

Course Objectives

- Learn how a typical modern computer is structured
 - I.e. its *architecture*
 - Will concentrate on CPU architecture
- Learn how a computer operates as it executes instructions
 - I.e. the *fetch-execute* cycle
- Learn how data and instructions are represented internally in a computer
 - Signed and unsigned integers
 - Characters and strings
 - Floating-point numbers
 - Machine instructions
- Learn how to write programs in an assembly language
- Often needed for creating:
 - Embedded systems
 - OS kernels
 - Device drivers
 - Code generator part of a compiler
 - Applications (rare today)
- Useful for:
 - Nothing
 - Understanding a computer's architecture
 - Understanding the details of Operating Systems
 - Writing more efficient high-level programs
- Understand the connection between high-level languages and machine operation
- Learn another high-level language (C)
- Learn how to mix assembly code and C code

Computer Architectures and Assembly Language Programming

High-Level Architecture

- A basic computer system consists of:
 - CPU
 - System clock
 - Primary memory
 - Also called Random Access Memory (RAM).

- Secondary memory
 - Usually on a HDD/SSD
- Peripheral input and output
 - Keyboard, monitor
- Bus

CPU

- Is the “brains” of any computer system
 - Executes instructions
 - Controls the transfer of data across the bus
- Is usually contained on a single microprocessor chip
- Consists of 3 main parts:
 - Control Unit (CU)
 - Arithmetic Logic Unit (ALU)
 - Registers - will vary from architecture to architecture. May be dozens to hundreds

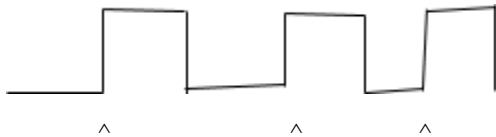


CLICK TO EXPAND <3 ;)

- **The CU** directs the execution of instructions
 - Loads the operation code (*opcode*) from primary memory into the Instruction Register (IR)
 - Decodes the opcode to identify the operation
 - If necessary, transfers data between primary memory and registers
 - If necessary, directs the ALU to operate on data in register
- **The ALU** performs arithmetic and logical operations on data stored in registers
 - Eg: add numbers in 2 source registers, and store the result in a destination register

- Eg: Do a bitwise AND using data in 2 registers
- **Registers** are binary storage units within the CPU
 - May contain:
 - Data
 - addresses
 - Instructions
 - Status information
 - Eg: *General-purpose registers* are used by a programmer to temporarily hold data and addresses
 - Eg: The *Program Counter* (PC) contains the address in memory of the currently executing instruction
 - Is incremented to execute the next instruction
 - Eg: The *Status Register* (SR) contains information (flags) about the result of a previous instruction
 - Eg: overflow, or carry

System Clock

- Generates a clock signal to synchronize the CPU and other clocked devices
 - Is a square wave at a particular frequency (0-3.3 to 5 volts)
- 
- Devices coordinate on the rising or falling edges
 - Example clock rates:
 - 3.2 GHz
 - Raspberry Pi (original): 700 MHz

Primary memory

- Often called Random Access Memory (RAM)
 - Any byte in memory can be accessed directly if you know it's address
- Can be written to and read from
- Is **volatile**
 - Data disappears when powered off
- Is used to store program instructions and program data (variables)
- Consists of a sequence of addressable memory locations
 - Each location is typically one byte long

-

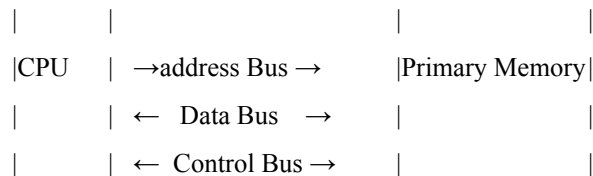
0
1
2
.
.
.
N

- Example sizes:

- iMac (2016): 8gb
- Raspberry Pi (original): 256 MB
- In a *von Neumann architecture*, RAM contains both data and programs (instructions)
- In contrast, a *Harvard architecture* uses separate memories for data and programs

Bus

- Is a set of parallel data/signal lines
- Is used to transfer information between computer components
- Often subdivided into address, data and control busses



Bus (Cont.)

1. address Bus:

- Specifies a memory location in RAM
 - Or sometimes a memory-mapped I/O device
- Common sizes 32 and 64bit

2. Data Bus:

- Used for bidirectional data transfer
- Common sizes 32 and 64bit

3. Control Bus:

- Used to control or monitor any devices connected to the bus
 - e.g Read/Write signal for Ram

An expansion bus may be connected to the computer's local bus

- Makes it easy to connect additional I/O devices to the computer
- Example bus standards: USB, SCSI, PCIe

Secondary Memory (HDDs)

Is used to hold a computer's file system

- Store files containing programs or data

Is **non-volatile** read/write memory

- Its contents persist through a power cycle

Usually embodied on HDD

- But solid state drives (SSDs) are becoming more common

Peripheral I/O Devices

Allow communication between computer and the external environment

Example Input Devices:

- Keyboard
- Pointing devices: Mouses, Tablets, etc.
- Microphone

Example Output Devices:

Well meme'd, my men

- Monitor
- Printer
- Speakers

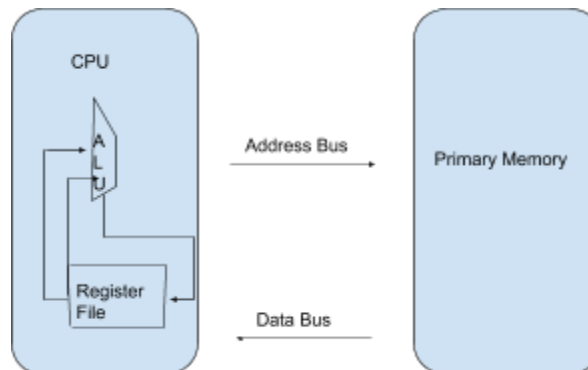
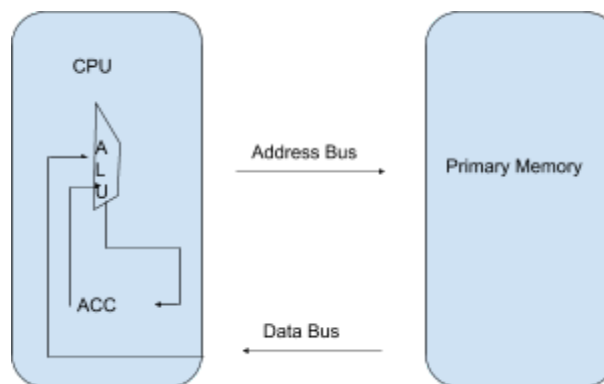
Examples of I/O Devices:

- HDD
- Modem

Basic CPU Architecture:

→ **Accumulator**

→ **Load/Store Machines:**



Basic CPU Architectures (Cont)

- SPARC is an example of LOAD and STORE machine

Only *load* and *store* instructions in the RAM

Other instructions operate on specified registers in the register file, not RAM

- Registers are more quickly accessed than Ram, so this is fast

Typical program sequence:

