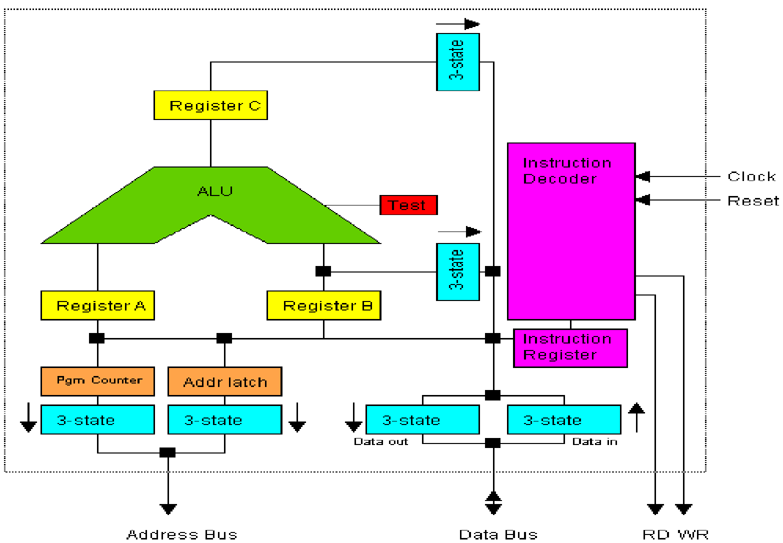
The SPARC Assembly Language

Prof. Gustavo Alonso
Computer Science Department
ETH Zürich
alonso@inf.ethz.ch
http://www.inf.ethz.ch/department/IS/iks/

SPARC Assembly Language

- Load/Store architectures
- Machine code, assembly, and high level languages
- The SPARC assembly language
  - Structure of an assembly program
  - SPARC register pool
  - Directives
  - Basic instructions
  - Instruction pipelining in the SPARC architecture
  - Branching instructions
  - mapping of control flow instructions to assembly language

Load/Store architectures



Machine code

- Processors are just a complex collection of digital gates and registers processing electronic signals that can be interpreted in binary form: 1 or 0
- The instruction decoder takes an instruction and generates (how it does not matter) the necessary signals to perform operations across the different components:
  - load data (bring some data into a register)
  - store data (move data from a register to somewhere else)
  - operate on data (add, shift, compare, etc.)
- When the operations are done on a set of registers rather than directly in memory, the architecture is called a load/store architecture
- Machine code is the binary representation of the instructions understood by the processor (by the instruction decoder in the processor)
- How the machine code is processed determines many characteristics of the processor:
  - CISC (complex instruction set computer): the machine code contains many different operations, including complex ones (e.g., matrix multiplication). The execution often involves executing "subroutines" (micro-code)
  - RISC (reduced instruction set computer): machine code contains only instructions that can be quickly executed. The number of instructions is small (reduced)

Assembly language

- Machine code is binary and, therefore, unsuitable for direct manipulation by humans
- To program at the machine code level, one uses an assembly language. The assembly language is simply a textual representation of machine code plus some syntactic rules that can be interpreted by the assembler. The assembler is the program that takes assembly code as input and produces machine code as output
- Assembly code is closely tied to the underlying processor architecture. Its basic instruction set is the machine code instruction set of the processor. Each assembler adds a dialect the programmer can use to build real programs in assembly language

```
.data
a:  .word 1
b:  .word 2

start:  .text
        set    a, %r1
        ld     [%r1], %r1
        set    b, %r2
        ld     [%r2], %r2
        add    %r1, %r2, %r3
end:    ta     0
```

```
^A^C^A^K   \234
  ^C^P\202^P`^DÂ@^E^P\204^
  P Ä\200\206X@^B\221D
  ^Oÿ^Oÿ^D^C^N^D
  ^U^F@^X^F@^D^[^D  !^D
  <%^E @-^E @4 @^H:^G@^HB
  @^HN
```


