CPSC 213
Introduction to Computer Systems

Unit 1d
Static Control Flow

All slides adapted from materials by Mike Feeley, Jonatan Schroeder, Robert Xiao, and Jordon Johnson

Announcements

- Google doc for lecture questions
  - See Piazza for link, section 102
  - https://docs.google.com/document/d/1G6hkekQS7mT9lFpP8AVftYao8vLRujIrRLAvOuX_07w/edit
  - Add your question anonymously (at the top)
  - Help answer questions too!

Overview

- Reading
  - Companion: 2.7.1-3, 2.7.5-6
  - Textbook: 3.6.1-5
- Learning goals
  - explain the role of the program counter for normal execution and for branch and jump instructions
  - compare the relative benefits of pc-relative and absolute addressing
  - explain why condition branch instructions are necessary for an ISA to be "Turing-complete"
  - translate a for loop that executes a static number of times into an equivalent, unrolled loop that contains no branch instructions
  - translate a for loop into equivalent C code that uses only if-then and goto statements for control flow
  - translate C code containing for loops into SM213 assembly language
  - identify for loops in SM213 assembly language and describe their semantics by writing an equivalent C for loop
  - translate an if-then-else statement into equivalent C code that uses only if-then and goto statements for flow control
  - translate C code containing if-then-else statements into SM213 assembly language
  - identify if-then-else statements in SM213 assembly language and describe their semantics by writing an equivalent C if-then-else statement
  - explain why a procedure's return address is a dynamic value
  - translate the control-flow portion of a C static procedure call into SM213 assembly
  - translate the control-flow portion of a C return statement into SM213 assembly
  - identify procedure calls and returns in SM213 assembly language and describe their semantics by writing equivalent C procedure call and return statements.

Control flow

- The flow of control is
  - the sequence of instruction executions performed by a program

- Every program execution can be described by such a linear sequence

Example – loops

in Java and C

```java
public class Foo {
  static int s = 0;
  static int i;
  static int a[] = new int[] {2, 4, 6, 8, 10, 12, 14, 16, 18, 20};

  static void foo() {
    for (i = 0; i < 10; i++)
      s += a[i];
  }
}
```

```c
int s = 0;
int i;
int a[] = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20};

void foo() {
  for (i = 0; i < 10; i++)
    s += a[i];
}
```

Alternative:

```c
for (i = 0; i < sizeof(a) / sizeof(a[0]); i++)
```

(only works with array variables, not with pointers)

Index vs pointer arithmetic

- Copying an array using array (square-bracket) syntax

```
void icopy(int* s, int* d, int n) {
  for (int i = 0; i < n; i++)
    d[i] = s[i];
}
```

- Copying an array using pointer arithmetic

```
void icopy(int* s, int* d, int n) {
  while (n--)
    *d++ = *s++;
}
```

d++;
s++;
*d = *s;

Implementing loops in SM213 assembly

```c
int s = 0;
int i;
int a[] = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20};

void foo() {
  for (i = 0; i < 10; i++)
    s += a[i];
}
```

- Can we implement *this* loop using our existing ISA so far?
  - i.e. `ld`, `st`, `mv`, `add`, `and`, `inc`, etc.

Yes, but not generally

iClicker 1d.1

- Which of the following loops can we implement with our existing ISA?
  - assume i is an int and n is a value supplied by the user (at runtime)

```
1. for (i = 0; i < 76; i++)
2. for (i = 23; i > 0; i--)
3. for (i = 0; i < n; i++)
4. for (i = 99; i > n; i--)
```

A. 1 and 3

B. 1 and 2

C. 3 and 4

D. 2 and 4

E. all of them

Loop unrolling

- This loop:

```
int s = 0;
int i;
int a[] = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20};

void foo() {
  for (i = 0; i < 10; i++)
    s += a[i];
}
```

- … is the same as this unrolled version:

```
int s = 0;
int i;
int a[] = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20};

void foo() {
  s += a[0];
  s += a[1];
  ...
  s += a[9];
}
```

Will this technique generalize?

only possible for statically-known number of iterations

Dissecting loops

using GOTO statements

- A simple example, assuming the compiler does not unroll it:

```
for (i = 0; i < n; i++)
  s += a[i];
```

  - in general, the number of iterations is not known statically, so the compiler cannot unroll it

- Using GOTO statements:

```
          i = 0
loop:     goto end_loop if not (i < n)
          s += a[i]
          i++
          goto loop
end_loop: // whatever code comes after the loop
```