1. Load registers from memory
2. Execute instruction using two source registers, putting the result into a destination folder
3. Store the result back into memory

RISC and CISC Architecture:

- RISC: Reduced Instruction Set Computer
 - Uses only simple instructions that can be executed in one machine cycle
 - Which enables faster clock rates, thus faster overall execution, but makes programs larger, more complex
 - e.g Original SPARC had no multiply instructions (mult. is done using repeated add-shift operation)
- Machine instructions are always the same size
 - Makes decoding simpler and faster
 - e.g ARMv8 instructions are always 32bits wide
- CISC: Complex Instruction Set Computer
 - May have instructions that take many cycles to execute
 - Are provided for programmer convenience
 - But slows down overall execution speed
 - Eg: Intel Core 2
 - add: 1 cycle
 - mul: 5 cycles
 - div: 40 cycles
 - Machine instructions vary in length, and may be followed by “immediate” data
 - Makes coding difficult and slow
 - Eg: Intel x86
 - Can be as short as 1 byte long (eg: INC)
 - Some can be as long as 15 bytes

Instruction Cycle

- Also called the *fetch-execute* or *fetch-decode-execute* cycle
- The CPU executes each instruction in a series of small steps;
 - a. Fetch the instruction from memory into the *instruction register* (IR)
 - i. The *Program Counter* register (PC) contains its address
 - b. Increment PC to point to the next instruction
 - c. Decode the instruction
 - d. If the instruction uses an operand in RAM, calculate its address. Repeat if necessary
 - e. Fetch the operand. Repeat if necessary
 - f. Execute the instruction
 - g. If the instruction produces a result that is stored in RAM, calculate its address. Repeat if necessary
 - h. Store the result. Repeat if necessary

Assembly Language Programs

- Consists of a series of *statements*, each corresponding to a machine instruction
 - ARMv8 example:
`add x20, x20, x21 // first x20 is overwritten by adding the second x20 and x21`
 - Corresponds to:
`1000 1011 0001 0101 0000 0010 1001 0100`
 - Or in hexadecimal:
`0x8b150294`
- Each statement consists of an *opcode*, and a variable number of *operands*
- eg: `add x20, x20, x21`
opcode operands
- Instructions are stored sequentially in memory
- Each instruction (statement) thus has a unique address
- Optionally, a *label* can prefix any statement
- Form: `label: statement:`
- Eg: `start: add x20, x20, x21`
- Is a symbol whose value is the address of the machine instruction
 - May be used as a target for a branch instruction
- *Pseudo-ops (assembler directives)* do not generate machine instructions, but give the assembler extra information
- Form: `.pseudo-op`
- Eg: `.global start`
- Comments may be appended to the end of a statement
- In ARMv8, after a `//` delimiter
`start: add x20, x20, x21 // add comments like BOGON KEEPS TURNING MY NOTES INTO INVISIBLE INK`
- The labels, opcodes, operands, and comments should be formatted into columns
- | | | | |
|---------|---------|----------|------------|
| Labels: | opcodes | operands | //comments |
|---------|---------|----------|------------|
- Use an editor like *emacs* to automate this

Assemblers

- *Translate* assembly source code into machine code
- In this course, we will use the GNU *as* assembler
 - Part of the GNU *gcc* compiler suite
- To assemble ARMv8 source code use:

```
gcc myprog.s -o myprog
```

 - Gcc invokes the assembler *as* then links the code, producing an executable called “myprog”
 - Assumes files ending in *.s* contain assembly source code

Macro Preprocessor

- Many assemblers support macros
 - Allows you to define a piece of text with a macro name
 - Optionally, parameters can be specified
 - This text will be substituted inline wherever invoked
 - Called *macro expansion*
 - Provided as a convenience to help make your code more readable
- Unfortunately, *gcc* (actually *as*) has limited support for macros
 - We use *m4* instead, before invoking *gcc*
 - Is a standard UNIX (Linux) command
- Eg:

```
define(coef, 23) ← defines 2 macros
define(z_r, x18)
...
add x19, z_r, coef ← macros invoked
...
Expanded to:
add x19, x18, 23
...

```
- General procedure:
 - Put your source code containing macros into a file ending in *.asm*
 - Invoke *m4*, redirecting output to a file ending in *.s*
 - Eg: `m4 myprog.asm > myprog.s`
 - Run *gcc* as usual on the output file

Reading and Exercises

ARMv8-A Architecture

Introduction

- This course uses the Applied Micro X-Gene X-C1 servers
 - It's CPU is the APM883208-X1 X-Gene Multi-Core 64 bit processor
 - Is an implementation of the ARMv8-A specification, licensed from ARM Holdings, PLC
 - ARM: Advanced RISC Machine (Originally Acorn Risc Machine)
 - The installed operating system (OS) is Linux Fedora 24 (tip)(tip)(tip)
 - Includes up-to-date versions of *gcc*, *as*, *gdb*, and *m4*
 - The 3 servers can be accessed remotely using ssh (secure shell) with these addresses:
 - Csa1.cpsc.ucalgary.ca
 - Csa2.cpsc.ucalgary.ca
 - Csa3.cpsc.ucalgary.ca
 - Or through the load balancer: arm.cpsc.ucalgary.ca
 - Use your CPSC credentials to log in
- The ARMv8-A architecture is:
 - RISC
 - Load/store machine
 - Register file contains 31 64-bit-wide registers
 - Most instructions manipulate 64-bit or 32-bit data stored in these registers
 - Uses a von Neumann architecture for RAM
- The ARMv8-A architecture has two *execution states*:
 1. AArch64
 - Uses the A64 instruction set and 64-bit registers
 - Used exclusively in this course
 2. AArch32
 - Uses the A32 or T32 instruction sets
 - Provided for compatibility with the older ARM and THUMB instruction sets using 32-bit registers
 - The ARMv8-A architecture has 4 *exception levels*:
 - **EI0**: for normal user applications with limited privileges
 - Restricted access to a limited set of instructions and registers, and to certain parts of memory
 - Most programs work at this level
 - **EI1**: for the OS kernel

- Privileged access to instructions, registers, and memory
- Accessed indirectly by user programs using *system calls*
- **EL2:** for a *Hypervisor*
 - Support *virtualization*, where the computer hosts multiple guest operating systems, each on its own *virtual machine*
- **EL3:** Low-level firmware
 - Includes the *Secure Monitor*

EL0 [Application] [Application] [Application] [Application]

EL1 [OS Kernel] [OS Kernel]

EL2 [Hypervisor]

EL3 [Secure Monitor]

