Tennessee TECH

ARMv8 Assembler Language Tutorial

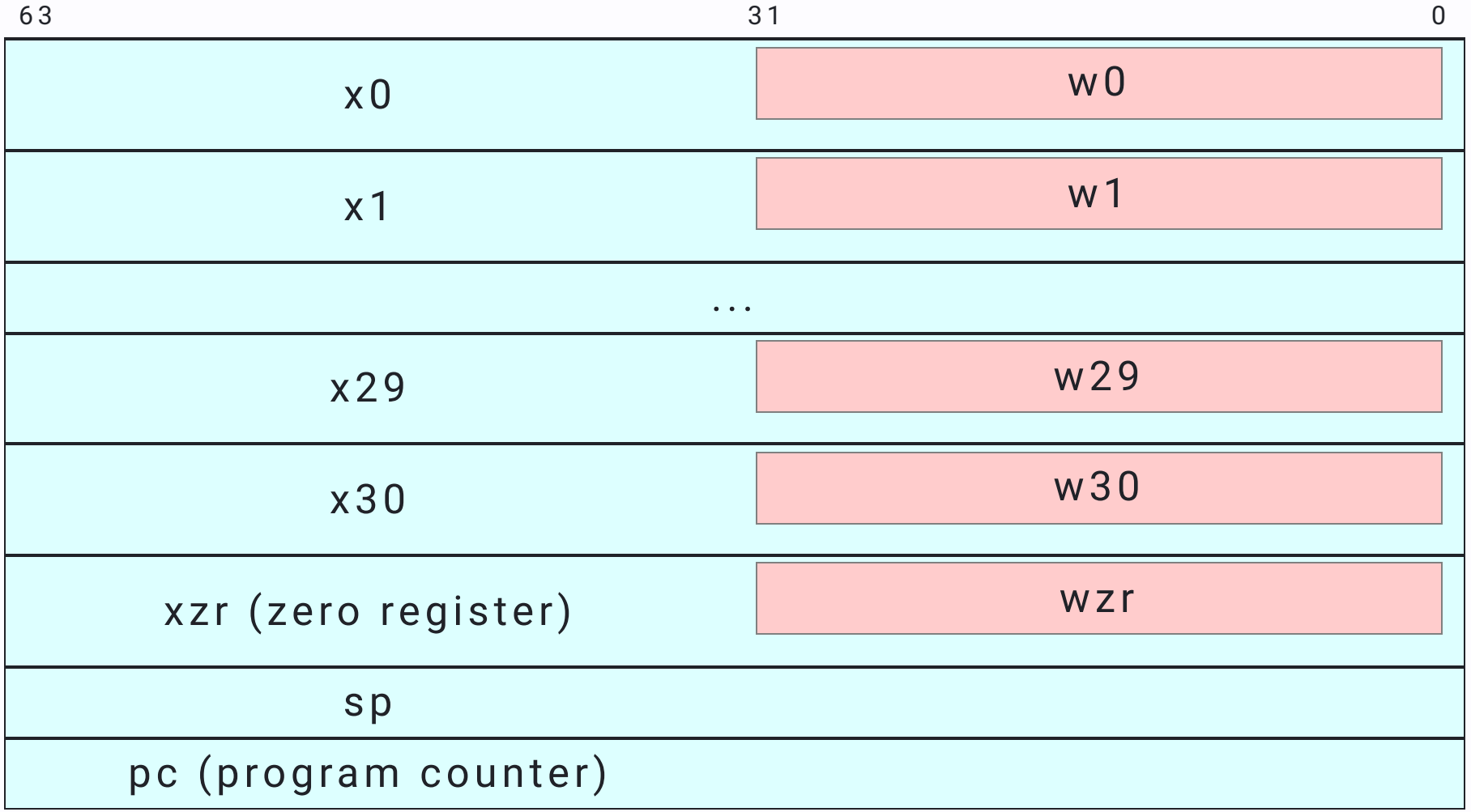
Mike Rogers

Introduction

- RISC pattern is Load/Store:
  - Load data from RAM to registers
  - Manipulate data in registers
  - Store data from registers to RAM
- On AARCH64 (also known as ARMv8) is RISC
  - Math and logic instructions can only access registers