Control flow in the machine

- Program counter (PC)
  - special CPU register that contains address of next instruction to execute

- For sequential instructions, PC is updated in the fetch stage
  - incremented by 2 or 6 depending on instruction size

- To "break" sequential execution, we need to change the PC from what it would normally contain
  - our "goto" commands will need to do this

Control flow: ISA extensions

- Conditional branches:
  - `goto <address> if <condition>`
  - RTL: `pc ← <address> if <condition>`

jump
branch
branch if equal
branch if greater than

```
j address
br address
beq r0, address
bgt r0, address
```

any general purpose register

- Options for evaluating condition:
  - Unconditional (always set PC to supplied address)
  - Conditional, based on the value of a register (`==0`, `>0`, etc.)
    - common in RISC, used in SM213
    - `goto <address> if <register> <condition> 0`
  - Conditional, based on result of last executed ALU instruction
    - CISC approach used in IA32 (x86) Intel architecture
    - `goto <address> if last ALU result <condition> 0`

Jump instruction size

- Problem: memory addresses are BIG
  - 32 bits in SM213, 64 bits in moderns ISAs
  - control flow instructions are common

```
j address
br address
beq r0, address
bgt r0, address
```

- Observation:
  - jumps usually move a short distance (forward or backward)
    - e.g. loops, if/else statements

PC-relative addressing

- PC-relative addressing
  - instead of specifying the destination address completely, specify the offset from the current location in code (in PC)
  - use the current value of PC (address of next instruction) as base address
    - remember that PC has already been updated during fetch phase
  - offset must be a signed number (can jump forward or backward)
  - Assembly still specifies the actual address/label
    - assembler converts address to offset
  - jumps that use PC-relative addressing are called branches

PC-relative addressing example

- Suppose we want to do something like this:

```
1000: goto 1008
1002: ...
1004: ...
1006: ...
1008: ...
```

← currently executing instruction

PC $\boxed{1002}$

goto 1008

offset: 1008 (destination)
− 1002 (current PC value)
+ 6

- Option 1: absolute addressing (jump)

$\underbrace{\texttt{X---}}_{\text{2 bytes}}\ \underbrace{\texttt{00001008}}_{\text{4 bytes}}$

but, this is a 6-byte instruction in our SM213 model

- Option 2: PC-relative addressing (branch)

opcode, offset

$\underbrace{\texttt{Y-06}}_{\text{2 bytes}}$

PC is 1002 (address of next instruction)
Target address is 1008 as specified in GOTO
so, offset to 1008 from 1002 is 6

  - since offsets are always even, we can expand our range by encoding offset / 2:

```
Y-03
```

divide actual offset by 2 when storing hardware instruction

ISA for static control flow

- We will need the following instructions:
  - at least one absolute jump
  - at least one PC-relative jump
    - specify relative distance using real distance / 2
    - PC-relative offset is a signed number
  - some conditional jumps (at least == 0 and > 0)
    - these should be PC-relative (why?) common, often don't move far away.

loop_start :

goto skip

half_loop goto loop_start

skip :

goto half_loop

- New instructions (so far):

Name	Semantics	Assembly	Machine
branch unconditional	`pc ← a (or pc + p*2)`	`br a`	`8-pp`
branch if equal conditional	`pc ← a (or pc + p*2) if r[c] == 0`	`beq rc, a`	`9cpp`
branch if greater	`pc ← a (or pc + p*2) if r[c] > 0`	`bgt rc, a`	`acpp`
jump unconditional	`pc ← a`	`j a`	`b--- aaaaaaaa`

iClicker 1d.2

- Convert the `bgt` instruction to machine code:

```
.pos 0x10
    bgt r0, L1
.pos 0x20
L1: halt
```

← currently executing

A. 0xa0 0x10
B. 0xa0 0x08
C. 0xa0 0x0e
$\boxed{\text{D.}}$ 0xa0 0x07
E. 0xa0 0x20

destination: 0x 20
PC : 0x12
actual offset 0x0E
encoded offset 0x07

Name	Semantics	Assembly	Machine
branch if greater	`pc ← a (or pc + p*2) if r[c] > 0`	`bgt rc, a`	`acpp`

iClicker 1d.3

- Convert the br instruction to machine code.
  - note: each instruction here is 2 bytes

```
loop: mov r0, r5         2
      add r4, r5         2
      beq r5, end_loop   2
      inc r0             2
      br  loop           2
```