ARMv8 Registers

- In AArch64, has 31 64-bit-wide general-purpose registers
 - Numbered from 0 to 30
 - When using all 64 bits, use “x” or “X” before the number (stands for extended)
 - Eg: x0, x1, x30
 - When using only the low-order 32 bits of the register, use “w” or “W” (word)
 - Eg: w2, w29
- Many of these registers have special uses:
 - x0 - x7: used to pass arguments into a procedure, and return results
 - x8: *indirect result location* register
 - x9 - x15: *temporary* registers
 - x16, x17: *intra-procedure-call* temporary registers (IP0, IP1)
 - x18: *platform register*
 - x29: *frame pointer* (FP) register
 - x30: *procedure link register* (LR)
- For now, **use registers x19-x28** for most of your work
 - Are *callee-saved* registers
 - Value is preserved by any function you call

- There are several special-purpose registers:

- Stack Pointer:
 - SP: 64 bit register used in A64 code
 - WSP: 32 bit register used in A32 code
 - Used to point to the top of the run-time stack
- Zero Register:
 - XZR: 64 bits wide
 - WZR: 32 bits wide
 - Gives 0 value when read from
 - Discards value when written to
- Program Counter
 - PC: 64 bits wide
 - Holds the address of the currently executing instruction
 - Cannot be accessed directly as a named register
 - Is changed *indirectly* by branch and other instructions
 - Is used *implicitly* by PC-relative load/stores
 - Can be accessed in *gdb* with \$pc
- There are 32 128-bit-wide *floating-point* registers
 - Discussed in detail later
- Has numerous *system registers*
 - Most are accessible only in EL1
 - Used in OS kernel code

A64 Assembly Language

- Consists of statements with one opcode and 0 to 4 operands
 - The allowed operands depend on the particular instruction
 - In general:
 - The first operand is a *destination register*
 - The others are *source registers*
 - Eg: `add x19, x20, x21`
- An *immediate value* (a constant) may be used as the final source operand for some instructions
 - Eg: `add x19, x20, 42`
 - A # symbol can prefix the immediate, but is optional when using gcc
 - Eg: `add x19, x20, #42`
 - The allowable range of constants depends on the particular instruction
 - Depends on the number of available bits within the machine instruction
- immediates are assumed to be decimal numbers unless prefix as follows:

- Hexadecimal: 0x
 - Eg: 0x6f
- Octal: 0
 - Eg: 0777
- Binary: 0b
 - Eg: 0b101
- Some instructions are *aliases* for other instructions
 - Eg: `mov x29, sp`
 - Is an alias for
 - `add x29, sp, 0`
 - Are provided for readability and programmer convenience
- Some commonly-used instructions are:
 - move immediate (32-bit)
 - Form: `mov Wd, #imm32`
 - *Wd*: destination register
 - *#imm32*: integer in range -2^{31} to $+2^{32}-1$
 - Eg: `mov w20, -237`
 - move immediate (64-bit)
 - Form: `mov Xd, #imm64`
 - *Xd*: destination register
 - *#imm64*: integer in range -2^{63} to $+2^{64}-1$
 - Eg: `mov 21, 0xFFFFE`
 - move register (32-bit)
 - Form `mov Wd, Wm`
 - *Wd*: destination register
 - *Wm*: source register
 - Alias for: `orr Wd, wzr, Wm`
 - Eg: `mov w21, w28`
 - move register (64-bit)
 - Similar form to above
 - Eg: `mov x22, x20`
- A function can be called using the *Branch and Link* instruction (bl)
 - Can be a library function or your own function
 - Form: `bl label`
 - Eg: `bl printf`
 - Arguments are put into x0 - x7 before the call

- Return value is in x0

Basic Program structure

- The *main* routine of a program can be structured as follows:

```
.global main
main: stp x29, x30, [sp, -16]! |
    mov x29, sp                | ← Saves State
                                |
    .
    .
    .
    ldp x29, x30, [sp], 16      | ← Restores State
    ret
```

- `.global main`
 - Makes the label “main” visible to the linker
 - The `main()` routine is where the execution always starts
- `...[sp, -16]!`
 - Allocates 16 bytes in stack memory (in RAM)
 - Does so by *pre-incrementing* the SP register by -16
- `stp x29, x30, ...`
 - **Stores** the contents of the **Pair** of registers to the stack
 - x29: frame pointer (FP)
 - x30: link register (LR)
 - SP points to the location in RAM where we write to
 - Saves the state of the registers used by calling code
- `mov x29, sp`
 - Updates FP to the current SP
 - FP may be used as a base address in the routine
- `ldp x29, x30, ...`
 - **Loads** the **pair** of registers from RAM
 - SP points to the location in RAM where we read from
 - Restores the state of the FP and LR registers
- `...[sp], 16`

- Deallocates 16 bytes of stack memory
- Does so by *post-incrementing* SP by +16
- **I miss Boggy's sweet sensual smile**
- `ret`
- Stands for Return
 - Returns control to calling code (in OS)
 - Uses the address in LR

Basic Arithmetic Instructions

- addition
 - Uses 1 destination and 2 source operands
 - Register (64-bit and 32-bit):
Eg: `add x19, x29, x21 // x19 = x29 + x21`
`add w19, w19, w20 // w19 = w19 + w20`
 - Immediate (64-bit and 32-bit):
Eg: `add x20, x20, 1 // x20 = x20 + 1`
`add w20, w20, 1 // w20 = w20 + 1`
- Subtraction
 - Uses 1 destination and 2 operations
 - Register (64-bit and 32-bit)
Eg: `sub x0, x1, x2 // x0 = x1 - x2`
`sub w3, w6, w7 // w3 = w6 - w7`
 - Immediate (64-bit and 32-bit):
Eg: `sub x20, x20, 1 // x20 = x20 - 1`
`sub w27, w19, 4 // w27 = w19 - 4`
- Multiplication
 - Uses 1 destination and 2 or 3 source registers
 - NO IMMEDIATES ALLOWED
 - Form (32-bit) `mul, Wd, Wn, Wm`
 - Calculates: $Wd = Wn \times Wm$
 - Alias for: `madd Wd, Wn, Wm, wzr`
 - Eg: `mul w0, w1, w2`
 - The 64-bit form is similar:
 - Eg: `mul x19, x20, x20 // square number`

Multiply-add:

Well meme'd, my men

- Form(32 bit): `madd Wd, Wn, Wm, Wa`
 - $Wd = Wa + (Wn \times Wm)$
 - e.g `madd w20, w21, w22, w23`

- Form(64 bit): `madd x20, x0, x1, x20`

Multiply-Subtract

- Form(32 bit): `msub Wd, Wn, Wm, Wa`
 - $Wd = Wa - (Wn \times Wm)$
- Form(64 bit): similar

Multiply-Negate

- Form(32 bit): `mneg Wd, Wn, Wm`
 - $Wd = -(Wn \times Wm)$

Other variants possible (See ARM)

Division:

- Uses 1 destination and 2 source registers
- No immediates allowed
- Signed form(32 bit): `sdiv Wd, Wn, Wm`
 - Operands are signed integers
 - Calculates: $Wd = Wn / Wm$
 - E.g `sdiv x21, x22, x23`
- The *udiv* variants use unsigned integer operands
 - e.g: `udiv x21, x22, x23`
- These instructions do *integer division*
 - The calculated quotient is an integer
 - e.g $14/3$ is 4 with a remainder 2
 - The remainder (or modulus) can be calculated using **num - (quotient * denominator)**
- Dividing by 0 doesn't generate an error (a trap)
 - Writes 0 to the destination register

