ARMv8 – Program Flow Control

Branches, subroutines

Program Flow Control

- By default instructions are executed in the order they appear in the program.
- Constructs such as loops, function calls, and interrupts change this
  - by causing a different instruction to be executed after the current one
- Assembly languages have less support for high-level constructs like loops, and depend more on low-level *branches* – “gotos”
- Function calls and interrupts require explicit setup

Functions and Interrupts

- *Functions* transfer control to a different section of the program, with the expectation of eventual transfer back to the location of the function call
  - *a.k.a.* “procedures”, or “routines” (or “subroutines”)
  - similar to object “methods”
- *Interrupt handlers* are specialized functions, associated with processor-hardware events or system software features
  - OS services are often provided by means of a single, shared software interrupt

Branches

- *Conditional* branch instructions provide the assembly equivalent of “if-else”
  - Also used to construct while loops, do-while loops
- *Condition Codes* (*Flags*) – are set by data manipulation instructions, and govern what conditional branches do
- *Unconditional* branches work with conditional branches in if-else constructs
- Branches are also known as “jumps”
  - In higher-level languages, implemented as "goto"

Control Flow Instructions

Immediate operand:

- **b** <label>
  - Branch to a label
- **b.**condition <label>
  - Branch to label if condition is true
- **bl** <label>
  - Branch to label and *link* – save return address in register **x30**

Register operand:

- **br** xm
  - Branch to address in register **x**m
- **blr** xm
  - Branch and link to address in **x**m
- **ret** *[*xm*]*
  - Branch to **x**m; defaults to register **x30**
  - Similar to **br** xm

Conditional Tests

Encoding	Name(s)	Meaning	Flags
0000	**eq**	is equal	z==1
0001	**ne**	not equal	z==0
0010	**hs, cs**	unsigned higher or same	c==1
0011	**lo, cc**	unsigned lower	c==0
0100	**mi**	minus (negative)	n==1
0101	**pl**	plus (positive or zero)	n==0
0110	**vs**	overflow set	v==1
0111	**vc**	overflow clear	v==0

Encoding	Name(s)	Meaning	Flags
1000	**hi**	unsigned higher	c==1 && z==0
1001	**ls**	unsigned lower or same	c==0 || z==1
1010	**ge**	signed greater or equal	n==v
1011	**lt**	signed less than	n!=v
1100	**gt**	signed greater than	z==0 && n==v
1101	**le**	signed less than or equal	z==1 || n!=v
1110	**al**	always	*any*
1111	**nv**		

Test-and-Conditional-Branch

- Test for zero, combined with branch:
  - **cbz** Xn | Wn, label
    - Compare register to zero, branch if equal
  - **cbnz** Xn | Wn, label
    - Compare register to zero, branch if not equal
  - **tbz** Xn | Wn, #uimm16, label
    - Test bit uimm16 of register, branch if equal
  - **tbnz** Xn | Wn, #uimm16, label
    - Test bit uimm16 of register, branch if not equal

If-else patterns

- A typical high-level if-else block looks like:
  - *if (**something-is-true**)*
    do-this-block
    else
    do-that-block
  - Each block can be multiple statements (or empty)
- This isn't possible in Assembly – no execution blocks as such.
- Two alternatives exist:

If-else patterns in Assembly

- Reverse the test:

```
// if  is-NOT-true:
    // jmp to .FALSE
  test something
  b.ne  .FALSE
  .TRUE:
    do-this-block
    b  .ENDIF
  .FALSE:
    do-that-block
  .ENDIF:
```

- Reverse the order of the execution blocks:

```
// if is-true:
    // jmp to .TRUE
  test something
  b.eq  .TRUE
  .FALSE:
    do-that-block
    b  .ENDIF
  .TRUE:
    do-this-block
  .ENDIF:
```

Your choice!

Example program – add 2 numbers, then if-else branch

Add the array values

If too big, return -1 (but OS thinks it's 255, 0xff)

```
//--------
// Add two numbers, do a conditional branch
//--------
    .data
array:  .dword 256, -64     // just a couple of numbers
    .set threshold, 128     // arbitrary test value

    .text
    .global _start
array_addr:  .dword array
_start:
    ldr  x0, array_addr     // x0 <-- &(array[0])
    ldr  x1, [x0]           // x1 <-- array[0]
    add  x0, x0, #8         // x0 <-- &(array[1])
    ldr  x2, [x0]           // x2 <-- array[1]
    add  x1, x1, x2         // x1 <-- array[0] + array[1]
if_else:
    mov  x2, threshold      // x2 <-- test value
    cmp  x1, x2             // set status flags
    b.ge too_big
    mov  x0, x1             // use the sum as return code
    b    end_if
too_big:
    mov  x0, -1             // error-signal return code
end_if:

    mov  x8, 0x5d           // sys_exit ftn
    svc  #0                 // do it - terminate program
//--------
```

