MACHINE LEVEL PROGRAMMING 2: CONTROL

CS 045

Computer Organization and Architecture

Prof. Donald J. Patterson

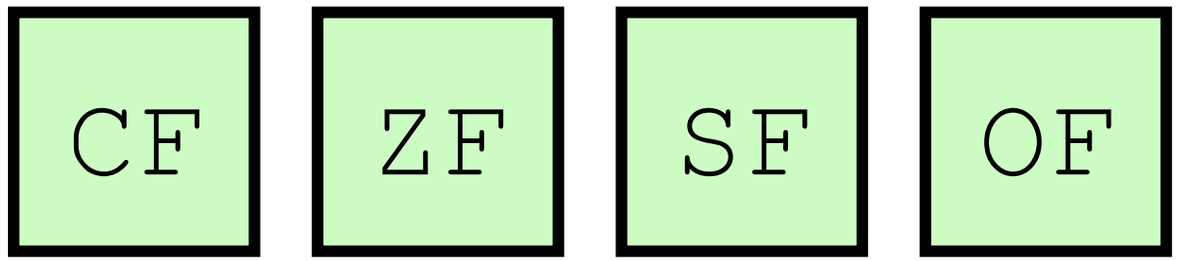
Adapted from Bryant and O'Hallaron,
Computer Systems:
A Programmer's Perspective, Third Edition

- CONTROL: CONDITION CODES
- CONDITIONAL BRANCHES
- LOOPS
- SWITCH STATEMENTS

- CONTROL: CONDITION CODES
- CONDITIONAL BRANCHES
- LOOPS
- SWITCH STATEMENTS

CONTROL

CONDITION CODES

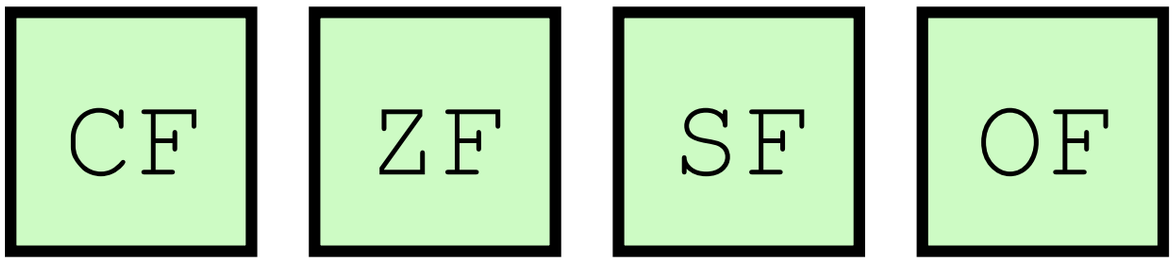


- Single bit registers
  - CF Carry Flag (for unsigned)
  - ZF Zero Flag
  - SF Sign Flag (for signed)
  - OF Overflow Flag (for signed)

CONTROL

CONDITION CODES

Condition Codes



- Single bit registers
  - CF Carry Flag (for unsigned)
  - ZF Zero Flag
  - SF Sign Flag (for signed)
  - OF Overflow Flag (for signed)

CONTROL

JUMPING

- jX Instructions
  - Jump to different part of 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

- wcpkneel> gcc –O1 -S –fno-if-conversion control.c

```
long absdiff
  (long x, long y)
{
  long result;
  if (x > y)
    result = x-y;
  else
    result = y-x;
  return result;
}
```

```
absdiff:
    cmpq    %rsi, %rdi   # x:y
    jle     .L4
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret
.L4:                     # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax
    ret
```

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rax	Return value

EXPRESSING WITH GOTO CODE

- C allows goto statement
- Jump to position designated by label

```
long absdiff
   (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff_j
   (long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```

GENERAL CONDITIONAL EXPRESSION TRANSLATION (USING BRANCHES)

C code