Tennessee TECH

Registers in the AARCH64



PSTATE (various system registers)

N Z C V D A I F FIQ SS IL EL nRW SP Q GE IT J T E M

General-Purpose Registers

- Registers X0 through X30
- these are 64-bit registers
- Used to hold data for instructions
- Used to pass parameters to and from functions
- Used to hold the return address for function calls (X0)
- Some have special purposes defined in hardware (e.g. X30) or defined by software convention (e.g. X29)
- Register W0 through W30 are the 32 bit versions, and take up the 32 least significant bits of the 64 bit registers

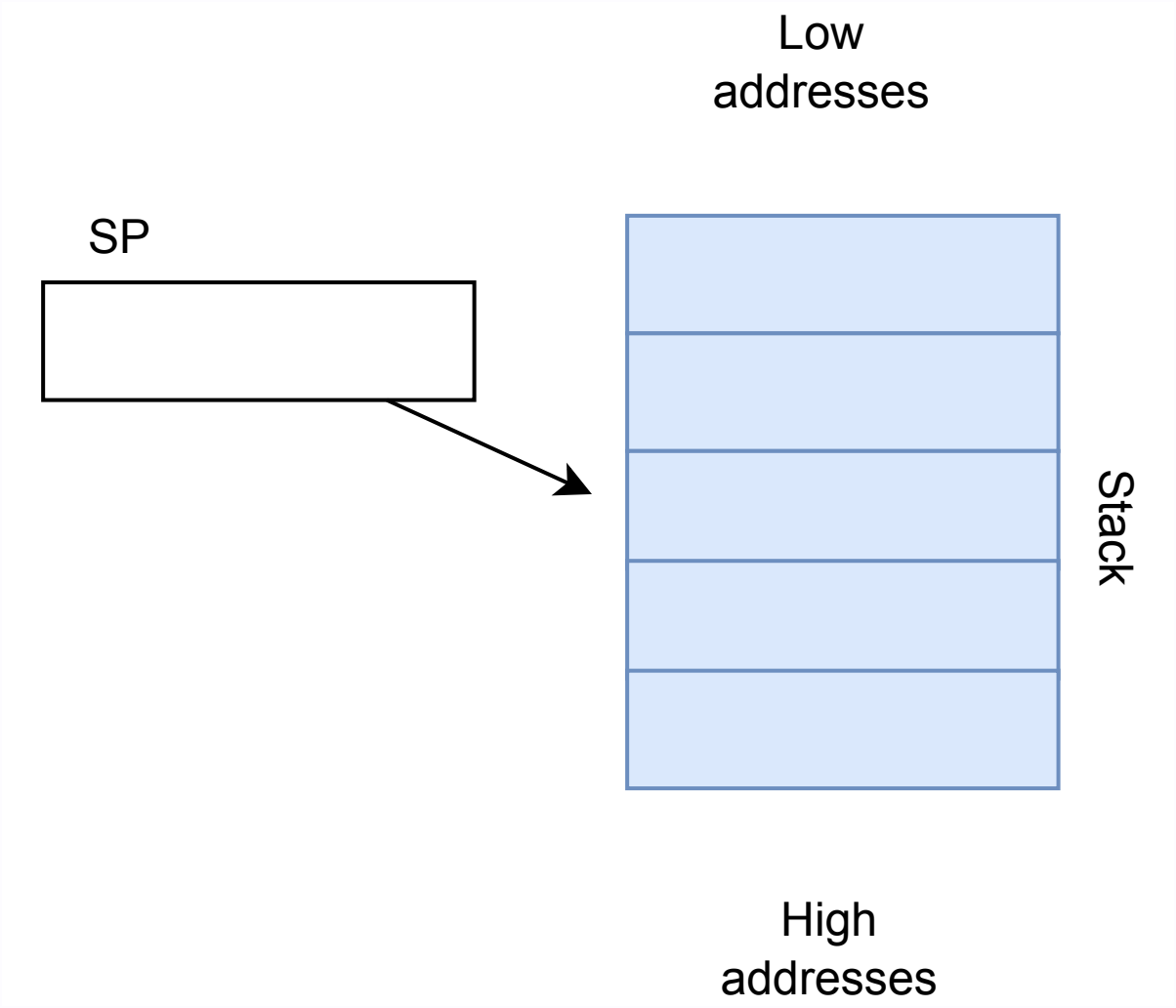
Zero Register

- The Zero register is XZR
- Reading it gives all zeros
- Any data written to it is thrown away