High level programming languages

- Advantages of assembly language:
  - very efficient
  - allows to manipulate the hardware almost directly (necessary for writing drivers and low level components of the operating system)
- Disadvantages of assembly language:
  - machine dependent (the language works on that processor and nowhere else)
  - programs are cumbersome and cryptic
  - very repetitive programs
- Higher level languages try to solve these problems by providing better abstractions

```
main()
{
    int a = 1;
    int b = 2;
    int c;
    c = a + b;
    printf("%d\n",c);
}
```

```
.data
a:  .word 1
b:  .word 2

start:  .text
        set    a, %r1
        ld     [%r1], %r1
        set    b, %r2
        ld     [%r2], %r2
        add    %r1, %r2, %r3
end:    ta     0
```


A toy assembly language

LOADA <u>mem</u> - Load register A from memory address <u>mem</u>
LOADB <u>mem</u> - Load register B from memory address <u>mem</u>
CONB con - Load a constant value into register B
SAVEB <u>mem</u> - Save register B to memory address <u>mem</u>
SAVEC <u>mem</u> - Save register C to memory address <u>mem</u>
ADD - Add register A and register B and store the result in register C
SUB - Subtract register A and register B and store the result in register C
MUL - Multiply register A and register B and store the result in register C
DIV - Divide register A and register B and store the result in register C
COM - Compare register A and register B and store result in register test
JUMP addr - Jump to address <u>addr</u>
JEQ addr - Jump if the previous comparison was equal (register test is 0), to address <u>addr</u>
JNEQ addr - Jump if the previous comparison was not equal (register test is 0), to address <u>addr</u>
JG addr - Jump if the comparison is Greater than (result is in register test), to address <u>addr</u>
JGE addr - Jump if Greater than or equal (result is in register test), to address <u>addr</u>
JL addr - Jump if Less than (result is in register test), to address <u>addr</u>
JLE addr - Jump if Less than or equal (result is in register test), to address <u>addr</u>
STOP - Stop execution

Structure of an assembly program

- Assembly language programs are line-oriented (the assembler translates an assembly program one line at a time). The assembler recognizes four types of lines: empty lines, label definition lines, directive lines, and instruction lines.
  - A line that only has spaces or tabs (i.e., white space) is an <u>empty line</u>. Empty lines are ignored by the assembler.
  - A <u>label definition line</u> consists of a label definition. A label definition consists of an identifier followed by a colon (":"). As in most programming languages, an identifier must start with a letter (or an underscore) and may be followed by any number of letters, underscores, and digits.
  - A <u>directive line</u> consists of an optional label definition, followed by the name of an assembler directive, followed by the arguments for the directive.
  - An <u>instruction line</u> consists of an optional label definition, followed by the name of an operation, followed by the operands.
- Comments within a line begin with the character "!". C-style type of comments (spanning several lines) are allowed using /* … */

Segments and statements

- An assembly program, is organized in three segments:
  - data segment: constants and data necessary for the program
  - text segment: the instructions of the program
  - BSS segment: (Block Storage Segment or Block Started by Symbol) space for dynamic data and non initialized global variables
- An statement:

```
label: instruction
label:
       instruction
```

- A label is a symbol or a single digit. An instruction is a pseudo-op (assembler directive), synthetic instruction, or instruction.
- Internally, machine code is binary and it is processed in binary form
- In assembly, one can work with different systems:
  - hexadecimal (0x...)
  - octal (0...)
  - decimal
- Later on we will discuss these different systems. However, keep in mind that we will use hexadecimal and octal more often than decimal
- Also keep in mind that load and store architectures are register based. Most of the instructions involve manipulating one or more registers

Assembly program (example 1)

```
        .data                       ! variables
a:      .word    0x42               ! a initialized to 0x42
b:      .word    0x43               ! b initialized to 0x43
c:      .word    0x44               ! c initialized to 0x44
d:      .word    0x45               ! d initialized to 0x45

        .text                       ! Instructions a = (a+b) - (c-d)
start:  set      a, %r1
        ld       [%r1], %r2            ! $a$ --> %r2
        set      b, %r1
        ld       [%r1], %r3            ! $b$ --> %r3
        set      c, %r1
        ld       [%r1], %r4            ! $c$ --> %r4
        set      d, %r1
        ld       [%r1], %r5            ! $d$ --> %r5

        add      %r2, %r3, %r2         ! $a+b$ --> %r2
        sub      %r4, %r5, %r3         ! $c-d$ --> %r3
        sub      %r2, %r3, %r2         ! $(a+b)-(c-d)$ --> %r2
        set      a, %r1
        st       %r2, [%r1]            ! $(a+b)-(c-d)$ --> a

end:    ta       0
```