```
val = Test ? Then_Expr : Else_Expr;
```

```
val = x>y ? x-y : y-x;
```

goto version

```
  ntest = !Test;
  if (ntest) goto Else;
  val = Then_Expr;
  goto Done;
Else:
  val = Else_Expr;
Done:
  . . .
```

- Create separate code regions for then & else expressions
- Execute appropriate one

USING CONDITIONAL MOVES

- Conditional Move Instructions
  - Instruction supports:
    - if (`test`) `dest <- src`
  - Supported in post-1995 x86 processors
  - GCC tries to use them
    - But, only when known to be safe
- Why?
  - Branches are very disruptive to instruction flow through pipelines
  - Conditional moves do not require control transfer

C code

```
val = Test
    ? Then_Expr
    : Else_Expr;
```

goto version

```
result = Then_Expr;
eval = Else_Expr;
nt = !Test;
if (nt) result = eval;
return result;
```

CONDITIONAL MOVE EXAMPLE

C code

```
long absdiff
   (long x, long y)
{
     long result;
     if (x > y)
          result = x-y;
     else
          result = y-x;
     return result;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

```
absdiff:
    movq     %rdi, %rax   # x
    subq     %rsi, %rax   # result = x-y
    movq     %rsi, %rdx
    subq     %rdi, %rdx   # eval = y-x
    cmpq     %rsi, %rdi   # x:y
    cmovle   %rdx, %rax   # if <=, result = eval
    ret
```

goto version

```
result = Then_Expr;
eval = Else_Expr;
nt = !Test;
if (nt) result = eval;
return result;
```

BAD CASES FOR CONDITIONAL MOVE

Expensive Computations

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

- Both values get computed
- Only makes sense when computations are very simple

Risky Computations

```
val = p ? *p : 0;
```

- Both values get computed
- May have undesirable effects

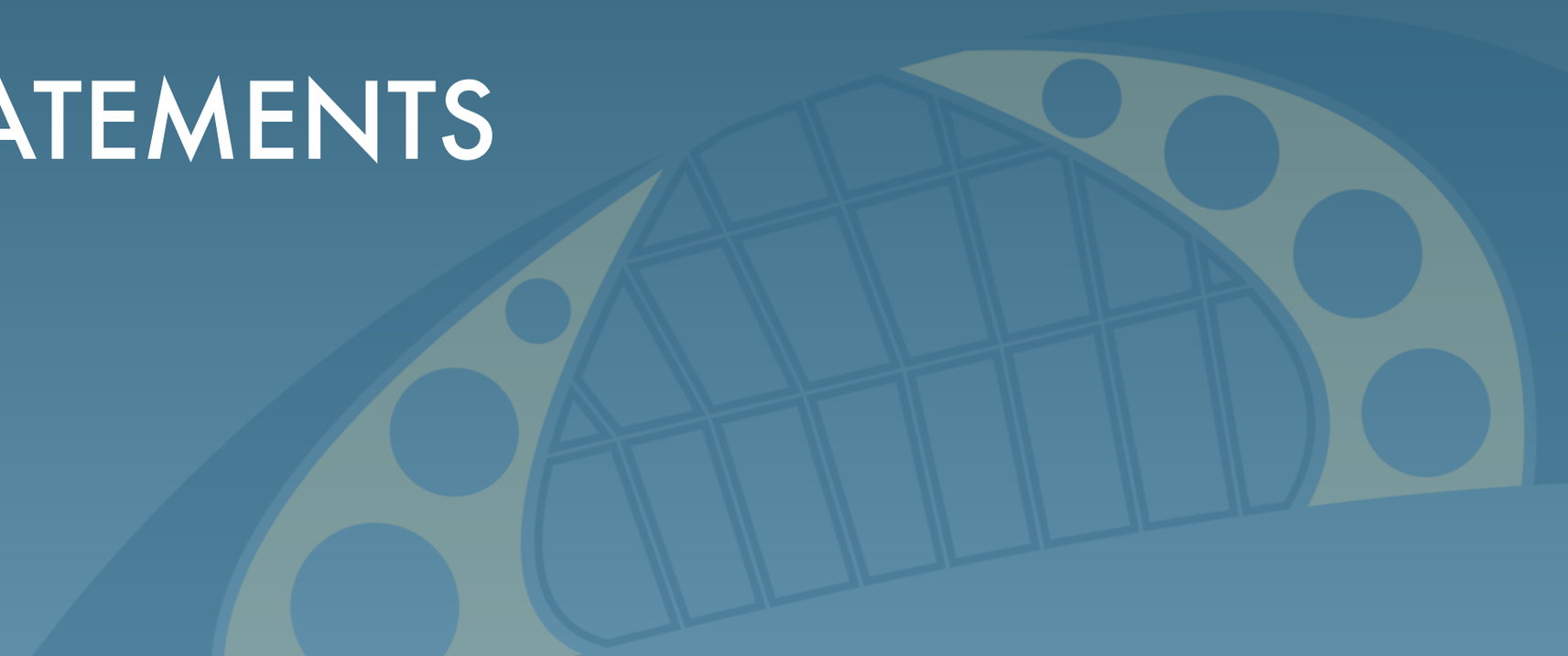
Computations with side effects

```
val = x > 0 ? x*=7 : x+=3;
```