The Stack Pointer Register

- The stack pointer register **SP** is a special-purpose register
- It contains the *Stack Pointer*:
  - which is the address of top (low address) of current function's stack frame
  - Stack grow toward low addresses

The Stack Pointer Register (2)



PSTATE

- The processor state, or PSTATE, is not held in a single register
- And is not accessible directly
- It is a set of bits that represent...
  - condition flags set from instruction execution:
    - n (Negative), z (Zero), c (Carry), v (oVerflow)
    - Programs often use the compare (cmp) instruction to set flags
    - and then use conditional branch instructions to execute the appropriate code
      - beq, bne, blo, bhi, ble, bge, …

Instruction Format

- The general format of AARCH64 instructions is:

```
instr_name dest_reg, src_reg1, src_reg2
```

```
instr_name dest_reg, src_reg1, immediate
```

- dest_reg, src_reg1, and src_reg2 are all 64-bit x registers for 64-bit operations
- dest_reg, src_reg1, and src_reg2 are all 32-bit w registers for 32-bit operations
- Note that you cannot "mix" register sizes

64 bit Math

- C example:

```c
static long long i = 20;
static long long j = 30;
static long long k = 40;
int main() {
    k = (i + j) * 2;
    //...
}
```

- How does i,j, and k get initialized? via compiler directives (later)

- AARCH64 Assembly

```
// move address of i into x0 and
// move value of i into x0
adr     x0, i
ldr     x0, [x0]
// move address of j into x1 and
// move value of j into x1
adr     x1, j
ldr     x1, [x1]
// add i and j, put result into x2  and
// left shift by 1 = mult by 2
add     x2, x0, x1
lsl     x2, x2, 1
// move address of k into x3 and
// store result into k
adr     x3, k
str     x2, [x3]
```

Another Math Example

- C example

```
static long long x;
static long long y;
static long long z;

int main() {
        z = x - y;
        z = x * y;
        z = x / y;
        z = x & y;
        z = x | y;
        z = x ^ y;
        z = x >> y;

        return z;
}
```

Another Math Example (2)

```
        adr     x0, x       // load x and y
        ldr     x1, [x0]
        adr     x0, y
        ldr     x0, [x0]
        sub     x1, x1, x0  // z = x - y;
        adr     x0, z
        str     x1, [x0]
        adr     x0, x       // load x and y
        ldr     x1, [x0]
        adr     x0, y
        ldr     x0, [x0]
        mul     x1, x1, x0  // z = x * y;
        adr     x0, z
        str     x1, [x0]
        adr     x0, x       // load x and y
        ldr     x1, [x0]
        adr     x0, y
        ldr     x0, [x0]
        sdiv    x1, x1, x0  // z = x / y
        adr     x0, z
        str     x1, [x0]
        adr     x0, x       // load x and y
        ldr     x1, [x0]
        adr     x0, y
        ldr     x0, [x0]
        and     x1, x1, x0  // z = x & y
        adr     x0, z
        str     x1, [x0]
```

```
        adr     x0, x        // load x and y
        ldr     x1, [x0]
        adr     x0, y
        ldr     x0, [x0]
        orr     x1, x1, x0  // z = x | y;
        adr     x0, z
        str     x1, [x0]
        adr     x0, x       // load x and y
        ldr     x1, [x0]
        adr     x0, y
        ldr     x0, [x0]
        eor     x1, x1, x0  // z = x ^ y
        adr     x0, z
        str     x1, [x0]
        adr     x0, x       // // load x and y
        ldr     x0, [x0]
        adr     x1, y
        ldr     x1, [x1]
        asr     x1, x0, x1  // z = x >> y
        adr     x0, z
        str     x1, [x0]
        adr     x0, z       // return z
        ldr     x0, [x0]
        ret
```

Instuctions that save typing

```
    mov x3, x1  // same as: orr x3, xzr, x1
    neg x3, x1  // same as: sub x3, xzr, x1
```