PRINTING TO STANDARD OUTPUT

- Is done by calling `printf()`
 - Is a standard function in the C library

Well meme'd, my men

- Invoked with 1 or more arguments
- Example C code:

```
...
int x = 1,000,000;
printf("Small Loan = %d dollars \n", x);
                        ^ int placeholder
```

Equivalent assembly code:

```
Fmt:      .string "Meaning of life = %d\n"    // creates the format string
          .balign 4                          // ensures instructions are
                                              // properly aligned

          .global main

Main:     . . .
          Adrp   x0, fmt                      -|} Arg 1: address of string
          Add    x0, x0, :lo12:fmt            -|
          Mov    w1, 42                      //Arg 2: int value
          Bl     printf                      //Function call
          . . .
```

// Hello World in Assembly

```
// Define string for call to printf()
```

```
hello: .string "Hello, world!\n"
```

```
// Define the main function of our program
```

```
.balign 4                                // Instructions must be word aligned
```

```
.global main                            // Make "main" visible to the OS
```

```
main: stp   x29, x30, [sp, -16]!         // Save FP and LR to stack, allocating 16 bytes, pre-increment
      SP
```

```
      mov   x29, sp                     // Update FP to current SP
```

```
    adrp x0, hello                // set 1st argument to pass to printf(format, var1, var2...)
    (high-order bits)
```

```
    add x0, x0, :lo12:hello      // set 1st argument to pass to printf(format, var1, var2...)
    (lower 12 bits)
```

```
    bl   printf                  // Call the printf() function
```

```
    // Set up return value of zero from from main()
```

```
    mov  w0, 0
```

```
    ldp  x29, x30, [sp], 16      // Restore FP and LR from stack, post-increment SP
```

```
    ret                                // Return to caller
```

Branch Instructions and Condition Codes:

A **branch** instruction transfers control to another part of a program

- like a *goto* in the C language
- PC register is not incremented as usual, but is set to the computed address of an instruction
 - Corresponds to the value of its label
- An *unconditional branch* is always taken
 - Form: `b label`
 - eg. `b top`
- *Condition Flags* may be used to store information about the result of an instruction
 - Are single-bit in the CPU
 - Record *process state*(PSTATE) information
 - 0 means **false**, 1 means **true**
 - There are **4** flags:
 - **Z**: true if the result is 0
 - **N**: true if the result is negative
 - **V**: true if the result overflows
 - **C**: true if result generates a carry out
 - Condition flags are set by instructions that end in “s” (short for set flags)
 - Eg: `subs`, `adds`
 - *subs* may be used to compute two registers
 - Eg: `subs x0, x1, x2`
 - But *cmp* is more intuitive:
 - Form(64-bit): `cmp Xn, Xm`
 - Is an alias for: `subs xzr, Xn, Xm`
 - Eg: `cmp x1, x2`
- *Conditional branch* instructions use the conditional flags to make a decision
 - If particular flags test true, then the branch is taken
 - I.e one “jumps” to the instruction at the specified level
 - Otherwise, control “drops through” to the following instruction
 - Eg: `b.eq top`

- Branches if Z is true
- Form: `b.cc`
 - Where *cc* is a *condition code*
- The condition codes for *signed* integers are:

Name	Meaning	C Equivalent	Flags
eq	Equal	<code>==</code>	<code>Z == 1</code>
ne	Not equal	<code>!=</code>	<code>Z == 0</code>
gt	Greater than	<code>></code>	<code>Z == 0 && N == V</code>
ge	Greater than or equal	<code>>=</code>	<code>N == V</code>
lt	Less than	<code><</code>	<code>N != V</code>
le	Less than or equal	<code><=</code>	<code>!(Z==0 && N == V)</code>

Loops

- Are formed by branching from the bottom of the loop to the top
- The *do* loop is the *post-test* loop
 - The loop body is will be executed at least once
 - Eg: C code


```
long int x;
```

```
x = 1;
do {
    //loop body
    X++;
} while (x <= 10);
```

- Equivalent assembly code:

```
define(x_r, x19)
```

```
mov x_r, 1
top: statements forming
     loop body
```

```
add x_r, x_r, 1
cmp x_r, 10
```

```
b.le top
```

- The *while* loop is the *pre-test* loop
 - Possible the loop body will not be executed
 - Eg: C Code

```
long int x;

x = 0
while (x < 10) {
    // loop body
    x++;
}
```

- Assembly

```
define(x_r, x19)

    mov    x_r, 0
test: cmp    x_r, 10
      b.ge done

      Statements forming
      Body of loop

      add    x_r, x_r, 1
      b      test
done:      statements following loop
```

- Note: We branch over the loop body if $x \geq 10$
 - The logic operation is complemented
- This code can be optimized by moving the test to the end of the loop
 - Loop reduced by one instruction
 - Branch does not use complemented logic
 - I.e matches the original C code
 - Must branch to test the first time through
 - But is still a pre-test loop!

```
define(x_r, x19)

    mov    x_r, 0
    b      test
top:  statements forming
      loop of body
```

```
        add    x_r, x_r, 1
test:   cmp    x_r, 10
        b.lt   top
```

Statement following loop

- A *for* loop can be formed by first converting it to the equivalent *while* loop
- Eg: C code

```
for (i = 10; i < 20, i++)
    x += i;
```

Is the same as:

```
i = 10;
while (i < 20) {
    x += i;
    i++;
}
```

- Assembly

```
define(i_r, x19)
define(x_r, x20)

        mov    i_r, 10      // init
        b      test
top:     add    x_r, x_r, i_r // loop body

        add    i_r, i_r, 1
test:    cmp    i_r, 20
        b.lt   top
```

The if Construct

- Is formed by branching over the statement body if the condition is **not** true
 - Must use the *logical complement*:
 - $b.lt \longleftrightarrow b.ge$
 - $b.le \longleftrightarrow b.gt$
 - $b.eq \longleftrightarrow b.ne$
- Eg: C Code

```
if (a > b) {
    c = a + b;
    d = c + 5;
}
```

- Assembly Code

```
define(a_r, x19)
define(b_r, x20)
```

```
define(c_r, x21)
define(d_r, x22)

...

cmp    a_r, b_r // test compare
b.le   next     // logical complement

add    c_r, a_r, b_r //body
add    d_r, c_r, 5
next:  statement after if-construct
```