SPARC register pool

- The assembly language we will learn is the assembly language of the SPARC V8 architecture (current V9)
- This is a RISC architecture with 32 integer registers. Each integer register holds 32-bits. The integer registers are called %r0 through %r31. In addition to the names %r0 through %r31, the integer registers have alternative names
  - <u>global registers</u> (%g0-%g7) correspond to registers %r0-%r7
  - <u>output registers</u> (%o0-%o7) correspond to registers %r8-%r15
  - <u>local registers</u> (%l0-%l7) correspond to registers %r16-%r23
  - <u>input registers</u> (%i0-%i7) correspond to registers %r24-%r31
- Global registers are used for global variables
  - %r0 is an special register that always holds the value 0 and cannot be modified
- Output registers are used for local data and arguments to/from subroutines
  - %r14 (%sp, %o6) is the stack pointer
  - %r15 is the return address of the called subroutine
- Local registers are for general use (local variables)
- Input registers are used for argument passing from subroutines
  - %r30 (%fp, %i6) is the frame pointer
  - %r31 is the subroutine return address

Directives

- The different sections of the program are marked with the following directives (also called pseudo-operations):
  - *.text* for the program code,
  - *.data* for the global writeable initialized data,
  - *.bss* for the global uninitialized data
- *.ascii* string1, ..., stringn

  Represents a sequence of bytes, initialized with the ASCII encoding of the strings, in sequence, without string terminators (.asciz adds \0)
- *.global* label

  Makes a label global
- *.byte* value1, ..., valuen

  Represents a sequence of bytes, initialized with the given data, in sequence. The values must fit within 8 bits each
- *.halfword* value1, ..., valuen

  Represents a sequence of halfwords, initialized with the given data, in sequence. The values must fit within 16 bits each
- *.word* value1, ..., valuen

  Represents a sequence of words, initialized with the given data, in sequence. The values must fit within 32 bits each
- *.include* "file_name"

  Used to add additional definitions from other files

Basic instructions (1)

Program begin

- The beginning of a program is indicated with the *.text* directive. The accepted format is

```
.text
start: first instruction
       second instruction
       …
```

Program termination

- You should terminate their execution by executing the instruction *ta*. This is a trap instruction that calls the operating system with a request encoded in register %g1

```
end:   ta   0
```

set

- The *set* operation allows to load a constant into a register

```
set   0x42, %r2
set   x, %r3
```

clear

- The *clr* (clear) operations sets a particular location in memory to 0 (this operation is synthetic, works with registers and labels denoting memory locations)

```
clr [%r3]
clr a        /* A declared in
             the .data
             segment */
```


Basic instructions (2)

load

- The load instruction brings a word from memory into a register

```
ld [%r2], %r3
```

store

- The store instruction copies the value in a register (a word) into a location in memory

```
st  %r3, [%r2]
```

In SPARC assembly language instructions, the destination is always specified as the last operand. load and store operate with different addressing modes, not just registers [%r2] means interpret the contents of %r2 as a memory address

move

- The *mov* (move) instruction copies the contents of a register or a small integer into another register (this instruction is synthetic)

```
mov    %r1, %r2
mov    1, %r2
```

add, sub

- The *add* and *sub* operators take three arguments: two operands, and a destination for the result. The operands can be either two registers or a register and a signed small constant (must fit in 13 bits)

```
add %r3, %r4, %r5
    ! %r5 = %r3 + %r4
sub %r3, 1, %r3
    ! %r3 = %r3 - 1
```