Assemble, link, debug the program:

- Assemble for gdb:
  - `as -g -o if-else.o if-else.s`
- Link:
  - `ld -g -o if-else if-else.o`
- Debug:
  - `gdb if-else`

Debug the program:

- gdb commands:
  - Use these instructions
    - list the source file
    - set a breakpoint
    - start the program
    - disassemble the code
    - show some registers
    - display some memory
    - step through the program, showing the registers after each steps

```
list 1,32
break 10
run
disas/r
info reg x0 x1 x2 x8
x/8xg 0x4000b0
x/2xg 0x4100ec
step
info reg x0 x1 x2 x8
step
info reg x0 x1 x2 x8
step
info reg x0 x1 x2 x8
step
info reg x0 x1 x2 x8
step
info reg x0 x1 x2 x8
step
info reg x0 x1 x2 x8
```

Loop Patterns

Two basic looping patterns

- "Repeat" (do-while):
  - *do {*
    execution-block-that-affects-something
    *} while (**something-is-true**);*
  - Always executes at least once

- "While":
  - *while (**something-is-true**) {*
    execution-block-that-affects-something
    }
  - Might not execute at all

A "for" loop is a structured version of a "while" loop

- "For":
  - *for (**starting-condition**;*
    *__**something-is-true**;__*
    *__**affect-something**) {__*
    execution-block
    }

- becomes:
  - ***starting-condition***
    *while (**something-is-true**) {*
    execution-block
    affect-something
    }

Loops in Assembly

- Assembly doesn't have "while", "do-while", or "for" loop statements
- Branches are "goto" statements – build loops with these

Loops in Assembly

- "Repeat" loop:

```
.DOLOOP:
    execution-block-that-affects-something
test something
b.eq .DOLOOP
.ENDDO:
```

  - Always executes the block at least once
  - Tight loop, few "*extra*" instructions

- "While" loop:

```
.WHILELOOP:
test something
b.ne .ENDWHILE
    execution-block-that-affects-something
    b .WHILELOOP
.ENDWHILE:
```

  - May not execute the block
  - More "*extra*" instructions than a repeat loop

Do-loop / repeat-loop in C

```c
long unsigned array[] =
   { 256, -64, 19, 0x777774, 1 0x4080 } ;
#define arraylen 6

int main(int argc, char **argv)
{
   long unsigned *ptr = array;
   long unsigned sum = 0;
   int ctr = arraylen;
   do {
      long unsigned t = *ptr;
      ptr++;  // ptr += (size of long unsigned)
      sum += t;
      ctr--;  // decrement element counter
   } while (ctr != 0);

   return sum;
}
```

- simple program, illustrates a "repeat" ("do-while") loop.

Example program –

- Add array elements in a loop, return sum as return-code
- Additional features:
  - change entry point
  - data-segment addressing
  - pseudo-instruction

```
//--------
// Add some numbers, using a do-while loop
//--------
    .data
array:  .dword 256, -64, 19, 0x777774, 1, 0x4080
    .set arraylen, 6

    .text
    .global main
main:
    ldr  x1, =array      // assembler sets up address
    mov  x0, #0          // assembler will translate this
    mov  x3, arraylen    // x3 <-- loop counter
do_loop:
    ldr  x2, [x1]        // x2 <-- array[i]
    add  x1, x1, #8      // x1 <-- &(array[i])
    add  x0, x0, x2      // x0 <-- sum of elements so far

    sub  x3, x3, 1       // decrement loop counter
    cbnz x3, do_loop     // any more elements?

    mov  x8, 0x5d            // sys_exit ftn
    svc  #0                  // do it - terminate program
//--------
```

more-familiar C entry point

assembler creates .text storage loc'n

Assemble, link, debug as before

```
as  -g -o loop.o
  loop.s

ld -e main –g
  -o loop  loop.o

gdb  loop
```

- gdb commands as before, shown here

```
list 1,32
break 10
run
disas/r
info reg x0 x1 x2 x8
x/8xg 0x4000b0
x/2xg 0x4100ec
step
info reg x0 x1 x2 x8
step
info reg x0 x1 x2 x8
step
info reg x0 x1 x2 x8
step
info reg x0 x1 x2 x8
step
info reg x0 x1 x2 x8
step
info reg x0 x1 x2 x8
```

