Introduction

- SPARC is a load/store architecture
- Registers used for all arithmetic and logical operations
- 32 registers available at a time
- Uses only **load** and **store** instructions to access memory
Registers

- Registers are accessed directly for rapid computation
- 32 registers – divided into 4 sets
  -- Global: %g0-%g7
  -- Out: %o0 – %o7
  -- In: %i0 – %i7
  -- Local: %l0 – %l7
- %g0 – always returns 0
- %o6, %o7, %i6, %i7 – do not use
- Register size = 32 bits each
# Table of Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Synonyms</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r0</td>
<td>%r0</td>
<td>Always discards writes and returns zero</td>
</tr>
<tr>
<td>%r1</td>
<td>%r1</td>
<td>First of seven registers for data with global context</td>
</tr>
<tr>
<td>%r2</td>
<td>%r2</td>
<td></td>
</tr>
<tr>
<td>%r3</td>
<td>%r3</td>
<td></td>
</tr>
<tr>
<td>%r4</td>
<td>%r4</td>
<td></td>
</tr>
<tr>
<td>%r5</td>
<td>%r5</td>
<td></td>
</tr>
<tr>
<td>%r6</td>
<td>%r6</td>
<td></td>
</tr>
<tr>
<td>%r7</td>
<td>%r7</td>
<td></td>
</tr>
<tr>
<td>%r8</td>
<td>%r8</td>
<td>First of six registers for local data and arguments to called subroutines</td>
</tr>
<tr>
<td>%r9</td>
<td>%r9</td>
<td></td>
</tr>
<tr>
<td>%r10</td>
<td>%r10</td>
<td></td>
</tr>
<tr>
<td>%r11</td>
<td>%r11</td>
<td></td>
</tr>
<tr>
<td>%r12</td>
<td>%r12</td>
<td></td>
</tr>
<tr>
<td>%r13</td>
<td>%r13</td>
<td></td>
</tr>
<tr>
<td>%r14</td>
<td>%r14</td>
<td></td>
</tr>
<tr>
<td>%r15</td>
<td>%r15</td>
<td></td>
</tr>
<tr>
<td>%r16</td>
<td>%r16</td>
<td></td>
</tr>
<tr>
<td>%r17</td>
<td>%r17</td>
<td></td>
</tr>
<tr>
<td>%r18</td>
<td>%r18</td>
<td></td>
</tr>
<tr>
<td>%r19</td>
<td>%r19</td>
<td></td>
</tr>
<tr>
<td>%r20</td>
<td>%r20</td>
<td></td>
</tr>
<tr>
<td>%r21</td>
<td>%r21</td>
<td></td>
</tr>
<tr>
<td>%r22</td>
<td>%r22</td>
<td></td>
</tr>
<tr>
<td>%r23</td>
<td>%r23</td>
<td></td>
</tr>
<tr>
<td>%r24</td>
<td>%r24</td>
<td>First of six registers for incoming subroutine arguments</td>
</tr>
<tr>
<td>%r25</td>
<td>%r25</td>
<td></td>
</tr>
<tr>
<td>%r26</td>
<td>%r26</td>
<td></td>
</tr>
<tr>
<td>%r27</td>
<td>%r27</td>
<td></td>
</tr>
<tr>
<td>%r28</td>
<td>%r28</td>
<td></td>
</tr>
<tr>
<td>%r29</td>
<td>%r29</td>
<td></td>
</tr>
<tr>
<td>%r30</td>
<td>%r30, %r16</td>
<td>Frame pointer</td>
</tr>
<tr>
<td>%r31</td>
<td>%r31</td>
<td>Subroutine return address</td>
</tr>
</tbody>
</table>
SPARC Assembler

- SPARC assembler as: 2-pass assembler
- First pass:
  - Updates location counter without paying attention to undefined labels for operands
  - Defines label symbol to location counter
- Second pass:
  - Values substituted in for labels
  - Ignores labels followed by colons
Assembly Language Programs

- Programs are line based
- Use mnemonics which generate machine code upon assembling
- Statements may be labeled
- Comments: ! or /* ... */

```c
/* instructions to add and to subtract the contents of %o0 and %o1 */

start:  add  %o0, %o1, %l0    !l0=o0+o1
        sub  %o0, %o1, %l1    !l1=o0-o1
```
Statements that do not generate machine code
- e.g. Data definitions, statements to provide the assembler information