Basic instructions (3)

signed and unsigned multiplication

- The integer multiplication operations multiply two 32-bit source values and produce a 64-bit result. The most significant 32 bits of the result are stored in the Y register (%y) and the remaining 32 bits are stored in one of the integer registers. The second operand can be a small integer

```
smul   %r1, %r2, %r3
umul   %r1, 10, %r3
```

null operation

- The *nop* operation skips a cycle without doing anything

```
nop
```

signed and unsigned division

- The integer division operations divide a 32-bit value into a 64-bit value and produce a 32-bit result. The Y register provides the most significant 32 bits of the 64-bit dividend. One of the source values provides the least significant 32 bits, while the other provides the 32 bit divisor

```
sdiv   %r1, %r2, %r3
  ! %r3 = {%y,%r1}/%r2

udiv   %r1, 10, %r3
  ! %r3 = {%y,%r1}/10
```


Example 2

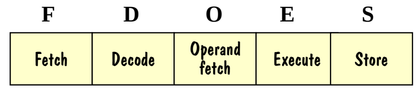
```
 .data
a:      .word   0x40
b:      .word   0x0A
c:      .word   0x04

        .text                   ! a = (a*b)/c
start:  set     a, %r1
        ld      [%r1], %r2
        set     b, %r1
        ld      [%r1], %r3
        set     c, %r1
        ld      [%r1], %r4

        smul    %r2, %r3, %r2   ! a* b --> %y, %r2
        sdiv    %r2, %r4, %r2   ! %y, %r2 / c --> %r2

        set     a, %r1
        st      %r2, [%r1]      ! %r2 --> a

end:    ta      0
```

Example 3

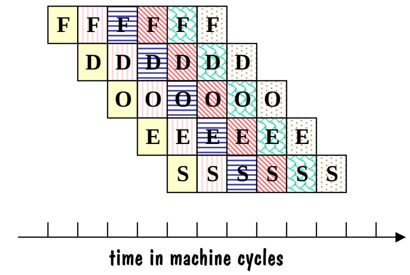
```
         .data
a2:      .word   1
a1:      .word   5
a0:      .word   7
x:       .word   9
y:       .word   0

         .text                              /*y= (x-a2)*(x-a1)/(x-a0) */
start:   set    x,  %r1
         ld     [%r1], %r1
         set    a2, %r2
         ld     [%r2], %r2
         set    a1, %r3
         ld     [%r3], %r3
         set    a0, %r4
         ld     [%r4], %r4
         set    y, %r7
         sub    %r1, %r2, %r5               ! %r5 = (x-a2)
         sub    %r1, %r3, %r6               ! %r6 = (x-a1)
         smul   %r6, %r5, %r5               ! %r5 = (x-a2)*(x-a1)
         sub    %r1, %r4, %r6               ! %r6 = (x - a0)
         sdiv   %r5, %r6, %r5               ! %r5 = %r5 / %r6
         st     %r5, [%r7]                  ! y = %r5
         end:   ta     0
```


Instruction pipelining

- To speed up the processing of instructions, most modern architectures use pipelining
- In a non-pipeline architecture, there is one single instruction executed at any given time
- In a pipelined architecture, an instruction is broken up in different parts and executed separately. At any given point in time there are several instructions being executed
- In the SPARC architecture
  - pipeline of depth 5
  - 2 instructions concurrently being executed are visible:
    - %pc
    - %npc



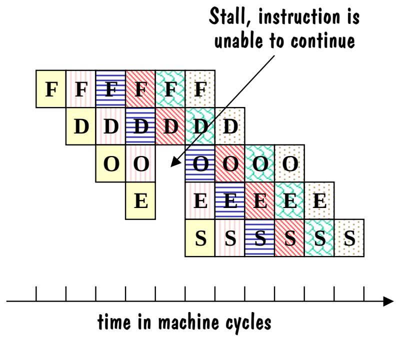
non-pipelined execution of an instruction



pipelined execution of an instruction

Problems with pipelining

- Pipelining is a great idea but not as easy as it looks due to several problems (hazards)
- *Hazard* = when an instruction's stage in the pipeline is unable to execute during the current cycle. Hazards occur in several situations:
  - (data) data dependencies: the data needed is not ready
  - (structural) shared resources: the functional unit needed is currently being used
  - (control) branches: we don't know what instruction will be executed next
- Hazards result in stalls, i.e., delays in the pipeline