x
x+2
x+4
x+6
x+8
PC x+10

actual offset: -10_{10}

+10 0 0 0 0 1 0 1 0

-10 1 1 1 1 0 1 1 0

$\underbrace{1\,1\,1\,1}_{f}\ \underbrace{0\,1\,1\,0}_{b}$

A. 0x80 0xf5
B. 0x80 0xf8
C. 0x80 0xfc
$\boxed{\text{D.}}$ 0x80 0xfb
E. 0x80 0xf6

Name	Semantics	Assembly	Machine
branch	`pc ← a (or pc + p*2)`	`br a`	`8-pp`

iClicker 1d.4

- What does the following instruction do?

```
0xa0 0x00
```

bgt r0, label

label

pc

A. infinite loop
B. sets PC to zero
C. sets PC to beginning of program
D. nothing
E. something else

Name	Semantics	Assembly	Machine
branch	`pc ← a (or pc + p*2)`	`br a`	`8-pp`
branch if equal	`pc ← a (or pc + p*2) if r[c] == 0`	`beq rc, a`	`9cpp`
branch if greater	`pc ← a (or pc + p*2) if r[c] > 0`	`bgt rc, a`	`acpp`
jump	`pc ← a`	`j a`	`b--- aaaaaaaa`

Implementing `for` loops

```
for (i=0; i<10; i++)
  s += a[i];
```

- General form
  - in C and Java

```
for (<init>; <continue-condition>; <step>) <statement-block>
```

    - each of init, continue, and step is optional or can be a compound expression

```
for (;;)
for (int i = 0, j = 10; i != j; i++, j--)
```

  - pseudo-code template

```
          <init>
loop:     goto end_loop if not <continue-condition>
          <statement-block>
          <step>
          goto loop
end_loop:
```

Implementing `for` loops

```
for (i=0; i<10; i++)
  s += a[i];
```

- By the template:

```
           i = 0
loop:      goto end_loop if not (i < 10)
           s += a[i]
           i++
           goto loop
end_loop:
```

- Our ISA does not support comparison to 10
  - but can compare to 0
  - and no need to store `i` (or `s`) in memory each loop iteration
    - use `i'` (or `temp_i`) to indicate this

valid, but only if compiler can prove that the loop body doesn't change the value of `i`

```
           a' = a
           s' = s
           i' = 0
loop:      t' = i' - 10
           goto end_loop if t' == 0
           s' += a'[i']
           i'++
           goto loop
end_loop:  s = s'
           i = i'
```

for loop to assembly

```
          a' = a
          s' = s
          i' = 0
loop:     t' = i' - 10
          goto end_loop if t' == 0
          s' += a'[i']
          i'++
          goto loop
end_loop: s = s'
          i = i'
```

#static

s: .long 0
i: .long 0

Registers

r0	i'
r1	a' (array address)
r2	s'
r3	a'[i']
r4	-10
r5	t'

Assembly: assume that all variables are global

```
          ld  $a, r1         # r1 = a = &a[0]
          ld  $s, r2         # r2 = &s
          ld  (r2), r2       # r2 = s = s'
          ld  $0x0, r0       # r0 = i' = 0
          ld  $-10, r4       # r4 = -10
loop:     mov r0, r5         # r5 = t' = i'
          add r4, r5         # r5 = i' - 10
          beq r5, end_loop # if i' = 10, goto +8
          ld  (r1, r0, 4), r3 # r3 = a'[i']
          add r3, r2         # s' += a'[i']
          inc r0             # i'++
          br loop            # goto -14
end_loop: ld  $s, r1         # r1 = &s
          st  r2, (r1)       # s = s'
          st  r0, 4(r1)      # i = i'
```

Implementing conditionals

if-then-else

```
if (a > b)
  max = a;
else
  max = b;
```

- General form
  - in Java and C:
    - `if <condition> <then-statements> else <else-statements>`
  - pseudo-code template

```
        c' = not <condition>
        goto then if (c' == 0)
else:   <else-statements>
        goto end_if
then:   <then-statements>
end_if:
```

or

```
        c' = <condition>
        goto then if (c' > 0)
else:   <else-statements>
        goto end_if
then:   <then-statements>
end_if:
```

Conditionals to assembly

$\underline{a - b} > 0$

if (a > b)
max = a;
else
max = b;

- pseudo-code template:

```
           a' = a
           b' = b
           c' = a' – b'
           goto then if (c' > 0)
else:      max' = b'
           goto end_if
then:      max' = a'
end_if:    max = max'
```

Registers

r0	a'
r1	b'
r2	c' = a - b
r3	max'