- Both values get computed
- Must be side-effect free

- CONTROL: CONDITION CODES
- CONDITIONAL BRANCHES
- LOOPS
- SWITCH STATEMENTS

- CONTROL: CONDITION CODES
- CONDITIONAL BRANCHES
- LOOPS
- SWITCH STATEMENTS



CONTROL

"DO-WHILE" LOOP EXAMPLE

C code

```
long pcount_do
  (unsigned long x) {
  long result = 0;
  do {
    result += x & 0x1;
    x >>= 1;
  } while (x);
  return result;
}
```

goto version

```
long pcount_goto
  (unsigned long x) {
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

- Count number of 1's in argument x ("popcount")
- Use conditional branch to either continue looping or to exit loop

CONTROL

"DO-WHILE" LOOP EXAMPLE

Register	Use(s)
%rdi	Argument x
%rax	result

goto version

```
long pcount_goto
   (unsigned long x) {
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

```
   movl    $0, %eax    #  result = 0
.L2:                   # loop:
   movq    %rdi, %rdx
   andl    $1, %edx    #  t = x & 0x1
   addq    %rdx, %rax  #  result += t
   shrq    %rdi        #  x >>= 1
   jne     .L2         #  if (x) goto loop
   ret
```

CONTROL

GENERAL "DO-WHILE" TRANSLATION

C code

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

goto version

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

- Body

```
{
   Statement1;
   Statement2;
      …
   Statementn;
}
```

CONTROL

GENERAL "WHILE" TRANSLATION

C code

```
while (Test)
  Body
```

goto version

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

- "Jump-to-middle" translation

CONTROL

GENERAL "WHILE" TRANSLATION EXAMPLE

C code

```
long pcount_while
  (unsigned long x) {
  long result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

goto version

```
long pcount_goto_jtm
  (unsigned long x) {
  long result = 0;
  goto test;
 loop:
  result += x & 0x1;
  x >>= 1;
 test:
  if(x) goto loop;
  return result;
}
```

- Compare to do-while version of function
- Initial goto starts loop at test

CONTROL

GENERAL "WHILE" TRANSLATION

while version

```
while (Test)
   Body
```

- do while conversion

do-while version

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

goto version

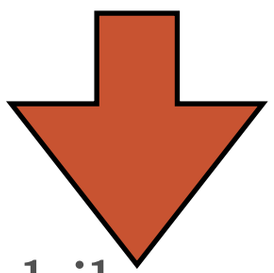
```
   if (!Test)
      goto done;
loop:
   Body
   if (Test)
      goto loop;
done:
```

CONTROL

GENERAL "WHILE" TRANSLATION

while version

```
while (Test)
   Body
```



do-while version

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

- do while conversion

goto version

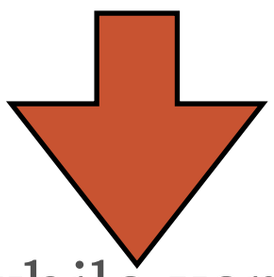
```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