- remember, xzr is the zero register

Signed and Unsigned Math

- C code:

```
unsigned long long x = 20;
long long y = 20;
x++;
y--;
```

- AARCH64 assembly

```
adr     x0, x
ldr     x1, [x0]
add     x1, x1, 1
str     x1, [x0]

adr     x0, y
ldr     x1, [x0]
sub     x1, x1, 1
str     x1, [x0]
```

- The same instructions
  - However, sets different condition flags in PSTATE
- Exception is division: `sdiv` vs `udiv` instructions

32-bit Arithmetic

- C code

```c
static int length = 20;
static int width = 40;
static int perim;
int main() {
    perim = (length + width) * 2;
    return perim;
}
```

- AARCH64 code

```
    adr     x0, length
    adr     x1, width
    adr     x2, perim
    ldr     w0, [x0]
    ldr     w1, [x1]
    add     w0, w0, w1
    lsl     w0, w0, 1 // mult by 2
    str     w0, [x2]
```

- Note that addresses are still 64 bit, so you must use the Xn registers (in brackets)

8- and 16-bit Arithmetic

- C Code

```
static short int x = 20;
static short int y = 30;
static short int z;
int main() {
    z = x + y;
    return z;
}
```

- AARCH64 code

```
    adr     x0, x
    adr     x1, y
    adr     x2, z
    ldrsh   w0, [x0]
    ldrsh   w1, [x1]
    add     w0, w0, w1
    strh    w0, [x2]
```

- `strh` is *store register halfword*
- Use special *load* and *store* instructions for transfer of shorter data types from and to memory
- more later

Loading and storing

```
ldr dest, [src]
str src, [dest]
```

- dest and src must be registers
- Registers in `[` brackets `]` contain memory addresses (i.e. pointer)

Loading and storing (2)

- C code

```c
static int length = 1;
static int width = 2;
static int perim = 0;
int main() {
    perim = (length + width) * 2;
    return 0;
}
```

- AARCH64 code

```
.section data
length:     .int 1
width:      .int 2
perim:      .int 0

.section    .text
.global     main
main:
    adr     x0, length
    ldr     w1, [x0]
    adr     x0, width
    ldr     w2, [x0]
    add     w1, w1, w2
    lsl     w1, w1, 1
    adr     x0, perim
    str     w1, [x0]
    mov     w0, 0
    ret
```

Sections

- Declaring sections in GAS assembly
  - `.data` is a read-write data section of initialized data
  - `.rodata` is a read-only section of initialized data
  - `.bss` is a read-write section of data initialized to zero (zeroed when loaded)
  - `.text` is execute-only, program code
- More later on the stack

Memory Labels

- Humans do not like remembering addresses for memory
- But, we like names - they are easy to remember
- *Labels* are names for locations in memory
- Example

```
.section .data
my_int:      .word 1234 // .word and .int are the same size for AARCH64
my_int2:     .int  1234
```

Global Symbols

- Declare `main` to be a globally-visible label
- Globally visible labels can be used outside of the file in which they are defined
- So, global variables and functions can be *exported*

```
.section .data
length:     .int 1
width:      .int 2
perim:      .int 0
.section .text
.global main
main:
    adr     x0, length
    ldr     w1, [x0]
    adr     x0, width
    ldr     w2, [x0]
    add     w1, w1, w2
    lsl     w1, w1, 1
    adr     x0, perim
    str     w1, [x0]
    mov     w0, 0
    ret
```

Addressing

```
.section .data
length:  .int 1
width:   .int 2
perim:   .int 0
.section .text
.global main
main:
    adr  x0, length
    ldr  w1, [x0]
    adr  x0, width
    ldr  w2, [x0]
    add  w1, w1, w2
    lsl  w1, w1, 1
    adr  x0, perim
    str  w1, [x0]
    mov  w0, 0
    ret
```

- `adr` is an instruction that loads an address of a label into a register

Loading and Storing with Addresses

```
.section .data
length: .int 1
width:  .int 2
perim:  .int 0
.section .text
.global main
main:
    adr  x0, length
    ldr  w1, [x0]
    adr  x0, width
    ldr  w2, [x0]
    add  w1, w1, w2
    lsl  w1, w1, 1
    adr  x0, perim
    str  w1, [x0]
    mov  w0, 0
    ret
```

