# CPSC 213

## Introduction to Computer Systems

Summer Session 2019, Term 2

*Unit 1d – Jul 16*

## *Static Control Flow*

# 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 register 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
- Controlling flow in languages like Java

# Loops (S5-loop)

- In 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];
  }
}
```

- In 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];
}
```

# An Aside: Other ways to write 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];
}
```

`i < sizeof(a) / sizeof(a[0])`

Would this work if a was a dynamic array?

- Use pointer arithmetic instead of a[?] syntax

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

void foo() {
  int *ip = a;
  while(ip < a + sizeof(a) / sizeof(a[0]))
    s += *ip++;
}
```

What is this and why is it needed?

`s += *a++;` does not work, the value of a is static — a is a static array

- Sometimes pointer arithmetic has its uses
  - and its use is common in *hard-core* C programs
  - but usually it is better to access arrays using array syntax
- You decide
  - copying an array using array syntax

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

  - exactly the same thing, but with pointer arithmetic

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

# Implement loops in the machine

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

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

- Can we implement ***this*** loop with the existing ISA?

# Loop unrolling

- This loop

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

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

- Is the same as this *unrolled* version

```c
int s = 0;
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?

# Dissecting a Loop

- A simple example

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

  - assuming that the compiler doesn't unroll it
    - in general, the compiler will not know number of iterations and so it can't unroll it

- Using **goto**'s

```
            i=0;
loop:       if (i>=10) goto end_loop;
            s += a[i];
            i++;
            goto loop;
end_loop:
```

# Control-Flow ISA Extensions

- Conditional branches
  - goto <address> if <condition>
  - *pc ← <address> if <condition>*
- Options for evaluating condition
  - unconditional
  - conditional based on value of a register (==0, >0 etc.)
    - RISC approach that we will use
    - goto <address> if <register> <condition> 0
  - conditional check result of last executed ALU instruction
    - CISC approach used by IA32 (x86) Intel architecture
    - goto <address> if last ALU result <condition> 0

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

```
          i=0
loop:     if i-10 >= 0 goto end_loop
          s+=a[i]
          i++
          goto loop
end_loop:
```

```
bgt r0 end_loop
beq r0 end_loop
```

```
br loop
```

# Control-Flow in the Machine

- Program Counter (PC)
  - special CPU register that stores address of next instruction to execute
- Sequential execution

```
@Override protected void fetch() … {
      int             pcVal  = pc.get();
      UnsignedByte[] ins    = mem.read (pcVal, 2);
      …
      pcVal += 2;
      switch (opCode) {
       …
       case 0xb:
          ins [2..5] = mem.read (pcVal, 4)
           pcVal += 4;
           …
       …
      }
      pc.set(pcVal);
}
```

- And so **goto X** is really just
  - change the value of the PC register to X

```
pc.set(X);
```

# PC Relative Addressing

## Problem

- jump instructions that include target address are BIG instructions
  - 32-bits (in our ISA; 64 in modern ISAs) are needed for address
  - jump instruction will be 6 bytes
- and control-flow instructions are common

## Observation

- jumps inside of a procedure jump small distances from current location
  - i.e., loops, if statements etc.

## PC Relative Addressing

- specify the *offset* to the target address in the instruction
  - must be a signed number so that you can jump backward
- use the current value of program counter (PC) as base address
  - remember that PC stores the address of the NEXT SEQUENTIAL instruction
- in assembly language you still specify the actual address (usually a label)
  - assembler converts address to an offset
- jumps that use pc-relative addressing are called ***branches***

# PC Relative Addressing Example

- If we want to do something like this

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

- We could use absolute addressing like this

```
X--- 00001008
```

But, that's a 6-byte instruction in our machine

- Or PC relative addressing
  - like this

```
Y-06
```

PC is 1002 (address of next instruction)

Target address is 1008 as specified in GOTO

And so, offset from 1002 to get to 1008 is 6

  - but since offsets will always be even we can compress

```
Y-03
```

divide actual offset by 2 when storing in hardware instruction

# ISA for Static Control Flow (part 1)

- ISA requirement (apparently)
  - at least one PC-relative jump
    - specify relative distance using real distance / 2
    - pc-relative value is a signed number
  - at least one absolute jump
  - some conditional jumps (at least = and > 0)
    - make these PC-relative — why?
- New instructions (so far)

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

# Implementing *for* loops (S5-loop)

```
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:
```

## This example

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

- pseudo code template

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

- ISA suggests two transformations
  - only conditional branches we have compared to 0, not 10
  - no need to store i (or s) in memory in each loop iteration, so use `i'` (or temp_i) to indicate this

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

Only if compiler can prove that loop body doesn't change the value of i

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

| | |
|---|---|
| r0 | i' |
| r1 | a' |
| r2 | s' |
| r3 | a[i'] |
| r4 | -10 |
| r5 | t' |

- assembly code

Assume that all variables are global variables

```
            ld    $0x0, r0            # r0 = i' = 0
            ld    $a, r1              # r1 = a = &a[0]
            ld    $s, r2              # r2 = &s
            ld    (r2), r2            # r2 = s = s'
            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, 0x0(r1)         # s = s'
            st    r0, 0x4(r1)         # i = i'
```

# Implementing if-then-else (S6-if)

```
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 sometimes:**

c′ = <condition>

goto then if c′ > 0

## This example

- 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'
```

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

```
         goto then if a>b
else:    max = b
         goto end_if
then:    max = a
end_if:
```

- assembly code

```
         ld   $a, r0              # r0 = &a
         ld   0x0(r0), r0         # r0 = a
         ld   $b, r1              # r1 = &b
         ld   0x0(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 +2
else:    mov  r1, r3              # max' = b
         br   end_if              # goto +1
then:    mov  r0, r3              # max' = a
end_if:  ld   $max, r0            # r0 = &max
         st   r3, 0x0(r0)         # max = max'
```

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

# Reverse Engineering

```
.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   $x, r0
    st   r1, (r0)
    halt

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

Step 1: Comment the lines ...