The if-else construct
-Is formed

Introduction to the gdb Debugger

- To start a program under debugger control, use:
 - gdb myprogram
- To set a breakpoint, type: *b label*
 - Eg. b main
- Use r to run your program
 - Will stop at the first breakpoint
- Use c to continue to the next breakpoint
 - Or to the end of the program, if no other breakpoints
- To *single step* through your program, use:
 - si
 - Executes the next instruction
 - ni
 - Also executes the next instruction
 - o But if a function call, proceeds until the function returns
- Use display/I \$pc to automatically show the current instruction when single stepping
 - Do before running the program
- Use p \$reg to print the contents of the register
 - Eg. p \$x19
 - Can append a format character:
 - Signed decimal: p/d
 - Hexadecimal: p/x
 - Binary: p/t
- Use q to quit gdb
- P&H: Sections 2.1 -2.3, 2.7
- ARMv8 instruction set overview: Sections:
 - 2
 - 3.1-3.1
 - 4.1-4.4
 - 5.1-5.2, 5.4.1, 5.5.1, 5.6

Binary Numbers and Integers Representations

Binary Numbers

- Are base 2 numbers
- Use only the binary digits (bits) 0 and 1

- Easy to encode on a computer, since only 2 states need to be distinguished
 - o Use voltage:
 - 0: 0V
 - 1: 3.3V
- Using paper tape or cards:
 - 0: unpunched
 - 1: punched
- Using toggle switches or bulbs
 - o 0: off
 - o 1: on
- -Binary numbers are encoded using 1 or more bits in combination
 - o Eg. 101 binary is decimal 5
- -An N-bit register can hold 2^N bit patterns
 - o Eg. a 4-bit register can hold 16 distinct bit patterns
- 0000, 0001, 0010, ..., 1111

Unsigned Integers

- Are encoded using binary numbers
 - o Range 0 to $2^N - 1$, where N is the number of bits
 - o Eg. 8-bit register: ranges from 0 to 225
 - In binary from 0000000 to 1111111
 -

Signed Integers

- Are most commonly encoded using the two's complement representation
 - o Range: $-2^{(N-1)}$ to $+2^{(N-1)} - 1$
 - 4-bits: -8 to +7
 - 8-bits: -128 to +127
 - 16-bits: -32,768 to +32,767
 - 32-bits: -2,147,483,648 to +2,147,483,647
 - 64-bits: $-2^{(63)}$ to $+2^{(63)} - 1$
- Negating a number is done by:
 - o 1. Taking the one's complement
 - toggle all 0's to 1's, and vice versa
 - o 2. adding 1 to the result
 - o Eg. finding the bit pattern for -5 in a 4-bit register
 - +5 is 0101
 - One's complement: 1010
 - add 1: 1011
 - -5 is 1011
- Also works when negating negative numbers
 - o Eg. -5 to +5
 - One's complement of 1011: 0100
 - add 1: 0101
- All positive numbers will have a 0 in the left-most bit
 - o And all the negative numbers will have 1
 - o Called the sign bit
- Eg: 4-bit clock-face:
- The *sign-magnitude* and *one's complement* representations are also possible
 - But are awkward to handle in hardware
 - Have to zeroes: +0 and -0
 - Rarely used today

Hexadecimal Numbers

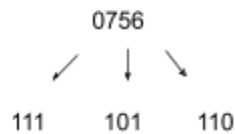
- Are base 16 numbers
- Use the digits 0, 1, 2, ..., 9, A, B, C, D, E, F
- Are commonly used as a shorthand for denoting bit patterns
 - Each hex digit corresponds to a particular 4-bit pattern

0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111



Octal Numbers

- Base 8 numbers
- Use digits: 0, 1, 2, ..., 7
- May be used as a shorthand for denoting bit patterns
- Each digit corresponds to a 3 bit pattern



Integer classes and subtypes

- Linux on ARMv8 in AArch64 uses the LP64 data model

- Long ints and pointers are 64-bit long

A64 Keyword	Size in bits	C Keyword
byte	8	char
halfword	16	Short int
word	32	int
doubleword	64	Long int, void *
quadword	128	---

- In C, use the “unsigned” keyword to denote unsigned integers
 - unsigned int x; //32 bits
 - unsigned char y; //8bits
- Readings and Exercises
- P & H section 2.4

Bitwise Operators

Bitwise logical instructions

- Manipulate one or more bits in a register
- AND

-

a	b	a AND b
0	0	0
0	1	0
1	0	0
1	1	1

- Form (64-bit): and Xd, Xn, Xm
 - mov x19, 0xAA // 1010 1010
 - mov x20, 0xF0 // 1111 0000
 - and x21, x19, x20 // 1010 0000
- Note that 0xF0 forms a *bitmask*
 - It “masks out” all bits except bits 4-7
- 32-bit and immediate forms also exist
- Eg:


```
mov w19, 0x55 // 0101 0101
add w19, w19, 0xF // 0000 1111
// 0000 0101
```
- “Ands” sets or clears N or Z flags according to the result (V and C always cleared)
 - Eg: test if bit 3 is a set in x20

...

- ands x19, x20, 0x8
 - b.eq bitclear
 - bitset: ...
 - bitclear: ...
- “tst” is an alias for “ands”
 - Form (64-bit): tst xn, xm
 - Alias for: ands xzr, xn, xm
 - Eg: ...
 - tst x20, 0x8
 - b.eq bitclear
 - bitset: ...
 - bitclear: ...
- **Inclusive OR**
 - Truth table: I’m not fucking writing a truth table, if there’s four instances, it goes true, true, true, false
 - Only false if both are false. Truth tables can suck it.
 - Form (64-bit): orr xd, xn, xm
 - 32-bit and immediate forms also exist
 - Eg: set bits 4 and 5 in x19
 - mov x19, 0xAA // 1010 1010
 - mov x20, 0x30 // 0011 0000
 - orr x19, x19, x20 // 1011 1010
 - “mov” is an alias of orr
 - Form: mov xd, xm
 - Alias for: orr xd, xzr, xm

- Exclusive OR

- Truth table:

-

a	b	b EOR a
0	0	0
0	1	1
1	0	1
1	1	0

- Note, the difference is the last right table, where both a and b are 1, and a EOR b is 0
- Form (64-bit): eor xd, xn, xm
 - 32-bit and immediate forms also exist
- Eg: toggle bits 0-3 in w20
 - mov w20, 0x55 // 0101 0101
 - eor w20, w20, 0x0f // 0000 1111
 - // 0101 1010 - Note, 1 with 1 is 0, one of each is 1

- Bit Clear

- Is actually the logic operation “AND NOT” (do NOT first, and then do AND with the result of that)
- Truth table:

-

a	b	a AND NOT b
---	---	-------------

0	0	0
0	1	0
1	0	1
1	1	0

- Form (64-bit): `bic` `xd, xn, xm`
 - 32-bit form also exists (but no immediate)
- Eg: clear bits 2-5


```
mov    x20, 0xFF    // 1111 1111
mov    x21, 0x3C    // 0011 1100
bic    x19, x20, x21 // 1100 0011
```

- **OR NOT**

- Truth table:
-

a	b	a OR NOT b
0	0	1
0	1	0
1	0	1
1	1	1

- Form (64-bit): `orn` `xd, xn, xm`
 - 32-bit form also exists (but no immediate)