Generally start with a period

```
a: .word 3
```

Can be labeled

```
.global main
main:
```
Compiling Code – 2 step process

- C compiler will call `as` and produce the object files
- Object files are the machine code
- Next calls the linker to combine `.o` files with library routines to produce the executable program – `a.out`
Compiling a C program

%gcc -S program.c : produces the .s assembly language file

%gcc expr.s -o expr : assembles the program and produces the executable file

NOTE: You will only do this for the 1st assignment
Start of Execution

- C compiler expects to start execution at an address main
- The label must be at the first statement to execute and declared to be global
  ```
  .global main
  main:  save %sp, -96, %sp
  ```
- `save` instruction provides space to save registers for the debugger
If we have macros defined, then the program should be a `.m` file.

We can expand the macros to produce a `.s` file by running `m4` first.

```
% m4 expr.m > expr.s
% gcc expr.s -o expr
```
3 operands: 2 source operands and 1 destination operand

Source registers are unchanged

Result stored in destination register

Constants: $-4096 \leq c < 4096$

\begin{align*}
\text{op} & \quad \text{reg}_{rs1}, \text{reg}_{rs2}, \text{reg}_{rd} \\
\text{op} & \quad \text{reg}_{rs1}, \text{imm}, \text{reg}_{rd}
\end{align*}
Sample Instructions

clr reg_{rd}
- Clears a register to zero

mov reg_or_imm, reg_{rd}
- Copies content of source to destination

add reg_{rs1}, reg_or_imm, reg_{rd}
- Adds oper1 + oper2 → destination

sub reg_{rs1}, reg_or_imm, reg_{rd}
- Subtracts oper1 - oper2 → destination
Multiply and Divide

- No instruction available in SPARC
- Use function call instead
- Must use %o0 and %o1 for sources and %o0 holds result

```
mov    b, %o0
mov    c, %o1
call    .mul
a = b * c

mov    b, %o0
mov    c, %o1
call    .div
a = b ÷ c
```
Instruction Cycle

- Instruction cycle broken into 4 stages:
  - **Instruction fetch**
    - Fetch & decode instruction, operands, update PC
  - **Execute**
    - Execute arithmetic instruction, branch target address, address
  - **Memory access**
    - Access memory for load or store instruction; fetch instruction branch instruction
  - **Store results**
    - Write instruction results back to register file
Pipelining

- SPARC is a RISC machine – want to complete **one instruction per cycle**
- Overlap stages of different instructions to achieve parallel execution
- Can obtain a speedup by a factor of 4
- Hardware does not have to run 4 times faster – break h/w into 4 parts to run **concurrently**
Laundry Analogy – wash / dry / iron

- Wash 3 loads sequentially
  - Use only one m/c at a time

- Wash 3 loads in pipeline fashion
  - Use multiple m/c at a time
Pipelining

- **Sequential**: each h/w stage idle 75% of the time
  \[ \text{time}_{ex} = 4 \times i \]
- **Parallel**: each h/w stage working after filling the pipeline
  \[ \text{time}_{ex} = 3 + i \]

Example: for 100 instr:
- 400 vs 103 cycles

![Diagram](image.png)

**Figure 2.1: Pipelined Execution**
Data Dependencies – Load Delay Problem

load [%o0], %o1
add %o1, %o2, %o2
Branch Delay Problem

- Branch target address not available until after execution of branch instruction
- Insert branch delay slot instruction

Figure 2.3: Branch Delay Slot
Branch delays

- Try to place an instruction after the branch that is useful – can also use `nop`
- The instruction following a branch instruction will always be fetched
- Updating the PC determines which instruction to fetch next
cmp %l0, %l1
bg next
mov %l2, %l3
sub %l3, 20, %l4

Condition **true**: branch to **next**

Condition **false**: continue to **sub**
/* This program computes the expression: \( y = (x - 1) \times (x - 7) \div (x - 11) \) for \( x = 9 \)

The polynomial coefficients are:
 */

define(a2, 1)
define(a1, 7)
define(a0, 11)

/* Variables \( x \) and \( y \) are stored in %l0 and %l1 */

define(x_r, l0)
define(y_r, l1)

.global main

main:
save %sp, -96, %sp
mov 9, %x_r
sub %x_r, a2, %o0
sub %x_r, a1, %o1
call .mul
nop
sub %x_r, a0, %o1
call .div
nop
mov %o0, %y_r
mov 1, %g1
mov 0, %ta
Expanding Macros