CONTROL

GENERAL "WHILE" TRANSLATION

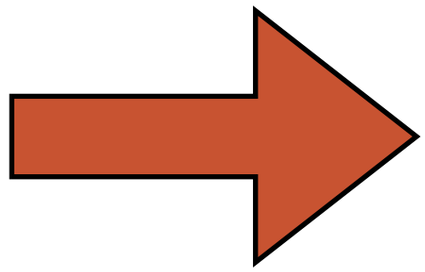
while version

```
while (Test)
   Body
```



do-while version

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



- do while conversion

goto version

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

GENERAL "WHILE" TRANSLATION EXAMPLE

C code

```
long pcount_while
  (unsigned long x) {
  long result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

goto version

```
long pcount_goto_dw
  (unsigned long x) {
  long result = 0;
  if (!x) goto done;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
 done:
  return result;
}
```

- Compare to do-while version of function
- Initial conditional guards entrance to loop

CONTROL

"FOR" LOOP FORM

General Form

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

```
#define WSIZE 8*sizeof(int)
long pcount_for
  (unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
       (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

init

```
i = 0
```

test

```
i < WSIZE
```

update

```
i++
```

body

```
{
  unsigned bit =
     (x >> i) & 0x1;
  result += bit;
}
```

CONTROL

FOR LOOP TO WHILE LOOP

General Form of for loop

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

General Form of while loop

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

CONTROL

FOR-WHILE CONVERSION

General Form

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

```
#define WSIZE 8*sizeof(int)
long pcount_for
  (unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
       (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

init

```
i = 0
```

test

```
i < WSIZE
```

update

```
i++
```

body

```
{
  unsigned bit =
     (x >> i) & 0x1;
  result += bit;
}
```

CONTROL

FOR-WHILE CONVERSION

```
#define WSIZE 8*sizeof(int)
long pcount_for
   (unsigned long x)
{
   size_t i;
   long result = 0;
   for (i = 0; i < WSIZE; i++)
   {
      unsigned bit =
         (x >> i) & 0x1;
      result += bit;
   }
   return result;
}
```

init

```
i = 0
```

test

```
i < WSIZE
```

update

```
i++
```

body

```
{
   unsigned bit =
      (x >> i) & 0x1;
   result += bit;
}
```

```
long pcount_for_while
   (unsigned long x)
{
   size_t i;
   long result = 0;
   i = 0;
   while (i < WSIZE)
   {
      unsigned bit =
         (x >> i) & 0x1;
      result += bit;
      i++;
   }
   return result;
}
```

CONTROL

FOR-DO-WHILE CONVERSION

```
long pcount_for_while
  (unsigned long x)
{
  size_t i;
  long result = 0;
  i = 0;
  while (i < WSIZE)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
    i++;
  }
  return result;
}
```

```
long pcount_for_goto_dw
  (unsigned long x) {
  size_t i;
  long result = 0;
  i = 0;                          init
  if (!(i < WSIZE))               !test
    goto done;
 loop:
  {
    unsigned bit =                body
      (x >> i) & 0x1;
    result += bit;
  }
  i++;                            update
  if (i < WSIZE)                  test
    goto loop;
 done:
  return result;
}
```

CONTROL

FOR-DO-WHILE CONVERSION

```
long pcount_for_while
  (unsigned long x)
{
  size_t i;
  long result = 0;
  i = 0;
  while (i < WSIZE)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
    i++;
  }
  return result;
}
```

```
long pcount_for_goto_dw
  (unsigned long x) {
  size_t i;
  long result = 0;
  i = 0;
  if (!(i < WSIZE)
    goto done;
 loop:
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  i++;
  if (i < WSIZE)
    goto loop;
 done:
  return result;
}
```

init

!test

body

update

test

possibly optimize the initial test away

CONTROL

FOR-DO-WHILE CONVERSION

```
long pcount_for_while
  (unsigned long x)
{
  size_t i;
  long result = 0;
  i = 0;
  while (i < WSIZE)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
    i++;
  }
  return result;
}
```

```
long pcount_for_goto_dw
  (unsigned long x) {
  size_t i;
  long result = 0;
  i = 0;
  if (!(i < WSIZE))
    goto done;
 loop:
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  i++;
  if (i < WSIZE)
    goto loop;
 done:
  return result;
}
```

init

possibly optimize the initial test away

body

update

test

WORK IT OUT

WORK IT OUT

RECONSTRUCT THE C CODE