Assembly:

```
        ld   $a, r0       # r0 = &a
        ld   (r0), r0     # r0 = a
        ld   $b, r1       # r1 = &b
        ld   (r1), r1     # r1 = b
        mov r1, r2        # r2 = b
        not r2            # c' = !b
        inc r2            # c' = -b
        add r0, r2        # c' = a - b
        bgt r2, then      # if (a>b> goto then
else:   mov r1, r3        # max' = b
        br end_if         # goto end_if
then:   mov r0, r3        # max' = a
end_if: ld   $max, r0     # r0 = &max
        st   r3, (r0)     # max = max'
```

iClicker 1d.5

- What does this assembly code do?

```
    ld   $a, r0
    ld   (r0), r0
L0: deca r0
    bgt  r0, L0
    ld   $b, r1
    st   r0, (r1)
```

r0: a

a -= 4

goto L0 if a > 0

(will fail if a ≤ 0)

b = a;

a = ~~5~~ ~~X~~ -3

A. b = a - 4;

B. b = a % 4;

C. b = 0;

D. loops forever

E. something else

What does this code do?

```
.pos 0x1000
    ld  $0, r0
    ld  $0, r1
    ld  $1, r2
    ld  $j, r3
    ld  (r3), r3
    ld  $a, r4
L0: beq r3, L9
    ld  (r4, r0, 4), r5
    and r2, r5
    beq r5, L1
    inc r1
L1: inc r0
    dec r3
    br  L0
L9: ld  $o, r0
    st  r1, (r0)
    halt

.pos 0x2000
j:  .long 2
a:  .long 1
    .long 2
o:  .long 0
```

Step 1: comment the lines…

What does this code do?

```
.pos 0x1000
    ld  $0, r0           # r0 = 0
    ld  $0, r1           # r1 = 0
    ld  $1, r2           # r2 = 1
    ld  $j, r3           # r3 = &j
    ld  (r3), r3         # r3 = j = j' (j' is temp for j)
    ld  $a, r4           # r4 = a
L0: beq r3, L9           # goto L9 if j' == 0
    ld  (r4, r0, 4), r5  # r5 = a[r0]
    and r2, r5           # r5 = a[r0] & 1
    beq r5, L1           # goto L1 if (a[r0] & 1) == 0
    inc r1               # r1++ if (a[r0] & 1) != 0
L1: inc r0               # r0++
    dec r3               # j'--
    br  L0               # goto L0
L9: ld  $o, r0           # r0 = &o
    st  r1, (r0)         # o = r1
    halt

.pos 0x2000
j:  .long 2
a:  .long 1
    .long 2
o:  .long 0
```

Step 2: Refine the comments to C…

What does this code do?

```
.pos 0x1000
    ld   $0, r0            # r0 = 0 = i'
    ld   $0, r1            # r1 = 0 = o'
    ld   $1, r2            # r2 = 1
    ld   $j, r3            # r3 = &j
    ld   (r3), r3          # r3 = j = j'
    ld   $a, r4            # r4 = a
L0: beq r3, L9             # goto L9 if j' == 0
    ld   (r4, r0, 4), r5   # r5 = a[i']
    and r2, r5             # r5 = a[i'] & 1
    beq r5, L1             # goto L1 if (a[i'] & 1) == 0
    inc r1                 # o'++ if (a[i'] & 1) != 0
L1: inc r0                 # i'++
    dec r3                 # j'--
    br   L0                # goto L0
L9: ld   $o, r0            # r0 = &o
    st   r1, (r0)          # o = o'
    halt

.pos 0x2000
j:   .long 2
a:   .long 1
     .long 2
o:   .long 0
```

```
int i = 0;
int j = 2;
int a[2] = {1, 2};
int o;
```

Step 3: Look for basic blocks by examining branches

What does this code do?

```
.pos 0x1000
    ld   $0, r0            # r0 = 0 = i'
    ld   $0, r1            # r1 = 0 = o'
    ld   $1, r2            # r2 = 1
    ld   $j, r3            # r3 = &j
    ld   (r3), r3          # r3 = j = j'
    ld   $a, r4            # r4 = a
L0: beq r3, L9             # goto L9 if j' == 0
    ld   (r4, r0, 4), r5   # r5 = a[i']
    and r2, r5             # r5 = a[i'] & 1
    beq r5, L1             # goto L1 if (a[i'] & 1) == 0
    inc r1                 # o'++ if (a[i'] & 1) != 0
L1: inc r0                 # i'++
    dec r3                 # j'--
    br   L0                # goto L0
L9: ld   $o, r0            # r0 = &o
    st   r1, (r0)          # o = o'
    halt

.pos 0x2000
j:  .long 2
a:  .long 1
    .long 2
o:  .long 0
```

