 0$
  - $z_i = x$ when $p_i = 1$
- Complexity is $O(\log p)$

ipwr Computation

```
/* Compute x raised to nonnegative power p */
int ipwr_for (int x, unsigned p) {
    int result;
    for (result = 1; p != 0; p = p>>1) {
        if (p & 0x1)
            result *= x;
        x = x*x;
    }
    return result;
}
```

result	x	p
1	3	10
1	9	5
9	81	2
9	6561	1
59049	43046721	0

For Loop Example

General Form:

```
for ( Init; Test; Update )
    Body
```

C Code:

```
    int result;
    for (result = 1;
           p != 0;
           p = p>>1) {
        if (p & 0x1)
            result *= x;
       x = x*x;
    }
```

Init:

```
result = 1
```

Test:

```
p != 0
```

Update:

```
p = p >> 1
```

Body:

```
{
    if (p & 0x1)
        result *= x;
    x = x*x;
}
```

For → While

For Version:

```
for (Init; Test; Update)
    Body
```

which is equivalent to:

While Version:

```
Init;
while (Test) {
    Body
    Update;
}
```

which becomes:

Do-While Version

```
Init;
if (!Test)
   goto done;
do {
      Body
      Update;
   } while (Test);
done:
```

and finally into:

Goto Version:

```
    Init;
    if (!Test)
        goto done;
loop:
    Body
    Update;
    if (Test)
        goto loop;
done:
```

For Loop Compilation

Goto Version:

```
    Init;
    if (!Test)
        goto done;
loop:
    Body
    Update;
    if (Test)
        goto loop;
done:
```

Init: `result = 1`
Test: `p != 0`
Update: `p = p >> 1`

Body:

```
  {
      if (p & 0x1)
          result *= x;
      x = x*x;
  }
```

finally yields code:

```
    result = 1;
    if (p == 0)
        goto done;
loop:
    if (p & 0x1)
        result *= x;
    x = x*x;
    p = p >> 1;
    if (p != 0)
        goto loop;
done:
```

Switch Statements

```
typedef enum
  {ADD, MULT, MINUS, DIV,
   MOD, BAD} op_type;

char unparse_symbol
            ( op_type op )
{
    switch (op) {
    case ADD:
       return '+';
    case MULT:
       return '*';
    case MINUS:
       return '-';
    case DIV:
       return '/';
    case MOD:
       return '%';
    case BAD:
       return '?';
    }
}
```

Implementation Options

- Series of conditionals
  - Good if few cases, but
  - Slow if there are many.
- Jump Table
  - Lookup branch target
  - Avoids conditionals
  - Possible when cases are small integer constants
- GCC
  - Picks best implementation based on case structure.
- Bug in example code: no default given

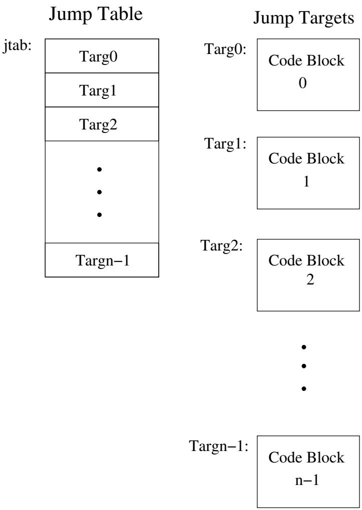
Jump Table Structure

Switch General Form

```
switch (op) {
  case val_0:
     Block 0
  case val_1:
     Block 1
    ...
  case val_n-1:
     Block n-1
}
```

Approx. Translation

```
target = JTab[op];
goto *target;
```



Switch Statement Example

```
typedef enum
  {ADD, MULT, MINUS, DIV,
   MOD, BAD} op_type;

char unparse_symbol
          ( op_type op )
{
   switch (op) {
      ...
   }
}
```

Enumerated Values

ADD	0
MULT	1
MINUS	2
DIV	3
MOD	4
BAD	5

Setup

```
unparse_symbol:
  pushl   %ebp             # setup
  movl    %esp, %ebp       # setup
  movl    8(%ebp), %eax    # eax = op
  cmpl    $5, %eax         # compare op : 5
  ja      .L49             # if > goto done
  jmp     *.L57(,%eax,4)   # goto Table[op]
```

Assembly Setup Explanation

Symbolic Labels

Labels of the form .LXX are translated into addresses by the assembler.