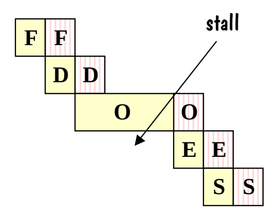


pipelined execution of an instruction

Solving hazards

- Structural hazards are solved by replicating functional units so that there are always enough of them to perform all steps of the pipeline
  - modern SPARC systems have 4 integer ALUs and 2 Floating point ALUs (and a pipeline depth of 14)
- Data hazards are typically solved by
  - forwarding: a hardware technique whereby a pipeline stage can access the results of another pipeline stage (rather than waiting for the instruction to complete)
  - code reordering: change the order of instructions to avoid data dependencies (this technique can be applied by the programmer or the compiler)
- Importance of code reordering:
- Accessing memory is much slower than accessing registers:

```
ld   [%r2], %r3
add  %r3, %r4, %r4
ld   [%r1], %r5
```




```
ld   [%r2], %r3
ld   [%r1], %r5
add  %r3, %r4, %r4
```

Branching hazards

- Branching creates its own set of problems with the pipeline:
  - when we reach the branching instruction, we don't know the result, i.e., we don't know what instruction should be executed next
  - the pipeline is automatically filled with the next instruction, which will be executed anyway (%pc, %npc) …
  - … but if we branch, the instruction execute is invalid (should not have been executed since the flow of control went somewhere else)
- The easiest solution is to include a *nop* after a branch instruction
- The branch delay slot can be used to execute instructions (more efficiency) but makes it difficult to understand the code

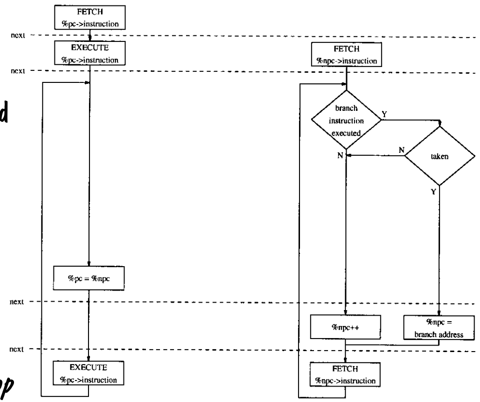


Figure 2.2: Simplified SPARC Machine Cycle

Branching in assembly language

- The SPARC architecture has a condition code (cc) register that can be used to test certain characteristics of the result of an operation. This special register has 4 bits:
  - Z (Zero): set to 1 if the result was 0
  - N (Negative): set to 1 if the result was negative
  - C (Carry): carry bit of the MSB of the result
  - V (oVerflow): indicates whether the result was too big to fit in one register
- Not all operations set these bits (special operations are needed):

  `addcc, subcc`

  `smulcc, sdivcc`
- Target is a label. The branch is taken only if condition is true, otherwise execution continues after the branch instruction

Table 3.2 Branching operations on the SPARC

Operation	Assembler syntax		Branch condition
Branch always	ba	*target*	1 (always)
Branch never	bn	*target*	0 (never)
Branch not equal	bne	*target*	not Z
Branch equal	be	*target*	Z
Branch greater	bg	*target*	not (Z or (N xor V))
Branch less or equal	ble	*target*	Z or (N xor V)
Branch greater or equal	bge	*target*	not (N xor V)
Branch less	bl	*target*	N xor V
Branch greater, unsigned	bgu	*target*	not (C or Z)
Branch less or equal, unsigned	bleu	*target*	C or Z
Branch carry clear	bcc	*target*	not C
Branch carry set	bcs	*target*	C
Branch positive	bpos	*target*	not N
Branch negative	bneg	*target*	N
Branch overflow clear	bvc	*target*	not V
Branch overflow set	bvs	*target*	V

