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 Srcl, 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:
  \[ (a>0 \land \land b>0 \land \land t<0) \lor (a<0 \land \land b<0 \land \land t \geq 0) \]
- Condition codes not set by leal instruction.
Explicitly set by Compare instruction

\texttt{cmp} \ Src2, \ Src1

- \texttt{cmp} \ b, \ a \text{ is like computing } a - b \text{ 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:
  \( (a>0 \land b>0 \land (a-b)<0) \lor (a<0 \land b<0 \land (a-b)\geq0) \)
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
**SetX Instructions**: Set single byte based on combinations of condition codes.

<table>
<thead>
<tr>
<th><strong>SetX</strong></th>
<th><strong>Condition</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>sete</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>setne</td>
<td>~ZF</td>
<td>Not equal / not zero</td>
</tr>
<tr>
<td>sets</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>setns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg</td>
<td>~(SF^OF)&amp;~ZF</td>
<td>Greater (signed)</td>
</tr>
<tr>
<td>setge</td>
<td>~(SF^OF)</td>
<td>Greater or equal (signed)</td>
</tr>
<tr>
<td>setl</td>
<td>(SF^OF)</td>
<td>Less (signed)</td>
</tr>
<tr>
<td>settle</td>
<td>(SF^OF)</td>
<td>ZF</td>
</tr>
<tr>
<td>seta</td>
<td>~CF&amp;~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>setb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>%eax</th>
<th>%ah</th>
<th>%al</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>%ch</td>
<td>%cl</td>
</tr>
<tr>
<td>%edx</td>
<td>%dh</td>
<td>%dl</td>
</tr>
<tr>
<td>%ebx</td>
<td>%bh</td>
<td>%bl</td>
</tr>
<tr>
<td>%esi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Reading Condition Codes

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

This might be compiled into the following:

```assembly
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.

<table>
<thead>
<tr>
<th>jX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne</td>
<td>~ZF</td>
<td>Not equal / not zero</td>
</tr>
<tr>
<td>js</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg</td>
<td>~(SF^OF)&amp;~ZF</td>
<td>Greater (signed)</td>
</tr>
<tr>
<td>jge</td>
<td>~(SF^OF)</td>
<td>Greater or equal (signed)</td>
</tr>
<tr>
<td>jl</td>
<td>(SF^OF)</td>
<td>Less (signed)</td>
</tr>
<tr>
<td>jle</td>
<td>(SF^OF)</td>
<td>ZF</td>
</tr>
<tr>
<td>ja</td>
<td>~CF&amp;~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
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:
popl %esp
retn
```
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:
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:

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

Goto Version:

```c
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:

```c
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  # 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
```
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:

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

First Goto Version:

```c
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.
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.
C Code

\[ \text{while} \ (\text{Test}) \]
\[ \text{Body} \]

which is equivalent to:

Do-While Version

\[ \text{if} \ (! \text{Test}) \]
\[ \text{goto} \ \text{done}; \]
\[ \text{do} \]
\[ \text{Body} \]
\[ \text{while} \ (\text{Test}); \]
\[ \text{done:} \]

which gets compiled as if it were:

Goto Version

\[ \text{if} \ (! \text{Test}) \]
\[ \text{goto} \ \text{done}; \]
\[ \text{loop:} \]
\[ \text{Body} \]
\[ \text{if} \ (\text{Test}) \]
\[ \text{goto} \ \text{loop}; \]
\[ \text{done:} \]

Are all three versions semantically equivalent?
For Loop Example

```c
/* 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 \( x^p = z_0 \cdot z_1^2 \cdot (z_2^2)^2 \cdot \ldots \cdot ((z_{n-1}^2)^2)^2 \)
  - \( z_i = 1 \) when \( p_i = 0 \)
  - \( z_i = x \) when \( p_i = 1 \)
- Complexity is \( O(\log p) \)
/* 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;
}
For Loop Example

General Form:

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

C Code:

```c
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:

```c
{
    if (p & 0x1)
        result *= x;
    x = x*x;
}
```
For → While

For Version:

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

which is equivalent to:

While Version:

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

which becomes:

Do-While Version

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

and finally into:

Goto Version:

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

char unparsesymbol( 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

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

Approx. Translation

```c
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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>0</td>
</tr>
<tr>
<td>MULT</td>
<td>1</td>
</tr>
<tr>
<td>MINUS</td>
<td>2</td>
</tr>
<tr>
<td>DIV</td>
<td>3</td>
</tr>
<tr>
<td>MOD</td>
<td>4</td>
</tr>
<tr>
<td>BAD</td>
<td>5</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Contents of section .rodata:</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048bc0 30870408 37870408 40870408 47870408</td>
</tr>
<tr>
<td>8048bd0 50870408 57870408 46616374 28256429</td>
</tr>
<tr>
<td>8048be0 203d2025 6c640a00 43686172 203d2025</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

E.g., 30870408 really means 0x08048730.
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?

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048730</td>
<td>b8 2b 00 00 00 movl</td>
</tr>
<tr>
<td>8048735</td>
<td>eb 25 jmp</td>
</tr>
<tr>
<td>8048737</td>
<td>b8 2a 00 00 00 movl</td>
</tr>
<tr>
<td>804873c</td>
<td>eb 1e jmp</td>
</tr>
<tr>
<td>804873e</td>
<td>89 f6 movl</td>
</tr>
<tr>
<td>8048740</td>
<td>b8 2d 00 00 00 movl</td>
</tr>
<tr>
<td>8048745</td>
<td>eb 15 jmp</td>
</tr>
<tr>
<td>8048747</td>
<td>b8 2f 00 00 00 movl</td>
</tr>
<tr>
<td>804874c</td>
<td>eb 0e jmp</td>
</tr>
<tr>
<td>804874e</td>
<td>89 f6 movl</td>
</tr>
<tr>
<td>8048750</td>
<td>b8 25 00 00 00 movl</td>
</tr>
<tr>
<td>8048755</td>
<td>eb 05 jmp</td>
</tr>
<tr>
<td>8048757</td>
<td>b8 3f 00 00 00 movl</td>
</tr>
</tbody>
</table>
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 translation is into if-then-else would have a maximum of 9 tests.
Sparse Switch Code

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

```assembly
movl 8(%ebp),%eax  # get x
cmpl $444,%eax   # x = 444
    :444
je   L8
jg   L16

cmp $111,%eax   # x = 111
    :111
je   L5
jg   L17
testl %eax,%eax # x = 0
    :0
je   L4
jmp   L14

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

Compares $x$ to possible case values.
Jumps different places depending on outcomes.
Organizes cases as binary tree.
Gives logarithmic performance.
What a clever algorithm!
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:
  \[ \text{cmp} \leq \] $16,1,$1
  Sets register $1$ to $1$ when register $16 \leq 1$