Loop

Conditional

```
int i = 0;
int j = 2;
int a[2] = {1, 2};
int o;
```

Step 4: Associate control structure with C

What does this code do?

```
.pos 0x1000
     ld   $0, r0           # r0 = 0 = i'
     ld   $0, r1           # r1 = 0 = o'
     ld   $1, r2           # r2 = 1
     ld   $j, r3           # r3 = &j
     ld   (r3), r3         # r3 = j = j'
     ld   $a, r4           # r4 = a
L0:  beq r3, L9            # goto L9 if j' == 0
     ld   (r4, r0, 4), r5  # r5 = a[i']
     and r2, r5            # r5 = a[i'] & 1
     beq r5, L1            # goto L1 if (a[i'] & 1) == 0
     inc r1                # o'++ if (a[i'] & 1) != 0
L1:  inc r0                # i'++
     dec r3                # j'--
     br   L0               # goto L0
L9:  ld   $o, r0           # r0 = &o
     st   r1, (r0)         # o = o'
     halt

.pos 0x2000
j:   .long 2
a:   .long 1
     .long 2
o:   .long 0
```

```
int i = 0;
int j = 2;
int a[2] = {1, 2};
int o;
```

Step 4: Associate control structure with C

```
for (j' = j; j' != 0; j'--) {

}
```

What does this code do?

```
.pos 0x1000
    ld   $0, r0           # r0 = 0 = i'
    ld   $0, r1           # r1 = 0 = o'
    ld   $1, r2           # r2 = 1
    ld   $j, r3           # r3 = &j
    ld   (r3), r3         # r3 = j = j'
    ld   $a, r4           # r4 = a
L0: beq r3, L9            # goto L9 if j' == 0
    ld   (r4, r0, 4), r5 # r5 = a[i']
    and r2, r5            # r5 = a[i'] & 1
    beq r5, L1            # goto L1 if (a[i'] & 1) == 0
    inc r1                # o'++ if (a[i'] & 1) != 0
L1: inc r0                # i'++
    dec r3                # j'--
    br   L0               # goto L0
L9: ld   $o, r0           # r0 = &o
    st   r1, (r0)         # o = o'
    halt

.pos 0x2000
j:   .long 2
a:   .long 1
     .long 2
o:   .long 0
```

```
int i = 0;
int j = 2;
int a[2] = {1, 2};
int o;
```

Step 4: Associate control structure with C

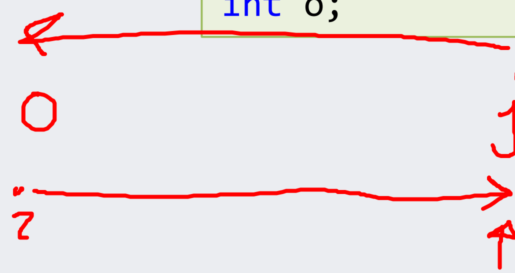
```
for (j' = j; j' != 0; j'--) {
  if (a'[i'] & 1)
    o++;
}
```

What does this code do?

```
.pos 0x1000
    ld   $0, r0           # r0 = 0 = i'
    ld   $0, r1           # r1 = 0 = o'
    ld   $1, r2           # r2 = 1
    ld   $j, r3           # r3 = &j
    ld   (r3), r3         # r3 = j = j'
    ld   $a, r4           # r4 = a
L0: beq r3, L9            # goto L9 if j' == 0
    ld   (r4, r0, 4), r5  # r5 = a[i']
    and r2, r5            # r5 = a[i'] & 1
    beq r5, L1            # goto L1 if (a[i'] & 1) == 0
    inc r1                # o'++ if (a[i'] & 1) != 0
L1: inc r0                # i'++
    dec r3                # j'--
    br   L0               # goto L0
L9: ld   $o, r0           # r0 = &o
    st   r1, (r0)         # o = o'
    halt

.pos 0x2000
j:  .long 2
a:  .long 1
    .long 2
o:  .long 0
```

```
int i = 0;
int j = 2;
int a[2] = {1, 2};
int o;
```



count the number of odd-valued elements in array a.

Step 5: Deal with what's left, bit by bit

```
for (j' = j, i' = 0; j' != 0; j'--, i'++) {
  if (a[i'] & 1)
    o++;
}
```

…and simplify

```
for (i = 0; i != j; i++) {
  if (a[i] & 1)
    o++;
}
```