# Comments added

```
    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   $x, r0             # r0 = &x
    st   r1, (r0)           # x = r1
```

Step 2: Refine the comments to C ...

# Comments Refined to C

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

```
     ld   $0, r0              # r0 = 0 = i'
     ld   $0, r1              # r1 = 0 = x'
     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                  # x'++ if if (a[i'] & 1) != 0
L1:  inc  r0                  # i'++
     dec  r3                  # j'--
     br   L0                  # goto L0
L9:  ld   $x, r0              # r0 = &o
     st   r1, (r0)            # x = x'
```

Step 3: Look for basic blocks by examining branches

# Basic blocks and Control Structure

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

```
    ld   $0, r0             # r0 = 0 = i'
    ld   $0, r1             # r1 = 0 = x’
    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                  # x'++ if if (a[i'] & 1) != 0
L1: inc r0                  # i'++
    dec r3                  # j'--
    br   L0                 # goto L0
L9: ld   $x, r0             # r0 = &x
    st   r1, (r0)           # x = x'
```

Step 4: Associate control structure with C

# What does this code do?

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

```
     ld   $0, r0              # r0 = 0 = i'
     ld   $0, r1              # r1 = 0 = x'
     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                  # x'++ if if (a[i'] & 1) != 0
L1:  inc  r0                  # i'++
     dec  r3                  # j'--
     br   L0                  # goto L0
L9:  ld   $x, r0              # r0 = &x
     st   r1, (r0)            # x = x'
```

Step 4: Associate control structure with C

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

}
```

# What does this code do?

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

```
    ld   $0, r0             # r0 = 0 = i'
    ld   $0, r1             # r1 = 0 = x'
    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                 # x'++ if if (a[i'] & 1) != 0
L1: inc  r0                 # i'++
    dec  r3                 # j'--
    br   L0                 # goto L0
L9: ld   $x, r0             # r0 = &x
    st   r1, (r0)           # x = x'
```

Step 4: Associate control structure with C

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

# What does this code do?

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

```
    ld   $0, r0              # r0 = 0 = i'
    ld   $0, r1              # r1 = 0 = x'
    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                  # x'++ if if (a[i'] & 1) != 0
L1: inc  r0                  # i'++
    dec  r3                  # j'--
    br   L0                  # goto L0
L9: ld   $x, r0              # r0 = &x
    st   r1, (r0)            # x = x'
```

Step 5: Deal with what is left, bit by bit

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

and simplify

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

# Static Procedure Calls

# Code Examples (S7-static-call)

```java
public class A {
  static void ping() {}
}

public class Foo {
  static void foo() {
    A.ping();
  }
}
```

```c
void ping() {}

void foo() {
  ping();
}
```

- Java
  - a ***method*** is a sub-routine with a name, arguments and local scope
  - method ***invocation*** causes the sub-routine to run with values bound to arguments and with a possible result bound to the invocation
- C
  - a ***procedure*** is ...
  - a procedure ***call*** is ...

# Diagraming a Procedure Call

```
void foo() {
  ping();
}
```

```
void ping() {}
```

- Caller
  - goto ping
    - `j ping`
  - continue executing
- Callee
  - do whatever ping does
  - goto foo just after call to ping()
    - ??????

Questions

How is RETURN implemented?

It's a jump, but is the address a static property or a dynamic one?

# Implementing Procedure Return

- return address is
  - the address the procedure jumps to when it completes
  - the address of the instruction following the call that caused it to run
  - a dynamic property of the program
- questions
  - how does procedure know the return address?
  - how does it jump to a dynamic address?

- saving the return address
  - only the caller knows the address
  - so the caller must save it before it makes the call
    - caller will save the return address in **r6**
      - there is a bit of a problem here if the callee makes a procedure call, more later ...
  - we need a new instruction to read the PC
    - we'll call it `gpc`
- jumping back to return address
  - we need new instruction to jump to an address stored in a register
    - callee can assume return address is in r6

# ISA for Static Control Flow (part 2)

- New requirements
  - read the value of the PC
  - jump to a dynamically determined target address
- Complete new set of instructions

| Name | Semantics | Assembly | Machine |
|---|---|---|---|
| *branch* | `pc ← (a==pc+pp*2)` | `br a` | `8-pp` |
| *branch if equal* | `pc ← (a==pc+pp*2) if r[c]==0` | `beq a` | `9cpp` |
| *branch if greater* | `pc ← (a==pc+pp*2) if r[c]>0` | `bgt a` | `acpp` |
| *jump* | `pc ← a` | `j a` | `b--- aaaaaaaa` |
| *get pc* | `r[d] ← pc + (o==p*2)` | `gpc $o,rd` | `6fpd` |
| *indirect jump* | `pc ← r[t] + (o==pp*2)` | `j o(rt)` | `ctpp` |

**Note:** offset o == p*2 in indirect jump is **unsigned**.

# Compiling Procedure Call / Return

```
void foo() {
  ping();
}
```

```
foo:    gpc   $6, r6          # r6 = pc of next instruction
        j     ping            # goto ping ()
        …
```

```
void ping() {}
```

```
ping:   j     (r6)            # return
```