- Once the address is loaded into x0, use the address (or *pointer*) in x0 to load the data from (or store it to) memory

Trace

x0	addr of length
w1	?
w2	?

length	1
width	2
perim	0

```
.section .data
length:     .int 1
width:      .int 2
perim:      .int 0
.section .text
.global main
main:
    adr     x0, length
    ldr     w1, [x0]
    adr     x0, width
    ldr     w2, [x0]
    add     w1, w1, w2
    lsl     w1, w1, 1
    adr     x0, perim
    str     w1, [x0]
    mov     w0, 0
    ret
```

Trace (2)

x0	addr of length
w1	1
w2	?

length	1
width	2
perim	0

```
.section .data
length:     .int 1
width:      .int 2
perim:      .int 0
.section .text
.global main
main:
    adr     x0, length
    ldr     w1, [x0]
    adr     x0, width
    ldr     w2, [x0]
    add     w1, w1, w2
    lsl     w1, w1, 1
    adr     x0, perim
    str     w1, [x0]
    mov     w0, 0
    ret
```

Trace (3)

x0	addr of width
w1	1
w2	?

length	1
width	2
perim	0

```
.section .data
length:     .int 1
width:      .int 2
perim:      .int 0
.section .text
.global main
main:
    adr     x0, length
    ldr     w1, [x0]
    adr     x0, width
    ldr     w2, [x0]
    add     w1, w1, w2
    lsl     w1, w1, 1
    adr     x0, perim
    str     w1, [x0]
    mov     w0, 0
    ret
```

Trace (4)

x0	addr of width
w1	1
w2	2

length	1
width	2
perim	0

```
.section .data
length:     .int 1
width:      .int 2
perim:      .int 0
.section .text
.global main
main:
    adr     x0, length
    ldr     w1, [x0]
    adr     x0, width
    ldr     w2, [x0]
    add     w1, w1, w2
    lsl     w1, w1, 1
    adr     x0, perim
    str     w1, [x0]
    mov     w0, 0
    ret
```

Trace (5)

x0	addr of width
w1	3
w2	2

length	1
width	2
perim	0

```
.section .data
length:     .int 1
width:      .int 2
perim:      .int 0
.section .text
.global main
main:
    adr     x0, length
    ldr     w1, [x0]
    adr     x0, width
    ldr     w2, [x0]
    add     w1, w1, w2
    lsl     w1, w1, 1
    adr     x0, perim
    str     w1, [x0]
    mov     w0, 0
    ret
```

Trace (6)

x0	addr of width
w1	6
w2	2

length	1
width	2
perim	0

```
.section .data
length:     .int 1
width:      .int 2
perim:      .int 0
.section .text
.global main
main:
    adr     x0, length
    ldr     w1, [x0]
    adr     x0, width
    ldr     w2, [x0]
    add     w1, w1, w2
    lsl     w1, w1, 1
    adr     x0, perim
    str     w1, [x0]
    mov     w0, 0
    ret
```

Trace (7)

x0	addr of perim
w1	6
w2	2

length	1
width	2
perim	0

```
.section .data
length:     .int 1
width:      .int 2
perim:      .int 0
.section .text
.global main
main:
    adr     x0, length
    ldr     w1, [x0]
    adr     x0, width
    ldr     w2, [x0]
    add     w1, w1, w2
    lsl     w1, w1, 1
    adr     x0, perim
    str     w1, [x0]
    mov     w0, 0
    ret
```

Trace (8)

x0	addr of perim
w1	6
w2	2

length	1
width	2
perim	6

```
.section .data
length:     .int 1
width:      .int 2
perim:      .int 0
.section .text
.global main
main:
    adr     x0, length
    ldr     w1, [x0]
    adr     x0, width
    ldr     w2, [x0]
    add     w1, w1, w2
    lsl     w1, w1, 1
    adr     x0, perim
    str     w1, [x0]
    mov     w0, 0
    ret
```

Cleanup

```
.section .data
length:     .int 1
width:      .int 2
perim:      .int 0
.section .text
.global main
main:
    adr     x0, length
    ldr     w1, [x0]
    adr     x0, width
    ldr     w2, [x0]
    add     w1, w1, w2
    lsl     w1, w1, 1
    adr     x0, perim
 →  str     w1, [x0]
 →  mov     w0, 0
    ret
```