- After running through m4: `m4 expr.m > expr.s`
- Produce executable: `gcc expr.s -o expr`
- Execute file: `./expr`

```assembly
.global main
main:
save %sp, -96, %sp
mov 9, %l0 !initialize x
sub %l0, 1, %o0 !(x - a2) into %o0
sub %l0, 7, %o1 !(x - a1) into %o1
call .mul
nop !result in %o0
sub %l0, 11, %o1 !(x - a0) into %o1, the divisor
call .div
nop !result in %o0
mov %o0, %l1 !store it in y
mov 1, %g1 !trap dispatch
ta 0 !trap to system
```
Filling Delay Slots

- The call instruction is called a *delayed control transfer instruction*: changes address from where future instructions will be fetched.
- The following instruction is called a *delayed instruction*, and is located in the *delay slot*.
- The delayed instruction is executed before the branch/call happens.
- By using a *nop* for the delay slot – still wasting a cycle.
- Instead, we may be able to move the instruction prior to the branch instruction into the delay slot.
Move sub instructions to the delay slots to eliminate nop instructions

.global main
main:
save %sp, -96, %sp
mov 9, %l0  !initialize x
sub %l0, 1, %o0  !(x - 1) into %o0
call .mul
sub %l0, 7, %o1  !(x - 7) into %o1
call .div
sub %l0, 11, %o1  !(x - 11) into %o1, the divisor
mov %o0, %l1  !store it in y
mov 1, %g1  !trap dispatch
ta 0  !trap to system
Filling Delay Slots

- Executing the `mov` instruction, while fetching the `sub` instruction

```
.global main
main:
  save %sp, -96, %sp
  mov 9, %l0  !initialize x
  sub %l0, 1, %o0  !(x - 1) into %o0
  call .mul
  sub %l0, 7, %o1  !(x - 7) into %o1
  call .div
  sub %l0, 11, %o1  !(x - 11) into %o1, the divisor
  mov %o0, %l1  !store it in y
  mov 1, %g1  !trap dispatch
  ta 0  !trap to system
```
Now executing the `sub` instruction, while fetching the `call` instruction

```assembly
.global main
main:
save  %sp, -96, %sp
mov   9, %l0  !initialize x
sub   %l0, 1, %o0  !(x - 1) into %o0
call  .mul
sub   %l0, 7, %o1  !(x - 7) into %o1
call  .div
sub   %l0, 11, %o1  !(x - 11) into %o1, the divisor
mov   %o0, %l1   !store it in y
mov   1, %g1   !trap dispatch
ta    0   !trap to system
```
Now executing the `call` instruction, while fetching the `sub` instruction

```
.global main
main:
save    %sp, -96, %sp
mov     9, %l0  !initialize x
sub     %l0, 1, %o0  !(x - 1) into %o0
call    .mul
sub     %l0, 7, %o1  !(x - 7) into %o1
mov     %o0, %l1   !store it in y
mov     1, %g1   !trap dispatch
ta      0     !trap to system
```

Execution of `call` will update the PC to fetch from `mul` routine, but since `sub` was already fetched, it will be executed before any instruction from the `mul` routine.
Now executing the `sub` instruction, while fetching from the `mul` routine.

```
.global main
main:
    save  %sp, -96, %sp
    mov   9, %l0  !initialize x
    sub   %l0, 1, %o0  !(x - 1) into %o0
    call  .mul
    sub   %l0, 7, %o1  !(x - 7) into %o1
    sub   %l0, 11, %o1  !(x - 11) into %o1, the divisor
    mov   %o0, %l1  !store it in y
    mov   1, %g1  !trap dispatch
    ta    0  !trap to system
        ......
.mul:
    save .....  ......
```

EXECUTE →

FETCH →
Now executing the `save` instruction, while fetching the next instruction from the `mul` routine

```assembly
.global main
main:
save  %sp, -96, %sp
mov   9, %l0  !initialize x
sub   %l0, 1, %o0  !(x - 1) into %o0
call  .mul
sub   %l0, 7, %o1  !(x - 7) into %o1

call  .div
sub   %l0, 11, %o1  !(x - 11) into %o1, the divisor
mov   %o0, %l1  !store it in y

mov   1, %g1  !trap dispatch
ta    0  !trap to system
......
.mul:
save ....
......
```
While executing the last instruction of the `mul` routine, will come back to `main` and fetch the `call .div` instruction.

```
.global main
main:
save %sp, -96, %sp
mov 9, %l0  !initialize x
sub %l0, 1, %o0  !(x - 1) into %o0
call .mul
sub %l0, 7, %o1  !(x - 7) into %o1
call .div
sub %l0, 11, %o1   !(x - 11) into %o1, the divisor
mov %o0, %l1   !store it in y
mov 1, %g1   !trap dispatch
ta 0   !trap to system
```

At this point %00 has the result from the multiply routine – this is the first operand for the divide routine.