```
fun_a:
        movl    $0, %eax
        testq   %rdi, %rdi
        je      .L3
.L6:
        xorq    %rdi, %rax
        shrq    %rdi
        jne     .L6
.L3:
        andl    $1, %eax
        ret
```

```
long fun_a(unsigned long x){

    long val = 0;
    while (□){

        }
    return
}
```

EXPLAIN IN ENGLISH WHAT THE CODE DOES

- CONTROL: CONDITION CODES
- CONDITIONAL BRANCHES
- LOOPS
- SWITCH STATEMENTS

- CONTROL: CONDITION CODES
- CONDITIONAL BRANCHES
- LOOPS
- SWITCH STATEMENTS

CONTROL

SWITCH STATEMENT EXAMPLE

- Multiple case labels
  - Here: 5 & 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4

C code

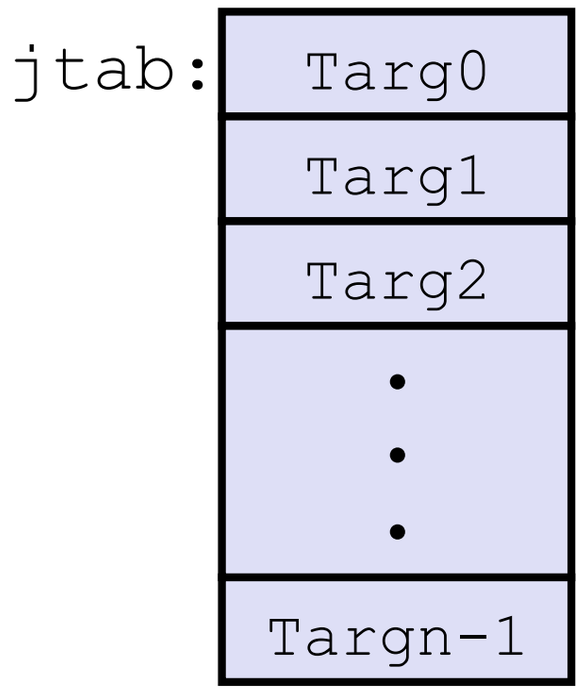
```
long switch_eg
   (long x, long y, long z)
{
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

JUMP TABLE STRUCTURE

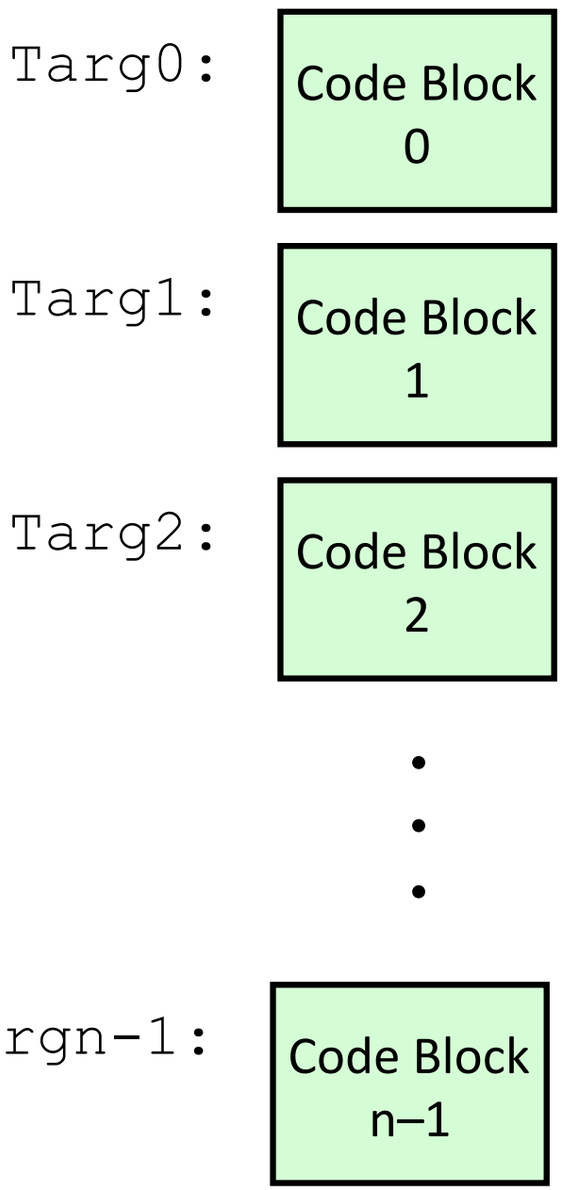
Switch Form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