- Note that the return value is put in W0
- and we return with `ret`

More addressing modes

- A64 has more addressing modes
- These modes allow you to flexibly manipulate addresses, and are particularly useful when accessing arrays
  - in loops
- However, as this is a beginning tutorial, these addressing modes will not be covered in these slides

More on defining data

- C code

```c
static char  c = 'Q';  // 8 bits
static short s = 7;    // 16 bits
static int   i = 777;  // 32 bits
static long  l = 888;  // also 32 bits
static long long ll = 999;  // 64 bits
```

- ARM code

```
.section .data
c:      .byte   'Q'    // 8 bits
s:      .short  7      // 16 bits
i:      .word   777    // 32 bits
                       // or .int or .long
l:      .int    888    // like this
ll:     .quad   999    // 64 bits
str:    .asciz  'Hello World'   // a string
```

Assembler Global Variables

```
.section .data
.global c  // c and s can be used from code in
.global s  // other files, but not i, l, or ll
c:      .byte   'Q'    // 8 bits
s:      .short  7      // 16 bits
i:      .word   777    // 32 bits
                 // or .int or .long
l:      .int    888    // like this
ll:     .quad   999    // 64 bits
str:    .asciz  'Hello World' // a string
```

- Use `.global` to export global variables define in assembly for use from other files (e.g. your C program)

What IF

- C example

```
static int i = 9;
int main () {
    if (i == 9) {
        i = 0;
    }
    return i;
}
```

- How do you turn this into assembly?

What IF (2)

- C example

```
static int i = 9;
int main () {
    if (i != 9) goto skipit;
    i = 0;
skipit:
    return i;
}
```

- First, make it easier to deal with by only having a `goto` in the body

What IF (3)

- Now, turn it into assembly

```
.section .data
i:  .int 9
.section .text
.global main
main:
    adr     x0, i
    ldr     w1, [x0]
    cmp     w1, 9          // subs wzr, w1, 9
    bne     skipit
    mov     w1, 0
    str     w1, [x0]
skipit:
    mov     w0, w1
    ret
```

IF...ELSE Example

```c
int i = 10;
int j = 20;
int smaller;
int main() {
    if (i < j)
        smaller = i;
    else
        smaller = j;
}
```

```asm
// ...
    adr     x0, i
    ldr     w1, [x0]
    adr     x0, j
    ldr     w2, [x0]
    cmp     w1, w2  // if !(i < j)
    bge     else    //    goto else
    adr     x0, smaller
    str     w1, [x0]
    b       endif
else:   // i is _not_ less than j
    adr x0, smaller // j is smaller
    str w2, [x0]
endif:
```

WHILE Example

```c
static int n = 20;
static int fact = 0;
int main() {
    fact = 1;
    while (n > 1) {
        fact *= n;
        n--;
    }
    return fact;
}
```

```asm
    // ...
    adr x0, n
    ldr w1, [x0]
    mov w2, 1
while:
    cmp w1, 1   // !(n > 1)
    ble endwhile
    mul w2, w2, w1
    sub w1, w1, #1  // the # is optional
                    //  for immediates
    b while
endwhile:
    // ...
```

Branch Modifiers

Table 3.2: AArch64 condition modifiers.

Condition code	Meaning	Condition flags	Binary encoding
EQ	Equal	$Z = 1$	0000
NE	Not Equal	$Z = 0$	0001
HI	Unsigned Higher	$(C = 1) \wedge (Z = 0)$	1000
HS	Unsigned Higher or Same	$C = 1$	0010
LS	Unsigned Lower or Same	$(C = 0) \vee (Z = 1)$	1001
LO	Unsigned Lower	$C = 0$	0011
GT	Signed Greater Than	$(Z = 0) \wedge (N = V)$	1100
GE	Signed Greater Than or Equal	$N = V$	1010
LE	Signed Less Than or Equal	$(Z = 1) \vee (N \neq V)$	1101
LT	Signed Less Than	$N \neq V$	1011
CS	Unsigned Overflow (Carry Set)	$C = 1$	0010
CC	No Unsigned Overflow (Carry Clear)	$C = 0$	0011
VS	Signed Overflow	$V = 1$	0110
VC	No Signed Overflow	$V = 0$	0111
MI	Minus, Negative	$N = 1$	0100
PL	Plus, Positive or Zero	$N = 0$	0101
AL	Always Executed	*Any*	1110
NV	Never Executed	*Any*	1111