While loop

- The while loop is a basic instruction in computer programs. It is important to implement it as efficiently as possible in assembly (the compiler does this implementation for you)
- while loop in C:

```
int temp;
int x = 0;
int y = 0x9;
int z = 0x42;

temp = y;
while( temp > 0 ) {
     x = x + z;
     temp = temp - 1;
}
```

```
        .data          / *While loop in assembly (version 1) */
x:      .word   0
y:      .word   0x9
z:      .word   0x42

        .text
start:  set     y, %r1
        ld      [%r1], %r2         ! %r2 = temp
        set     z, %r1
        ld      [%r1], %r3         ! %r3 = z
        mov     %r0, %r4           ! %r4 = x = 0

        add     %r2, 1, %r2        ! set up for decrement
        ba      test               ! test the loop condition
        nop                        ! BRANCH DELAY SLOT
top:    add     %r4, %r3, %r4      ! x + z --> x
test:   subcc   %r2, 1, %r2        ! temp - 1 --> temp
        bg      top                ! temp > 0 ?
        nop                        ! BRANCH DELAY SLOT

        set     x, %r1
        st      %r4, [%r1]         ! store x
end:    ta      0
```


While loop (optimization)

- Why does the code look so complicated?
- The problem is that the instruction after the branch is always executed. Thus, this would not work:

```
        add     %r2, 1, %r2        ! set up for decrement
top:    subcc   %r2, 1, %r2        ! temp - 1 --> temp
        bg      top                ! test the loop condition
        add     %r4, %r3, %r4      ! x + z --> x
```

- The addition takes place in the branch delay slot (good!!) but it happens one too many times (bad!!)
- To allow using the branch delay slot and yet avoid this problem, one can nullify (annul) the branch delay slot: if the branch is not taken, the instruction in the branch delay slot is "undone"

```
        .data          /* While loop in assembly (version 2) */
x:      .word   0
y:      .word   0x9
z:      .word   0x42

        .text
start:  set     y, %r1
        ld      [%r1], %r2         ! %r2 = temp
        set     z, %r1
        ld      [%r1], %r3         ! %r3 = z
        mov     %r0, %r4           ! %r4 = x = 0

        add     %r2, 1, %r2        ! set up for decrement
top:    subcc   %r2, 1, %r2        ! temp - 1 --> temp
        bg,a    top                ! temp > 0 ?
        add     %r4, %r3, %r4      ! x + z --> x

        set     x, %r1
        st      %r4, [%r1]         ! store x
end:    ta      0
```

More on while loops

- When the condition is more complex (not simply a comparison with 0), we need an extra instruction:

```
cmp   register1, register2
cmp   register2, const.

/* cmp is synthetic */

subcc register1, register2, g0
subcc register2, const, g0

/* compare is implemented by
   subtracting the values with
   subcc and then checking the
   sign. Note that %go is
   register %r0, it canot be
   modified */
```

```
main () {
  int a= 0;
  int b= 3;
  while (a<=17) {
    a= a+ b;
  }
}
```


While loops with cmp

```
        .data                   ! First attempt
a:      .word 0
b:      .word 3
        .global _main
        .text
_main:  set   a, %r1
        ld    [%r1], %r2        ! %r2 = a
        set   b, %r1
        ld    [%r1], %r3        ! %r3 = b

loop:   cmp   %r2, 17           ! a>17
        bg    store
        nop                     !1. delay slot
        add   %r2, %r3, %r2
        ba    loop
        nop                     !2. delay slot

store:  set   a, %r1
        st    %r2, [%r1]
end:    ta    0
```

```
        .data        ! Optimized version
a:      .word  8
b:      .word  3
        .global _main
        .text
_main:  set   a, %r1
        ld    [%r1], %r2
        set   b, %r1
        ld    [%r1], %r3

loop:   cmp   %r2, 17
        ble,a loop
        add   %r2, %r3, %r2

store:  set   a, %r1
        st    %r2, [%r1]
end:    ta    0
```