Jump Table



Jump Targets



Extended-C translation

```
goto *JTab[x];
```

CONTROL

SWITCH STATEMENT EXAMPLE

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

Setup

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi        # x:6
    ja      .L8             # Use default
    jmp     *.L4(,%rdi,8)   # goto *JTab[x]
```

Note w is not initialized here

What range of values takes default?

CONTROL

SWITCH STATEMENT EXAMPLE

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

Setup

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi        # x:6
    ja      .L8             # Use default
    jmp     *.L4(,%rdi,8)   # goto *JTab[x]
```

What range of values takes default?

Jump Table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8  # x = 0
  .quad     .L3  # x = 1
  .quad     .L5  # x = 2
  .quad     .L9  # x = 3
  .quad     .L8  # x = 4
  .quad     .L7  # x = 5
  .quad     .L7  # x = 6
```

CONTROL

JUMP TABLE

Jump Table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8  # x = 0
  .quad     .L3  # x = 1
  .quad     .L5  # x = 2
  .quad     .L9  # x = 3
  .quad     .L8  # x = 4
  .quad     .L7  # x = 5
  .quad     .L7  # x = 6
```

```
    switch(x) {
    case 1:       // .L3
        w = y*z;
        break;
    case 2:       // .L5
        w = y/z;
        /* Fall Through */
    case 3:       // .L9
        w += z;
        break;
    case 5:
    case 6:       // .L7
        w -= z;
        break;
    default:      // .L8
        w = 2;
    }
```

CONTROL

JUMP TABLE

Jump Table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8  # x = 0
  .quad     .L3  # x = 1
  .quad     .L5  # x = 2
  .quad     .L9  # x = 3
  .quad     .L8  # x = 4
  .quad     .L7  # x = 5
  .quad     .L7  # x = 6
```

```
    switch(x) {
    case 1:       // .L3
        w = y*z;
        break;
    case 2:       // .L5
        w = y/z;
        /* Fall Through */
    case 3:       // .L9
        w += z;
        break;
    case 5:
    case 6:       // .L7
        w -= z;
        break;
    default:      // .L8
        w = 2;
    }
```

CONTROL

CODE BLOCKS (X == 1)

```
  switch(x) {
  case 1:         // .L3
         w = y*z;
         break;
    . . .
}
```

```
.L3:
    movq     %rsi, %rax   # y
    imulq    %rdx, %rax   # y*z
    ret
```

Register	Use(s)
`%rdi`	Argument **x**
`%rsi`	Argument **y**
`%rdx`	Argument **z**
`%rax`	Return value

CONTROL

SUMMARIZING

- C Control
  - if-then-else
  - do-while
  - while, for
  - switch
- Assembler Control
  - Conditional jump
  - Conditional move
  - Indirect jump (via jump tables)
  - Compiler generates code sequence to implement more complex control
- Standard Techniques
  - Loops converted to do-while or jump-to-middle form
  - Large switch statements use jump tables
  - Sparse switch statements may use decision trees (if-elseif-elseif-else)

WORK IT OUT

WORK IT OUT

RECONSTRUCT THE C CODE

WORK IT OUT

RECONSTRUCT THE C CODE

```
workItOut:
        cmpq    $7, %rdi
        ja      .L8
        jmp     *.L7(,%rdi,8)
.L7:
        .quad   .L3
        .quad   .L8
        .quad   .L4
        .quad   .L8
        .quad   .L5
        .quad   .L6
        .quad   .L8
        .quad   .L4
.L5:
        movl    $4, %esi
        jmp     .L8
.L6:
        movq    %rsi, %rdx
        xorq    $15, %rdx
.L3:
        leaq    112(%rdx), %rsi
        jmp     .L8
.L4:
        leaq    (%rdx,%rsi), %rsi
        salq    $2, %rsi
.L8:
        movq    %rsi, (%rcx)
        ret
```

WORK IT OUT

RECONSTRUCT THE C CODE