From *ARM 64-Bit Assembly Language*, Larry D. Pyeatt with William Ughetta

Calling functions

- Use `bl` (branch and link)
  - stores return point in X30
- ret (return) instruction returns to address in X30

```
int main() {
    f();
}
void f() {
    // do something
}
```

```
    // ...
    main:
        bl      f
        mov     w0, 0
        ret
    f:
        // do something
        ret
```

Nested Function Calls

- But what about this?

```c
int main() {
    f();
    return 0;
}

void f() {
    g();
}

void g() {
    // do something
}
```

```asm
main:
    bl      f
    mov     x0, 0
    ret
f:
    bl      g
    ret
g:
    // do domething
    ret
```

- This will not work. Why?

Use the stack

- Push X30 on stack when entering a function
- Pop X30 from stack before returning from a function

```
main:
    // Save X30
    sub sp, sp, 16
    str x30, [sp]
    bl f
    // Restore X30
    ldr x30, [sp]
    add sp, sp, 16
    ret
...
```

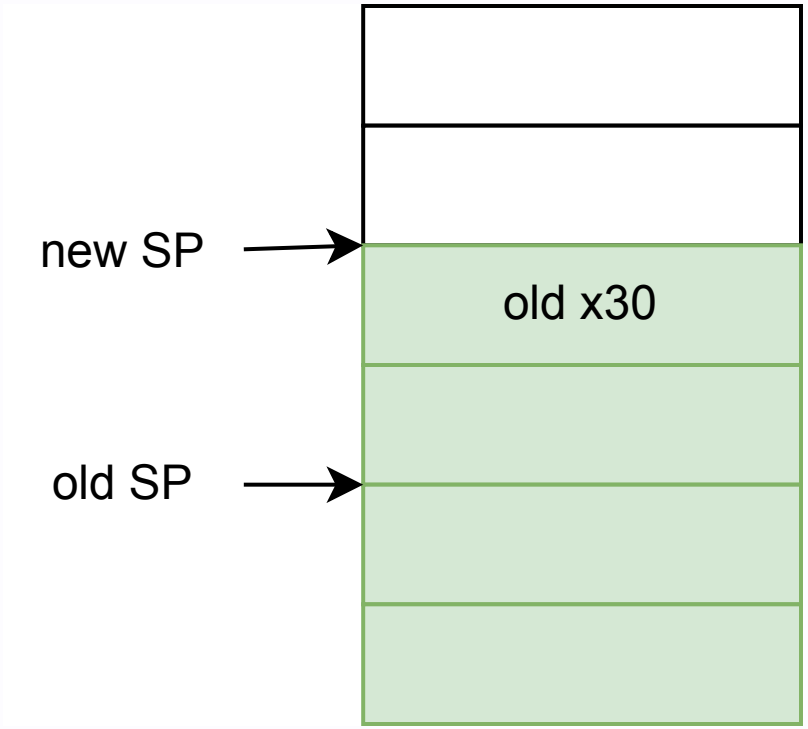
```
f:
    // Save X30
    sub sp, sp, 16
    str x30, [sp]
    bl g
    // Restore X30
    ldr x30, [sp]
    add sp, sp, 16
    ret
```

The Stack

- SP (*stack pointer*) register points to top of stack
- It can be used in ldr and str instructions
- It can be used in arithmetic instructions
- However, the address in SP *must* be multiple of 16
- That's why, even though in the previous example we only needed 8 bytes to store x30...
  - we still needed to grow the stack by 16 bytes