- **NOT**

- Truth table:
-

a	NOT a
0	1
1	0

- Implemented using `mvn` (move NOT)
- Form (64-bit): `mvn` `xd, xm`
 - Alias for: `orn` `xd, xzr, xm`
 - 32-bit form also exists, (but no immediate)

- **EOR NOT**

- Truth table:
-

a	b	a EOR NOT b
0	0	1
0	1	0
1	0	0

1	1	1
---	---	---

- Form (64-bit): `eon xd, xn, xm`
 - 32-bit form also exists (but no immediate)

Master Truth Table:

a	b	NOT a	a AND b	a OR b	a EOR b	a AND NOT b	A OR NOT b	a EOR NOT b
0	0	1	0	0	0	0	1	1
0	1	1	0	1	1	0	0	0
1	0	0	0	1	1	1	1	0
1	1	0	1	1	0	0	1	1

Bitwise Shift Instructions

- Logical Shift Left
 - Form (64-bit): `lsl xd, xn, xm`
 - Xn: bit pattern to be shifted
 - Xm: shift count
 - 0 is shifted into rightmost bit
 - Shifted out bits are lost
 - LSL is a quick way to do multiplication by a power of two
 - I.e. by 2^n , where n is the shift count
 - Eg: 5×8

```

mov    x20, 5           // 0000 0101
mov    x21, 3           // 2^3 = 8
lsl     x19, x20, x21    // 0010 1000 (which = 40)

```
 - 32-bit and immediate forms exists
 - Eg:

```

mov     w20, 5           // 0000 0101
lsl     w19, w20, 3      // 0010 1000 (which also = 40)

```
- Logical Shift Right
 - Form (64 bit) `lsr, xd, xn, xm`
 - Xn: Bit pattern to be shifted
 - Xm: Shift count
 - 0 is shifted into leftmost bit
 - Shifted out bits are lost
 - LSR Is a way to division by a power of two
 - Any remainder is lost
 - Eg: $15 / 4 = 3$

```

mov    x20, 15 // 0000 1111
mov    x21, 2  // 2^2 = 4
lsr    x19, x20, x21 // 0000 0011

```
 - Does not work for negative signed integers
 - Use ASR to preserve the sign bit
 - 32 bit and immediate forms exist
 - 0 is inserted at 31 bit when using W operands
- Arithmetic shift right
 - Form 64 bit: `asr xd, xn, xm`
 - Xn: Bit pattern to be shifted

- Xm: Shift count
- Sign bit is duplicated when shifting
 - Called *sign extension*
- ASR preserves the sign when dividing by a power of two
 - Eg: $-8 / 2 = -4$
`mov x20, -8 // 1111 ... 1111 1000`
`mov x21 1 // $2^1 = 2$`
`Asr x19, x20, x21 // 1111 ... 1111 1100`
- 32 bit and immediate forms exist
 - Bit 31 is the sign bit when using w operands
- Rotate Right
 - Form (64 bit): `ror Xd, Xn, Xm`
 - Xn: Bit pattern to be rotated
 - Xm: Shift count
 - Bits shifted out on the right are inserted on the left
 - 32 bit form uses the bits 0-31 only

Sign/Zero extend operations

- Signed Extend Byte
 - Form (32 bit): `sxtb, Wd, Wn`
 - Sign-extends bit 7 in Wn to bits 8-31
 - Eg:
`mov x20, 0xFF // 0000 ... 0000 1111 1111`
`Sxtb w19, w20 // 1111 ... 1111 1111 1111`

Eg:
`mov w20, 0x7F // 0000 ... 0000 0111 1111`
`Sxtb w19, w20 // 0000 ... 0000 0111 1111`
- Signed Extend Halfword
 - Form (32 bit): `sxth Wd, Wn`
 - Sign extends bit 15 in Wn to bits 16-31
- 64 bit forms exist for `sxtb` and `sxth`
 - Eg: `sxth w19, w20`
- Sign Extend Word
 - Form (64 bit): `sxtw Xn, Wn`
 - Sign-extends bit 31 to bits 32-63
- Unsigned Extend Byte
 - Form (32 bit only): `uxtb Wd, Wn`
 - Zero extends bits 8-31
 - Eg:
`mov w20, 0xFF // 0000 ... 0000 1111 1111`
`uxtb w19, w20 // 0000 ... 0000 1111 1111`
- Unsigned Extend Halfword
 - Form (32 bit): `uxth Wd, Wn`
 - Zero extends bits 16-31

Bitfield Operations

- Bitfield insert
 - Form (32 bit): `bfi Wd, Wn, #lsb, #width`
 - Wn: Source bit pattern
 - #lsb: Insertion Position

- #width: Number of bits in the bitfield
- Source bitfield occupies bits 0 to (*width*-1) in *Wn*
- Bits *lsb* to (*lsb* + *width* -) in *Wd* are replaced by the source bitfield
 - All other bits in *Wd* are unchanged
- Eg:

```
mov w20, 0xAA          //0000 ... 0000 1010 1010 (mov 4 to left)
mov w19, 0xFFFFFFFF    //1111 ... 1111 1111 1111
bfi w19, w20, 4, 5     //1111 ... 1100 1010 1111
```
- Unsigned Bitfield Insert in Zero (*ubfiz*)
 - Similar to *bfi* except that *Wd* is zeroed first
- Signed Bitfield Insert in Zero (*sbfiz*)
 - Similar to *ubfiz* except that bits on the left are sign extended

- Bitfield Extract and Insert Low
 - Form(32 bit): *bxfil Wd, Wn, #lsb, #width*
 - *Wn*: source bit pattern
 - *#lsb*: extraction position
 - *#width*: size of bits taken
 - Source bitfield is bits *lsb* to (*lsb* + *width* - 1) in *Wn*
 - Bits 0 to (*width*-1) in *Wd* will be replaced by the source bitfield
 - All other bits in *Wd* are unchanged
 - Eg:

```
mov w20, 0xAA          //0000... 0000 1010 1010
mov w19, 0xFFFFFFFF    //1111... 1111 1111 1111
bxfil w19, w20, 6, 3    //1111... 1111 1111 1010
```

We take the bit at position index 6, make it 3 wide, and move it all the way to the left (indicated in red)

- Unsigned Bitfield Extract (*ubfx*)
- Signed Bitfield Extract (*sbfx*)