Table Structure

- Each target requires 4 bytes
- Base address at .L57

Jumping

- `jmp .L49`: jump target is denoted by label .L49
- `jmp *.L57(,%eax,4)}`
  - Start of jump table denoted by label .L57
  - Register `%eax` holds op
  - Must scale by a factor of 4 to get offset into table
  - Fetch target from effective address `.L57 + op*4`

Jump Table

Table Contents

```
.section .rodata
    .align 4
.L57:
    .long  .L51  # op = 0
    .long  .L52  # op = 1
    .long  .L53  # op = 2
    .long  .L54  # op = 3
    .long  .L55  # op = 4
    .long  .L56  # op = 5
```

Enumerated Values

ADD	0
MULT	1
MINUS	2
DIV	3
MOD	4
BAD	5

Targets and Completion

```
.L51:
    movl  $43,%eax  # '+'
    jmp   .L49
.L52:
    movl  $42,%eax  # '*'
    jmp   .L49
.L53:
    movl  $45,%eax  # '-'
    jmp   .L49
.L54:
    movl  $47,%eax  # '/'
    jmp   .L49
.L55:
    movl  $37,%eax  # '%'
    jmp   .L49
.L56:
    movl  $63,%eax  # '?'
   # Fall through to .L49
```

Switch Statement Completion

```
.L49:                       # done:
    movl   %ebp,%esp        # finish
    popl   %ebp             # finish
    ret                     # finish
```

Puzzle: what value is returned when op is invalid?

Answer:

- Register %eax set to op at beginning of procedure.
- This becomes the returned value.

Advantage of Jump Table

- Can do k-way branch in $O(1)$ operations.

Object Code

Setup

- Label .L49 becomes address 0x804875c
- Label .L57 becomes address 0x8048bc0

```
08048718 <unparse_symbol>:
08048718:  55                  pushl  %ebp
08048719:  89 e5               movl   %esp,%ebp
0804871b:  8b 45 08            movl   0x8(%ebp),%eax
0804871e:  83 f8 05            cmpl   $0x5,%eax
08048721:  77 39               ja     804875c <unparse_symbol+0x44>
08048723:  ff 24 85 c0 8b jmp    *0x8048bc0(,%eax,4)
```

Object Code (Cont.)

Jump Table

- Doesn't show up in disassembled code.
- But can inspect using GDB:

```
gdb code-examples
(gdb) x/6xw 0x8048bc0
```

  - Examine 6 hexadecimal format words (4-bytes each)
  - Use command `help x` to get format documentation

```
0x8048bc0 <_fini+32>:
    0x08048730
    0x08048737
    0x08048740
    0x08048747
    0x08048750
    0x08048757
```

Extracting Jump Table from Binary

Jump Table is stored in read only data segment (.rodata)

- Various fixed values needed by your code.

You can examine it with objdump

```
objdump code-examples -s --section=.rodata
```

- Shows everything in the indicated segment.

It's hard to read; jump table entries are shown with reversed byte ordering.

```
Contents of section .rodata:
  8048bc0 30870408 37870408 40870408 47870408
  8048bd0 50870408 57870408 46616374 28256429
  8048be0 203d2025 6c640a00 43686172 203d2025
  ...
```

E.g., 30870408 really means 0x08048730.