do while loops

- The difference between while loops and do-while loops is that, in the latter, the loop is executed at least once;

```
main() {
  int a = 0;
  int b = 3;
  do {
     a = a + b;
  } while (a<=17);
}
```

```
        .data
a:      .word   0
b:      .word   3
        .global _main
        .text
_main:  set     a, %r1
        ld      [%r1], %r2
        set     b, %r1
        ld      [%r1], %r3

        add     %r2, %r3, %r2
loop:   cmp     %r2, 17
        ble,a   loop
        add     %r2, %r3, %r2

store:  set     a, %r1
        st      %r2, [%r1]
end:    ta      0
```


for loops

```
main() {
  int a = 0;
  int b = 3;
  int i;

  for(i=1; i<=15; i++) {
    a = a + b;
  }

}
```

```
main(){
  /*equivalent program */
  int a = 0;
  int b = 3;
  int i = 1;

  while(i<=15) {
    a = a + b;
    i++;
  }
}
```

```
        .data
a:      .word   0
b:      .word   3
i:      .word   1
con:    .word   15
        .global _main
        .text
_main:  set     a, %r1
        ld      [%r1], %r2      ! %r2 = a
        set     b, %r1
        ld      [%r1], %r3      ! %r3 = b
        set     i, %r1
        ld      [%r1], %r4      ! %r4 = i
        set     con, %r1
        ld      [%r1], %r5      ! %r5 = 15
        ba      test            !branch always
loop:   cmp     %r4, %r5
        add     %r2, %r3, %r2   !delay slot
test:   ble,a   loop
        inc     %r4
store:  set     a, %r1
        st      %r2, [%r1]
end:    ta      0
```

If-then-else

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

```
                !a->r2, b->r3, c->r4
        add     %r2, %r3, %r6
        cmp     %r6, %r4            ! if
        bl,a    else
        sub     %r2, %r3, %r2       !1. instruction of else
        add     %r2, %r3, %r2       !1. instruction of then
        inc     %r4
        ba      store
        add     %r4, 10, %r4
else:   dec     %r4
        add     %r4, 10, %r4
store:  set     a, %r1
        st      %r2, [%r1]
end:    ta      0
```


switch

- A switch statement involves a complex set of branching instructions
- It is implemented using a table. In the table, we put the different instructions for each case
- Since the cases must be integers (now you know why), we use the indexing variable to calculate where into the table we must branch to continue execution
- The important points to remember are:
  - *default* if no matching case is found
  - what to do in case of *break*

```
switch (i) {
  case 1:     i += 1;
              break;
  case 2:     i += 2;
              break;
  case 15:    i += 15;
  case 3:     i += 3;
              break;
  case 4:     i += 4;
  case 6:     i += 6;
              break;
  case 5:     i += 5;
              break;
  default:    i--;
}
```


switch table

```
        .data
        .align  4
table:  .word   L1,L2,L3,L4,L5,L6,L7,L8,L9
        .word   L10,L11,L12,L13,L14,L15
        .text
        .align  4
start:  set     i, %r1
        ld      [%r1], %l0
        ld      [%r1], %o0      ! %o0 = i
        cmp     %o0, 1
        blu     default         !expression too small
        cmp     %o0, 1
        bgu     default         !too large
        nop
        set     table, %o1      !jump table
        sll     %o0, 2, %o0     !%o0 x 4 (words)
        add     %o1, %o0, %o0
                        !%o0 points to case in table
        jmpl    %o0, %g0        !transfer control
        nop
```

```
L1:       ba    end           !first case in jump table
          add   %l0,1,%l0     !i++
L2:       ba    end
          add   %l0,2,%l0     !i += 2
L15:      add   %l0,15,%l0    !i+=15; no break
L3:       ba    end
          add   %l0,3,%l0     !i += 3
L4:       add   %l0, 4, %l0   !i+=4; no break
L6:       ba    end
          add   %l0,6,%l0     !i += 6
L5:       ba    end
          add   %l0,5,%l0     !i += 5
L7:
L8:
...       ! All other labels (lack of space in foil)
L13:
L14:
default:  sub   %l0, 1, %l0   !i--
end:      ta    0
```