Readings and Exercises

P&H : Section 2.6

ARMv8 Instruction Set OverviewL

- Section 5.4, subsection 2,5,7,8
- Section 5.5, subsection, 3, 4

Binary Arithmetic

1. Modulus Arithmetic

- Constrains numbers to the range 0 to M-1, where M is the *modulus*
- CPUs normally do modulus arithmetic
 - Calculation results must fit into a fixed-size register
 - If *n* is the size in bits, then $M = 2^n$
 - E.g: M is 16 for a 4 bit register
 - Unsigned integers range from 0-15 (0000 to 1111)
 - Any carry out is ignored when using ordinary arithmetic instructions
 - e.g: $9+8=17$ BUT on a 4-bit CPU looks like this: **$(9+8)\text{mod}16$**
1001

$$\begin{array}{r} + \quad 1000 \\ 1 \quad 0001 \\ \hline \end{array}$$

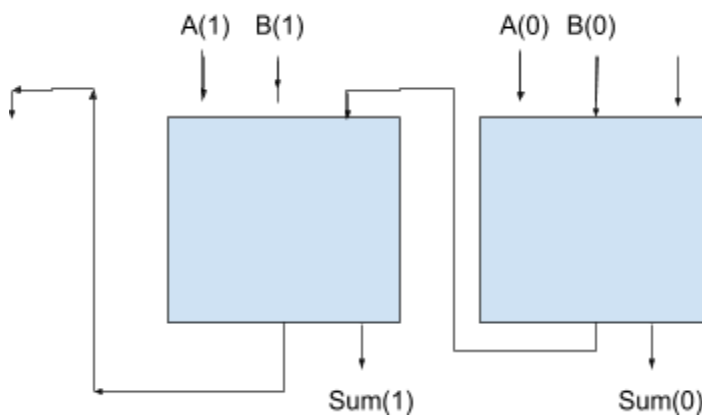
- Instructions like `adds` or `subs` do set the carry flag
 - Can be used to do extended precision arithmetic

2. addition

- Single bits can be added together using the following rules:

carry in	A	B	carry out	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

- Multi-bit numbers can be summed together using a full adder for each bit:



3. Subtraction

- Is done in the ALU by negating the subtrahend, and then adding
 - Reuses the addition circuitry, thus minimizing hardware complexity
 - 4 bit example: $7 - 5 = 7 + (-5) = 2$

```

0111
+ 1011
1 0010    // 1 is a carry out which is ignored

```
- Recall that signed branch instructions use the N, Z, V flags: **(refer to the branch instruction table in previous instructions)**
- The *subs* instruction (alias of *cmp*) sets the flags using these rules (64-bit form)


```

N = (Xd<63> == 1)
Z = (Xd == 0)
V = ((Xn<63> & ~Xm<63> & ~Xd<63>) | (~Xn<63> & Xm<63> & Xd<63>))

```
- Note that overflow occurs when the sign bits for Xn, Xm, Xd are: 1 0 0, or 0 1 1

Signed number branching conditions cont'd

- 4-bit example:
 - $-8 - 5$ is -13, which is out of range (the modulus result is 3)


```

1000          1000
- 0101  same as: + 1011
0011          0011

```

The bold: V set to 1, N to 0, Z to 0
 - Note that:
 - $-8 \neq 5$, since $Z == 0$
 - $-8 < 5$, since $N \neq V$
 - $-8 \leq 5$, since $!(Z == 0 \ \&\& \ N == V)$

Signed number branching conditions cont'd

- Another 4-bit example
 - $0 - (-8)$ is 8, which is out of range (the modulus result is -8)


```

0000          0000
-1000  same as: +1000
0000          1000

```

V set to 1, N to 1, Z to 0
 - Note that:
 - $0 \neq -8$, since $Z == 0$
 - $0 > -8$, since $Z == 0 \ \&\& \ N == V$
 - $0 \geq -8$, since $N == V$

Unsigned Arithmetic:

- Uses the same registers and hardware operations as signed arithmetic
 - However, one interprets the data as unsigned numbers
- When adding, a carry out ($C = 1$) indicates overflow
 - 4-bit example: $15 + 1$:


```

1111
+0001
1 0000 (1 is the carry set)

```


Unsigned Number Branching Conditions:

- Ignore N and V flags, since they have no meaning for unsigned integers:

-

Name:	Meaning:	C Equivalent:	Flags:
eq	equal	==	Z == 1
ne	Not equal	!=	Z == 0
hi	Unsigned higher	>	C == 1 && Z == 0
hs	Unsigned higher or same	>=	C == 1
lo	Unsigned lower	<	C == 1
ls	Unsign. low.or same	<=	!(C == 1 && Z == 0)

- Must use these condition codes in conditional branches when comparing unsigned integers

- Eg: C code

Unsigned int a, b;

...

If (a < b) {

...

}

- Assembly:

...

cmp a_r, b_r

b.hs endif //logical complement

...

endif: ...

Multiplication:

- Can be done with an integrated procedure
 - Repeated for every bit in the source registers
- Each step consists of two sub-steps:
 - A conditional addition
 - An arithmetic shift right
- The product may require twice as many bits as for the source registers
 - The result is put into 2 concatenated registers
- If the original multiplier is negative, an extra step is needed:
 - Subtract the multiplicand from the high-order part of the result (i.e. from the product register)
- 4-bit example: 2 x -5 0010 x 1011
 - Initial setup:

Multiplicand	Product	multiplier
[0010]	[0000]-----	[1011]

Connected, as final result uses all 8 bits concatenated together
 - Step 1:

- If LSB(insertion position) of multiplier == 1, add multiplicand to product
[0010]---[1011]
- Arithmetic shift right
[0001]---[0101]
- Step 2:
 - If LSB of multiplier == 1, add multiplicand to product
 - [0011]---[0101]
 - Arithmetic shift right
 - [0001]---[1010]
- Step 3:
 - If LSB of multiplier == 0, do nothing
 - [0001]---[1010]
 - Arithmetic shift right
 - [0000]---[1101]
- Step 4:
 - If LSB of multiplier == 1, add multiplicand to product
 - [0010]---[1101]
 - Arithmetic shift right
 - [0001]---[011]
- (Remember the original value was negative? (-5), so we do additional step)
- Since the original multiplier is negative, subtract multiplicand from product
 - I.e. add two's complement of 0010, which is 1110
 - (negate the number, and add instead of subtract)
[1111]---[0110]
 - Result is 11110110, which is -10
- Early RISCs used this technique for multiplication, embodied in a library function
 - Now done in hardware

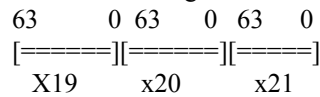
Division:

- Can be done using an iterated procedure similar to manual long division
 - Eg: $245 / 5 = 49$
 - Long division:

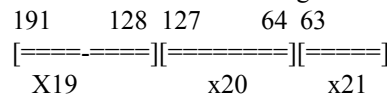
110001 <-- quotient
101|11110101
-101
0101
-101
0000101
-101
000
- Early RISC's used a similar technique, embodied in a callable library function
 - Now done in hardware
- Slowest instruction on a machine.