Example program again –

- use "base+index" addressing mode
  - Increment index register (loop counter) by 1 (element) instead of 8 bytes
  - Index register's value is left-shifted 3 bits (multiplied by 8) to form offset of each element in turn
  - Useful if element count is needed elsewhere

```
//--------
// Add some numbers, using a do-while loop
//--------
    .data
array:  .dword 256, -64, 19, 0x777774, 1, 0x4080
    .set arraylen, 6

    .text                  index register x2 gets left-shifted
    .global main
main:
    ldr  x1, =array      // assembler sets up address
    mov  x0, #0          // assembler will translate this
    mov  x2, #0          // assembler will translate this
                         // loop counter "i" = 0
    mov  x4, arraylen    // x4 <-- stopping condition

do_loop:
    ldr  x3, [x1, x2, lsl 3]    // x3 <-- *(array + 8*i)
    add  x2, x2, 1       // increment loop counter "i"
    add  x0, x0, x3      // x0 <-- sum of elements so far

    cmp  x2, x4          // i < 6 ???
    b.lo do_loop         // branch if more elements
// The sum serves as return code...
    mov  x8, 0x5d        // sys_exit ftn
    svc  #0              // do it - terminate program
//--------
```

Example: strlen()

```
int strlen(char *str)
{
    char *p;
    p = str;
    while (*p != '\0')
        p++ ;

    return (p - str);
}
```

- Assembly version has seven instructions
- Five instructions in the loop
- Three of the five instructions are used to implement the loop

strlen()

.asciz – null terminated strings

Comparison

Conditional branch

Unconditional branch

Loop contains 5 instructions

```
//--------
// Inline strlen(), using a "while loop"
//--------
    .data
message:  .asciz "(Your message here!)\n"

    .text
    .global main
main:
    ldr  x0, =message   // message's address into x1
strlen:
    orr  x1, x0, xzr    // a.k.a. "mov x1, x0"
strlen_test:
    ldrb w2, [x1]       // load next character
    cmp  w2, #0         // test against ASCII null
    b.eq strlen_done
    add  x1, x1, #1     // increment to next char.
    b    strlen_test
strlen_done:
    sub  x0, x1, x0     // x0 <-- str_end - str_start
                        // return code is string length
    mov x8, #0x5d       // sys_exit call
    svc #0              // syscall
//--------
```

Note: building strlen()

- Assemble as usual:
  ```
  as –g  -o strlen0.o  strlen0.s
  ```
  - "-g" flag is again for gdb support
- Program's entry point is "main", not "_start"
  - Linker assumes that the entry point is labeled "_start"
- Link command must specify desired entry point, using "-e":
  ```
  ld  -e main  -o strlen0  strlen0.o
  ```

Tightening Up the While Loop

- Get the unconditional branch out of the loop itself:
  - *b* .WHILETEST
    .WHILELOOP:
    execution-block-that-affects-something
    .WHILETEST:
    test ***something***
    b.eq .WHILELOOP
  - Only two "*extra*" instructions in the loop itself
  - Labels aren't instructions – they don't take up memory *or* time

Additional tightening step

- *Replace compare, branch instructions:*
  - *b* .WHILETEST
    .WHILELOOP:
    execution-block-that-affects-<register>
    .WHILETEST:
    cmp ***<register>*** *, <value>*
    b.eq .WHILELOOP // …or b.ne

- *…with single compare-and-branch instruction:*
  - *b* .WHILETEST
    .WHILELOOP:
    execution-block-that-affects-<register>
    .WHILETEST:
    cbz ***<register>*** *.WHILELOOP* // …or cbnz

strlen again

.asciz – null terminated string

Unconditional branch

Comparison and conditional branch, combined

Loop only has 3 instructions

```
//--------
// Inline strlen(), using a "while loop"
//--------
    .data
message: .asciz "(Your message here!)\n"

    .text
    .global main
main:
    ldr  x0, =message    // message's address into x1
strlen:
    orr  x1, x0, xzr     // a.k.a. "mov x1, x0"
    b    strlen_test
strlen_loop:
    add  x1, x1, #1      // increment to next char.
strlen_test:
    ldrb w2, [x1]        // load next character
    cbnz w2, strlen_loop         // ASCII null?
strlen_done:
    sub  x0, x1, x0      // x0 <-- str_end - str_start
                         // return code is str length
    mov x8, #0x5d        // sys_exit
    svc #0               // syscall
//--------
```