```
workItOut:
        cmpq    $7, %rdi
        ja      .L8
        jmp     *.L7(,%rdi,8)
.L7:
        .quad   .L3
        .quad   .L8
        .quad   .L4
        .quad   .L8
        .quad   .L5
        .quad   .L6
        .quad   .L8
        .quad   .L4
.L5:
        movl    $4, %esi
        jmp     .L8
.L6:
        movq    %rsi, %rdx
        xorq    $15, %rdx
.L3:
        leaq    112(%rdx), %rsi
        jmp     .L8
.L4:
        leaq    (%rdx,%rsi), %rsi
        salq    $2, %rsi
.L8:
        movq    %rsi, (%rcx)
        ret
```

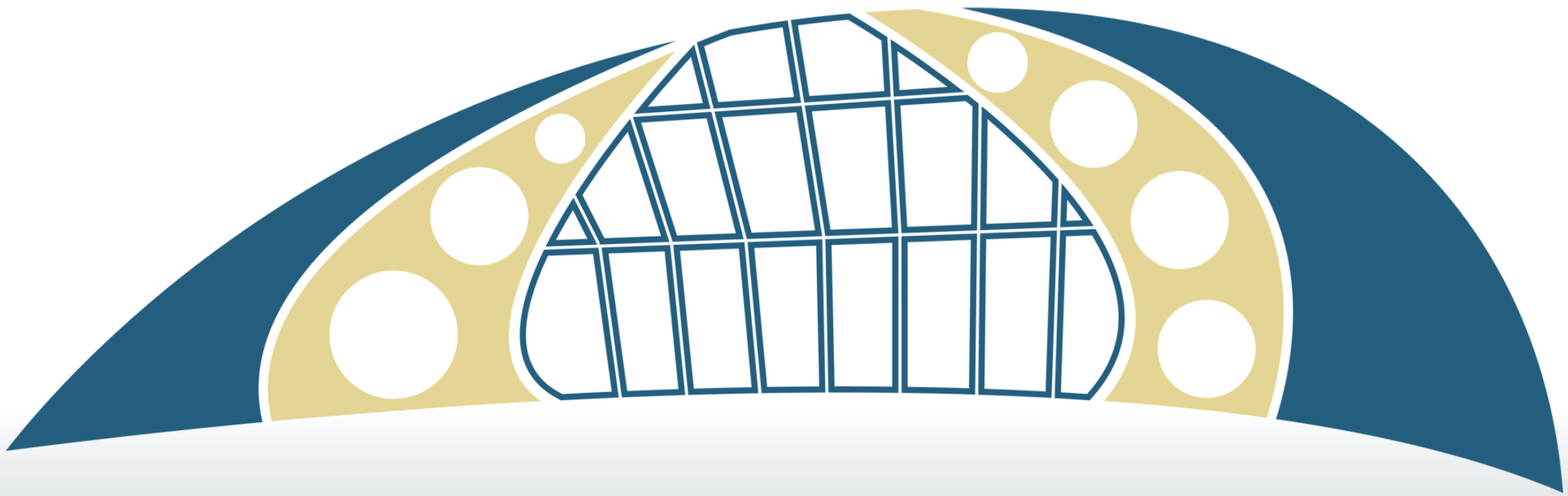
```
void workItOut(long a, long b, long c, long *dest)
{
    long val;
    switch(a){
        case □:
            c = □
        case □:
            val = □
            break;
        case □:
        case □:
            val = □
            break;
        case □:
            val = □
            break;
        default:
            val = □
    }
    *dest = val;
}
```

NEXT UP:

- STACK
- CALL / RETURN
- PROCEDURE CALL DISCIPLINE

NEXT UP:

- STACK
- CALL / RETURN
- PROCEDURE CALL DISCIPLINE



WESTMONT INSPIRED

COMPUTING LAB