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
  ```java
  public class Foo {
    static int s = 0;
    static int i;
    static int a[] = new int[10];
    static void foo () {
      for (i=0; i<10; i++)
        s += a[i];
    }
  }
  ```

- In C
  ```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];
  }
  ```

Reading

- Companion
  - 2.7.1-2.7.3, 2.7.5-2.7.6
- Textbook
  - 3.6.1-3.6.5
Implement loops in machine

Can we implement this loop with the existing ISA?

```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];
}
```

Loop unrolling

Using array syntax

Using pointer-arithmetic syntax for access to a?

Will this technique generalize

• will it work for all loops? why or why not?

Control-Flow ISA Extensions

Conditional branches
• goto <address> if <condition>

Options for evaluating condition
• unconditional
• conditional based on value of a register (==0, >0 etc.)
  • goto <address> if <register> <condition> 0
• conditional check result of last executed ALU instruction
  • goto <address> if last ALU result <condition> 0

Specifying target address
• absolute 32-bit address
  • this requires a 6 byte instruction, which means jumps have high overhead
  • is this a serious problem? how would you decide?
  • are jumps for for/while/if etc. different from jumps for procedure call?

PC Relative Addressing

Motivation
• jumps are common and so we want to make them as fast as possible
• small instructions are faster than large ones, so make some jumps be two bytes

Observation
• some jumps such as for/while/if etc. normally jump to a nearby instruction
• so the jump distance can be described by a small number that could fit in a byte

PC Relative Addressing
• specifies jump target as a delta from address of current instruction (actually next)
• in the execute stage pc register stores the address of next sequential instruction
• the pc-relative jump delta is applied to the value of the pc register
  • jumping with a delta of 0 jumps to the next instruction
• jump instructions that use pc-relative addressing are called branches

Absolute Addressing
• specifies jump target using full 32-bit address
• use when the jump distance too large to fit in a byte
ISA for Static Control Flow (part 1)

ISA requirement (apparently)
- at least one PC-relative jump
  - specify relative distance using real distance / 2 — why?
- at least one absolute jumps
- some conditional jumps (at least = and > 0)
  - make these PC-relative — why?

New instructions (so far)
- jump assembly uses label, not direct hex number
- PC-relative count starts from next instruction, after fetch increments PC

<table>
<thead>
<tr>
<th>Name</th>
<th>Semantics</th>
<th>Assembly</th>
<th>Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch</td>
<td>pc ← (a=pc+oo*2)</td>
<td>br a</td>
<td>8-oo</td>
</tr>
<tr>
<td>branch if equal</td>
<td>pc ← (a=pc+oo*2) if r[c]==0</td>
<td>beq rc, a</td>
<td>9coo</td>
</tr>
<tr>
<td>branch if greater</td>
<td>pc ← (a=pc+oo*2) if r[c]&gt;0</td>
<td>bgt rc, a</td>
<td>acoo</td>
</tr>
<tr>
<td>jump immediate</td>
<td>pc ← a (a specified as label)</td>
<td>j a</td>
<td>b~~~ aaaaaaaa</td>
</tr>
</tbody>
</table>

- jump assembly uses label, not direct hex number
- PC-relative count starts from next instruction, after fetch increments PC

This example
- pseudo code template
  ```
  i=0
  loop:   if not (i<10)  goto end_loop
          s += a[i]
          i++
          goto loop
  end_loop:
  ```

- ISA suggest two transformations
  - only conditional branches we have compared to 0, not 10
  - no need to store i and s in memory in each loop iteration, so use temp_ to indicate this

  ```
  temp_i=0
  temp_s=0
  loop:   temp_t=temp_i-9
          if temp_t>0 goto end_loop
          temp_s+=a[temp_i]
          temp_i++
          goto loop
  end_loop: s=temp_s
            i=temp_i
  ```

Implementing for loops ($S5$-loop)

General form
- in C and Java
  ```
  for (<init>; <continue-condition>; <step>) <statement-block>
  ```

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

Assembly code
Assume that all variables are global variables
  ```
  temp_i=0
  temp_s=0
  loop:   temp_t=temp_i-9
          if temp_t>0 goto end_loop
          temp_s+=a[temp_i]
          temp_i++
          goto loop
  end_loop: s=temp_s
            i=temp_i
  ```

  ```
  ld $0x0, r0    # r0 = temp_i = 0
  ld $a, r1     # r1 = address of a[0]
  ld $0x0, r2    # r2 = temp_s = 0
  ld $0xffffffff7, r4  # r4 = -9
  loop:   mov r0, r5    # r5 = temp_i
          add r4, r5    # r5 = temp_i-9
          bgt r5, end_loop    # if temp_i>9 goto +4
          ld (r1, r0, 4), r3  # r3 = a[temp_i]
          add r3, r2    # temp_s += a[temp_i]
          inc r0     # temp_i++
          br loop    # goto -7
  end_loop: ld $s, r1   # r1 = address of s
            st r2, 0x0(r1)  # s = temp_s
            st r0, 0x4(r1)  # i = temp_i
  ```
Implementing if-then-else (S6-if)

General form
• in Java and C
  - if <condition> <then-statements> else <else-statements>
• pseudo-code template
  ```
  temp_c = not <condition>
  goto then if (temp_c==0)
  else:   <else-statements>
          goto end_if
  then:   <then-statements>
  end_if:
  ```

This example
• pseudo-code template
  ```
  temp_a=a
  temp_b=b
  temp_c=temp_a-temp_b
  goto then if (temp_c>0)
  else:   temp_max=temp_b
          goto end_if
  then:   temp_max=temp_a
  end_if: max=temp_max
  ```

• 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
  # temp_c = ! b
  inc  r2
  # temp_c = - b
  add  r0, r2
  # temp_c = a-b
  bgt  r2, then
  # if (a>b) goto +2
  else:   mov  r1, r3
  # temp_max = b
  br   end_if
  # goto +1
  then:   mov  r0, r3
  # temp_max = a
  end_if: ld   $max, r0
  # r0 = &max
  st   r3, 0x0(r0)
  # max = temp_max
  ```

Code Examples (S6-static-call)

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

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

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

C
• a **procedure** is ...
• a procedure **call** is ...

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Semantics</th>
<th>Assembly</th>
<th>Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch</td>
<td>pc ← (a==pc+pp*2)</td>
<td>br a</td>
<td>8-pp</td>
</tr>
<tr>
<td>branch if equal</td>
<td>pc ← (a==pc+pp*2) if r[c]==0</td>
<td>beq a</td>
<td>9-cpp</td>
</tr>
<tr>
<td>branch if greater</td>
<td>pc ← (a==pc+pp*2) if r[c]&gt;0</td>
<td>bgt a</td>
<td>acpp</td>
</tr>
<tr>
<td>jump immediate</td>
<td>pc ← a (a specified as label)</td>
<td>j a</td>
<td>b-aaa</td>
</tr>
<tr>
<td>get pc</td>
<td>r[d] ← pc + (o==p*2)</td>
<td>gpc $o,rd</td>
<td>6-fpd</td>
</tr>
<tr>
<td>jump base+offset</td>
<td>pc ← r[t] + (o==pp*2)</td>
<td>j o(rt)</td>
<td>c-tp</td>
</tr>
</tbody>
</table>

• jump assembly uses label, not direct hex number
Compiling Procedure Call / Return

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

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

ping:  j    (r6)          # return
```