The subtract instruction will compute the 2\textsuperscript{nd} operand before starting execution of the divide routine.
Branching Instructions

- Delayed control transfer instructions
- Branch target address not available until after execution of branch instruction
- Can be conditional or unconditional
- If conditional – test the condition codes (flags)
- Operand of instruction is the target label

```
b ICC label
```
Testing

- Able to test information about a previous result
- State of execution saved in 4 integer codes or flags:
  - **Z**: whether the result was zero
  - **N**: whether the result was negative
  - **V**: whether execution of the result caused an overflow
  - **C**: whether execution of the result generated a carry out
Updating condition codes

- Certain instructions can be modified to update the condition codes
- Append “cc” to the instruction to save state of execution

```
addcc reg_{rs1}, reg_or_imm, reg_{rd}
subcc reg_{rs1}, reg_or_imm, reg_{rd}
```
<table>
<thead>
<tr>
<th>Assembler Mnemonic</th>
<th>Unconditional Branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>ba</td>
<td>Branch always, goto</td>
</tr>
<tr>
<td>bn</td>
<td>Branch never</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Assembler Mnemonic</th>
<th>Conditional Branches (signed arithmetic)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bl</td>
<td>Branch on less than zero</td>
</tr>
<tr>
<td>ble</td>
<td>Branch on less than or equal to zero</td>
</tr>
<tr>
<td>be</td>
<td>Branch on equal to zero</td>
</tr>
<tr>
<td>bne</td>
<td>Branch on not equal to zero</td>
</tr>
<tr>
<td>bge</td>
<td>Branch on greater than or equal to zero</td>
</tr>
<tr>
<td>bg</td>
<td>Branch on greater than zero</td>
</tr>
</tbody>
</table>
We can extend our program to compute $y$ for an input range of $0 \leq x \leq 10$

First consider a C program and then we will translate to assembly

**Do-while loop used**

```c
main ()
{
    int x, y;
    x = 0;
    do {
        y = ((x-1)*(x-7))/(x-11);
        x++;
    } while (x < 11);
}
```
define (x_r, l0)
define (y_r, l1)

.global main

main:  save %sp, -96, %sp
cr %x_r

loop: sub %x_r, 1, %o0  !(x-1)
call .mul
sub %x_r, 7, %o1  !(x-7)
call .div
sub %x_r, 11, %o1  !(x-11)
mov %o0, %y_r  !store result
add %x_r, 1, %x_r  !x++
subcc %x_r, 11, %g0  !set condition codes
bl  loop

nop  fills delay slot after the branch

nop after the call – sub fills delay slot

ret
restore
define (x_r, l0)
define (y_r, l1)
.global main

main: save %sp, -96, %sp
clr %x_r   !initialize x
loop: sub %x_r, 1, %o0  !(x-1)
call .mul
sub %x_r, 7, %o1  !(x-7)
call .div
sub %x_r, 11, %o1  !(x-11)
mov %o0, %y_r  !store result
add %x_r, 1, %x_r  !x++
subcc %x_r, 11, %g0  !set condition codes
bl loop
nop
ret    !ends program
restore

Can we use subbccc to fill the delay slot?
define (x_r, l0)
define (y_r, l1)
.global main

main: save %sp, -96, %sp
clr %x_r  !initialize x

loop: sub %x_r, 1, %o0  !(x-1)
call .mul
sub %x_r, 7, %o1  !(x-7)
call .div
sub %x_r, 11, %o1  !(x-11)
mov %o0, %y_r  !store result
add %x_r, 1, %x_r  !x++
subcc %x_r, 11, %g0  !set condition codes
bl loop
nop
ret  !ends program
restore