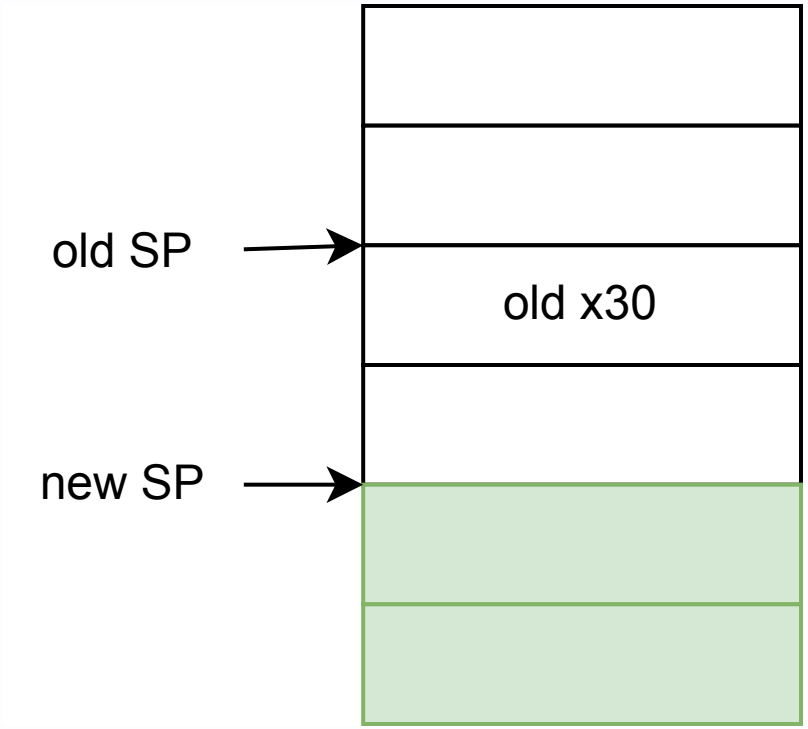
The Stack (2)

```
sub sp, sp, 16
str x30, [sp]
```



The Stack (2)

```
add sp, sp, 16
```



Passing Arguments

- Pass first 8 (integer or address) arguments in registers
  - X0..X7 and/or W0..W7
- More than 8 arguments
  - Pass them on the stack
- if arguments are structures (pass by value)
  - Pass them on the stack

Passing Arguments Example

```
// int do_something(int x, int y);
do_something:
    sub     sp, sp, 32      // Why 32? Because we will store
                            //  three values on the stack
                            //  x30 (link), x, and y.
                            //  remember, SP must be a multiple
                            //  of 16
    str     x30, [sp]       // Save x30
    str     x0, [sp, 16]    // Save x
    str     x1, [sp, 8]     // Save y

    // now we can use x0 and x1 as scratch registers
    //   or to pass parameters to functions we call.
    //   We load x and y when we need them.
    // x30 is also saved, which make calling other functions
    //   safe.

    //...

    ldr x30, [sp]
    // Restore x30
    add sp, sp, 32
    ret
```

Tennessee TECH

What Registers Do I Save?

Next, the PCS defines which registers can be corrupted, and which registers cannot be corrupted. If a register can be corrupted, then the called function can overwrite without needing to restore, as this table of PCS register rules shows:

X0-X7	X8-X15	X16-X23	X24-X30
Parameter and Result Registers (`X0-X7`)	`XR` (`X8`)	`IP0` (`X16`)	Callee-saved Registers (`X24-X28`)
-	Corruptible Registers (`X9-X15`)	`IP1` (`X17`)	`FP` (`X29`)
-	-	`PR` (`X18`)	`LR` (`X30`)
-	-	Callee-saved Registers (`X19-X23`)	-

For example, the function `foo()` can use registers `X0` to `X15` without needing to preserve their values. However, if `foo()` wants to use `X19` to `X28` it must save them to stack first, and then restore from the stack before returning.

Some registers have special significance in the PCS:

- `XR` - This is an indirect result register. If `foo()` returned a struct, then the memory for struct would be allocated by the caller, `main()` in the earlier example. `XR` is a pointer to the memory allocated by the caller for returning the struct.
- `IP0` and `IP1` - These registers are intra-procedure-call corruptible registers. These registers can be corrupted between the time that the function is called and the time that it arrives at the first instruction in the function. These registers are used by linkers to insert veneers between the caller and callee. Veneers are small pieces of code. The most common example is for branch range extension. The branch instruction in A64 has a limited range. If the target is beyond that range, then the linker needs to generate a veneer to extend the range of the branch.
- `FP` - Frame pointer.
- `LR` - `X30` is the link register (`LR`) for function calls.

https://developer.arm.com/documentation/102374/0102/Procedure-Call-Standard