Disassembled Targets

```
8048730:b8 2b 00 00 00   movl $0x2b,%eax
8048735:eb 25            jmp  804875c <unparse_symbol+0x44>
8048737:b8 2a 00 00 00   movl $0x2a,%eax
804873c:eb 1e            jmp  804875c <unparse_symbol+0x44>
804873e:89 f6            movl %esi,%esi
8048740:b8 2d 00 00 00   movl $0x2d,%eax
8048745:eb 15            jmp  804875c <unparse_symbol+0x44>
8048747:b8 2f 00 00 00   movl $0x2f,%eax
804874c:eb 0e            jmp  804875c <unparse_symbol+0x44>
804874e:89 f6            movl %esi,%esi
8048750:b8 25 00 00 00   movl $0x25,%eax
8048755:eb 05            jmp  804875c <unparse_symbol+0x44>
8048757:b8 3f 00 00 00   movl $0x3f,%eax
```

`movl %esi,%esi` does nothing; it's inserted to align instructions for better cache performance.

Matching Disassembled Targets

The jump table had entries:

- 0x08048730
- 0x08048737
- 0x08048740
- 0x08048747
- 0x08048750
- 0x08048757

Can you match them to the code?

```
8048730:b8 2b 00 00 00   movl
8048735:eb 25            jmp
8048737:b8 2a 00 00 00   movl
804873c:eb 1e            jmp
804873e:89 f6            movl
8048740:b8 2d 00 00 00   movl
8048745:eb 15            jmp
8048747:b8 2f 00 00 00   movl
804874c:eb 0e            jmp
804874e:89 f6            movl
8048750:b8 25 00 00 00   movl
8048755:eb 05            jmp
8048757:b8 3f 00 00 00   movl
```

Sparse Switch Example

```
/* Return x/111 if x is a
   multiple && <= 999; return
   -1 otherwise. */

int div111 (int x)
{
   switch (x) {
   case   0: return 0;
   case 111: return 1;
   case 222: return 2;
   case 333: return 3;
   case 444: return 4;
   case 555: return 5;
   case 666: return 6;
   case 777: return 7;
   case 888: return 8;
   case 999: return 9;
   default:  return -1;
   }
}
```

- It's not practical to use a jump table; it would require 1000 entries.
- The obvious translationis into if-then-else would have a maximum of 9 tests.

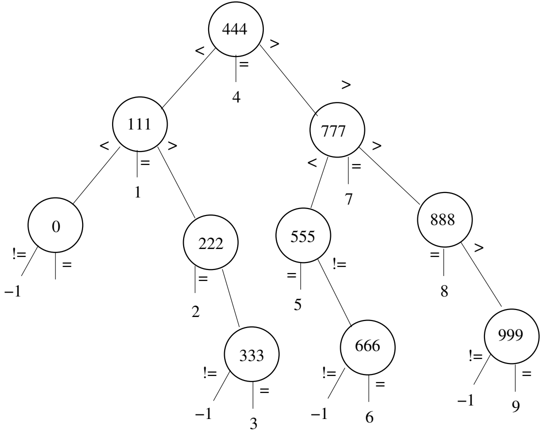
Sparse Switch Code

```
    movl   8(%ebp),%eax    #
        get  x
    cmpl   $444,%eax       # x
        :444
    je     L8
    jg     L16
    cmpl   $111,%eax       # x
        :111
    je     L5
    jg     L17
    testl  %eax,%eax       # x
        :0
    je     L4
    jmp    L14
    ...
```

- Compares x to possible case values.
- Jumps different places depending on outcomes.

```
     ...
L5:
    movl  $1,%eax
    jmp   L19
L6:
    movl  $2,%eax
    jmp   L19
L7:
    movl  $3,%eax
    jmp   L19
L8:
    movl  $4,%eax
    jmp   L19
    ...
```

Sparse Switch Code Structure



- Organizes cases as binary tree.
- Gives logarithmic performance.
- *What a clever algorithm!*

Summarizing

C Control

- if-then-else
- do-while
- while
- for
- switch

Assembler Control

- jump
- conditional jump

Compiler

- must generate assembly code to implement more complex control

Summarizing

Standard Techniques

- All loops converted to do-while form
- Large switch statements use jump tables

Conditions in CISC

- CISC machines generally have condition code registers

Conditions in RISC

- Use general registers to store condition information
- Have special comparison instructions
- E.g., on Alpha:

```
cmple  $16,1,$1
```

Sets register \$1 to 1 when register \$16 $<= 1$