Can we use add to fill the delay slot?
define (x_r, l0)
define (y_r, l1)
.globl main

main:  save  %sp, -96, %sp
       clr  %x_r  !initialize x

loop:  sub  %x_r, 1, %o0  !(x-1)
call   .mul
sub    %x_r, 7, %o1  !(x-7)
call   .div
sub    %x_r, 11, %o1  !(x-11)
mov    %o0, %y_r  !store result
add    %x_r, 1, %x_r  !x++

subcc  %x_r, 11, %g0  !set condition codes
bl     loop
nop
ret
restore

Can we use mov to fill the delay slot?
define (x_r, 10)
define (y_r, 11)

.global main

main: save %sp, -96, %sp
clr %x_r !initialize x

loop: sub %x_r, 1, %o0 !(x-1)
call .mul
sub %x_r, 7, %o1 !(x-7)
call .div
sub %x_r, 11, %o1 !(x-11)
add %x_r, 1, %x_r !x++
subcc %x_r, 11, %g0 !set condition codes
bl loop
mov %o0, %y_r !store result
ret !ends program

restore

---

Revised code with nops removed

initialize x
(x-1)
(x-7)
(x-11)
x++
set condition codes
store result
ends program
Synthetic Instructions

- Easier for us to read and understand logically
- Assembler automatically substitutes synthetic instruction
- Similar to the way macros work and are updated
Example Synthetic Instructions

1) $\text{cmp } \text{reg}_{rs1}, \text{ reg\_or\_imm}$
   \[\text{same as}\]
   $\text{subcc } \text{reg}_{rs1}, \text{ reg\_or\_imm}, %g0$

2) $\text{inc } \text{reg}$
   \[\text{same as}\]
   $\text{add } \text{reg, 1, reg}$
Original code without synthetic instructions

add %x_r, 1, %x_r
subcc %x_r, 11, %g0
bl loop
mov %o0, %y_r

Code with synthetic instructions

inc %x_r
cmp %x_r, 11
bl loop
mov %o0, %y_r

Note: .m program will include the synthetic instructions, but machine code will correspond to the substitution instructions on left hand side
When writing assembly programs, always first write pseudo-code, then translate.

Consider the following control structures first in C and then map to assembly:

-- do while loop
-- for loop
-- while loop
-- if-then / if-then-else
While – check condition first

while ( a <= 17)  
{  a = a + b;  
c++;  }

Now see if we can optimize the code...
We test at start of loop to see if we should enter the body of the loop.
Then we have a `ba` with `nop` at end of the loop.
This brings us back up to start to test again.
We do have to test before entering the first time, but then we can just test at end to see if we should get back into the loop.
This will take away one `branch/nop` in body of the loop.
Revised While Loop

test:
  cmp       %a_r, 17

  bg done
  nop

loop:
  add       %a_r, %b_r, %a_r
  inc       %c_r
  cmp       %a_r, 17
  ble       test
  nop

done:

notice it is ble not bg now
### Annulled Conditional Branches

- Rather than using a nop after a conditional branch at the end of a loop, we can fill it with the first instruction of the loop.
- If annulled, then the delay instruction is only executed if the condition is true – i.e. if going back into the loop.
- If condition is false, the delay instruction is still fetched, but annulled instead of executed.
While Loop with Annulled Conditional Branch

```
  ba  test  !check if we should enter
  nop  !loop the first time
loop:
  inc  %c_r
  test:
      cmp  %a_r, 17 !testing condition
      ble,a  loop  !if true go back to loop & pick
      add  %a_r, %b_r, %a_r  !up add instr on the way
```
Do Loop – execute at least once

**Steps to modify loop:**
- Pull 1st instr out of loop
- Repeat 1st instr in delay slot
- Annul the branch
- Change target to 2\(^{nd}\) instr

Previous code:
```
loop:
    sub   %x_r, 1, %o0
    call   .mul
    sub   %x_r, 7, %o1
    call   .div
    sub    %x_r, 11, %o1
    mov   %o0, %y_r
```

**Annulled branch**
```
loop:
    sub   %x_r, 1, %o0
    loop:
    call   .mul
    sub   %x_r, 7, %o1
    call   .div
    sub    %x_r, 11, %o1
    mov   %o0, %y_r
```
Do Loop – execute at least once

\[ y = \frac{(x - 1)(x - 7)}{(x - 11)} \]

Previous code

```assembly
loop:
    sub %x_r, 1, %o0
    call .mul
    sub %x_r, 7, %o1
    call .div
    sub %x_r, 11, %o1
    add %x_r, 1, %x_r
    cmp %x_r, 11
    bl loop
    mov %o0, %y_r
```

Using annulled branch

```assembly
sub %x_r, 1, %o0
loop:
    call .mul
    sub %x_r, 7, %o1
    call .div
    sub %x_r, 11, %o1
    mov %o0, %y_r
    add %x_r, 1, %x_r
    cmp %x_r, 11
    bl,a loop
    sub %x_r, 1, %o0
```
For Loop

- Syntax in C:
  
  ```c
  for ( ex1; ex2; ex3 ) st
  ```

- Define it as follows:

  ```c
  ex1;
  while ( ex2 ) {
    st
    ex3;
  }
  ```

A for loop is nothing more than a while loop.
For Loop Example

```plaintext
for (a=1; a <= b; a++)
    c *= a;
```

```
ba  test
mov  1, %a_r        !a = 1;

loop:
call  .mul
mov  %c_r, %o1
mov  %o0, %c_r
add  %a_r, 1, %a_r

test:
cmp  %a_r, %b_r    !test condition
ble,a  loop
mov  %a_r, %o0    !1st instr of loop
```
The statement following the test should be branched over if the condition is not true.

To do this, complement the branch test.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>bl</td>
<td>bge</td>
</tr>
<tr>
<td>ble</td>
<td>bg</td>
</tr>
<tr>
<td>be</td>
<td>bne</td>
</tr>
<tr>
<td>bne</td>
<td>be</td>
</tr>
<tr>
<td>bge</td>
<td>bl</td>
</tr>
<tr>
<td>bg</td>
<td>ble</td>
</tr>
</tbody>
</table>
C code

```c
    d = a;
    if ((a + b) > c) {
        a += b;
        c++;
    }
    a = c + d;
```

SPARC code

```sparc
    mov    %a_r, %d_r
    add    %a_r, %b_r, %o0
    cmp    %o0, %c_r
    ble    next
    nop
    add    %a_r, %b_r, %a_r
    inc    %c_r
    next:
    add    %c_r, %d_r, %a_r
```

Only executed if `ble` is false — which is the “true” condition for the C code.
Optimize If Then Code Segment

Move instr from before the if

add   %a_r, %b_r, %o0
cmp   %o0, %c_r
ble   next
mov   %a_r, %d_r
add   %a_r, %b_r, %a_r
inc   %c_r

next:
add   %c_r, %d_r, %a_r

If no instr to move, then use annulled branch & change target

mov   %a_r, %d_r
add   %a_r, %b_r, %o0
cmp   %o0, %c_r
ble,a next
add   %c_r, %d_r, %a_r
add   %a_r, %b_r, %a_r
inc   %c_r
add   %c_r, %d_r, %a_r

next:
If Else Example

```c
if ((a + b) >= c) {
    a += b;
    c++;
} else {
    a -= b;
    c--;
}
c += 10;
```

Now consider how to remove the 1st nop

```
add %a_r, %b_r, %o0
cmp %o0, %c_r
bl else
nop
add %a_r, %b_r, %a_r
inc %c_r
ba next

else:
    sub %a_r, %b_r, %a_r
    dec %c_r
next:
    add %c_r, 10, %c_r
```
Eliminate 1\textsuperscript{st} nop by using \texttt{bl,a} and placing 1\textsuperscript{st} statement of else in delay slot.

This way, if we take the branch, we will pick up 1\textsuperscript{st} instr for the else along the way.

If branch is not taken, then we kill the instr in the delay slot.

Now consider how to remove the next nop.
Eliminate next nop by moving an instr from end of the then section into the delay slot

Doesn’t matter if we execute the increment before we branch or “on the way” to target next

```
add    %a_r, %b_r, %o0
cmp    %o0, %c_r
bl,a   else
sub    %a_r, %b_r, %a_r
add    %a_r, %b_r, %a_r
ba     next
inc    %c_r
else:
  dec    %c_r
next:
  add    %c_r, 10, %c_r
```
To eliminate the 2\textsuperscript{nd} nop we could copy the instr following the if else into the delay slot.

We will need to copy it not only into the then section, but also in the else section.
Summary of Chapter 2

- Use of registers – with `add` & `sub`
- **Call instruction** – for `mul` & `div`
  - Have to use registers `%0` & `%1`
- Pipelining – program flow & delay slots
- **Gdb** – learn from practice
- Branching & testing
  - Control structures –– Annullled branches
  - Optimizing code