Extended Precision Arithmetic:

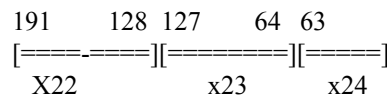
- Is used when you need more than 64 bits of precision
- Achieved by using two or more registers to represent a number
 - Eg: 192-bit unit uses 3 registers



- Extended precision numbers can be added together



+



- Procedure:
 - $X21 + x24$
 - Set the C flag to 0 or 1
 - $X20 + x23 + C$ (incorporate the carry generated by first step)
 - Set the C flag to 0 or 1
 - $X19 + x22 + C$
- The *adc* and *adcs* in instruction use the C flag when adding
 - Eg:


```
adds    x27, x21, x24    // bits 0-63
adcs    x26, x20, x23    // bits 64-127
adc     x25, x19, x22    // bits 128-191
```
- Subtraction is done in a similar way using the *subs*, *sbc*, and *sbc* instructions

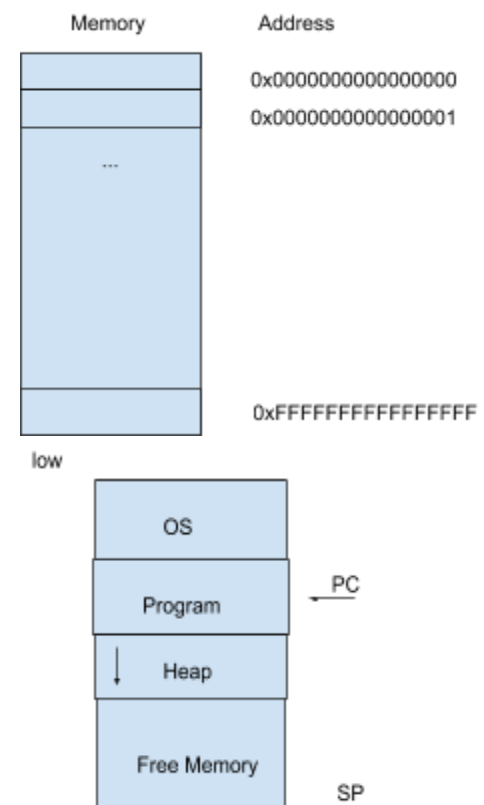
Readings and exercises:

- P & h: section 3.1-3.4
- Armv8 instruction set overview:
 - Section 5.5.6

The Stack

Memory and Memory addressing

- ARMv8 uses 64-bit addresses
 - Can address 2^{64} bytes of memory:
 - Memory:
 - This is a *virtual memory* address space (virtual = not physical, is an abstraction.)
 - Is mapped to *physical memory* by the OS and hardware
- A register can be loaded with 1, 2, 4, or 8 byte data from RAM
 - Data is placed into the low-order bits
 - If necessary, high-order bits are:
 - Sign-extended, if loading signed data
 - Zero-extended, if loading unsigned data



- Byte, halfword, word, or doubleword data in a register can be *stored* into RAM
 - Low-order bits of a register when the data is just part of a register
- Space in RAM provided by the OS to store data for functions
 - A *stack frame* is *pushed* onto the stack when the function is called
 - Holds the function's parameters, local variables, and return values
 - The frame is *popped* when the function returns
 - The stack uses high memory
 - It grows "Backwards" (towards 0)
 - Programs are loaded into low memory, just above the space reserved for the OS
 - The *heap* is used for dynamically allocated memory in a program
 - Done in C using `malloc()` and `free()`

The Stack and Frame Pointers

- The stack pointer register SP always points to the top of the stack
 - Is *decremented* when the stack grows
 - A program can allocate stack memory by subtracting the number of bytes needed from SP
 - Eg: `sub sp, sp, 16`
 - Decrement SP by 16 bytes
 - The stack must be word *aligned*
 - Will give you a bus error and crash :(
 - The address in SP must be evenly divisible by 16
 - May need to allocate more space than actually needed
 - To guarantee alignment:
 - add a negative number instead of subtracting
 - Clear the low 4 bits of this number
 - AND it with -16 (1111...11110000)
 - Eg: allocate 20 bytes
`add sp, sp, -20 & -16`
 - Actually allocates 32 bytes
 - The assembler evaluates -20 & -16, and substitutes the calculated constant (-32)

The Stack and Frame pointers

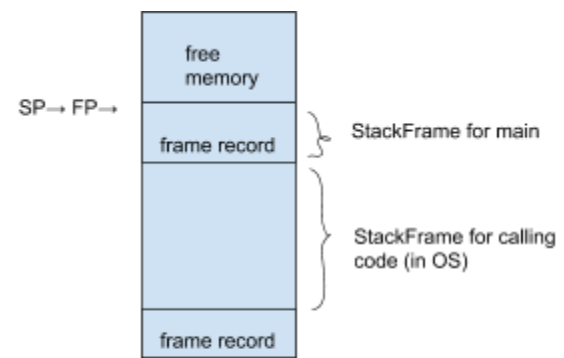
- The frame pointer (FP) is register x29
- Is used to point to local variables in a stack frame
 - Is stable, once set at the beginning of a function
 - In contrast, SP is unstable
 - Allowed to change as the function executes

Stack frames and frame records

- A *stack frame* is *pushed* onto the stack when entering a function
 - Must be at least 16 bytes long
 - Is created by the pre-increment part of the *stp* instruction
 - Eg: allocate 16 bytes for main

```
        .global main
Main:    stp    x29, x30, [sp, -16]! ← allocates 16 bytes on stack (sp=sp+16)
        mov    x29, sp
        ...
```

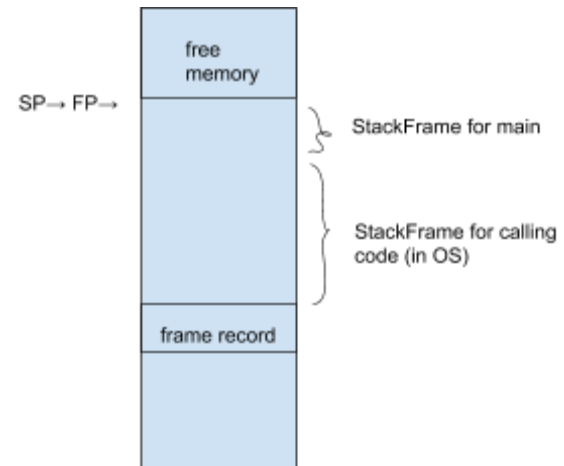
- *Stp* also stores the contents of x29 (FP) and x30 (LR) to these 16 bytes



- This part of the stack frame is called the *frame record*
- *mov* sets x29 (FP) to point to the frame record
 - Should not change until the end of the function
 - Is a stable “anchor” used to refer to elements in the stack frame
 - In contrast SP will change when allocating/deallocating temporary variables for *main()*
 - Once *stp* and *mov* have executed, the stack appears as follows
- The stack frame is *popped* when the function returns to calling code (I’m having troubles typing today)
 - Is done by the post-increment part of the *ldp* instruction
 - Must negate the number used in *stp*
- Deallocate 16 bytes

```
...
Ldp x29, x39, [sp], 16
Ret
```

- *ldp* also restores x29 and x30 from *main()*’s frame record
 - Will now have the values used in calling code
- The stack appears as follows, once *ldp* and *ret* have executed:



Stack Variables

- Are created by allocating extra space in the stack frame when entering a function
 - Is in addition to the frame record’s 16 bytes
 - Eg: three 4-byte ints

```
...
main:    stp    x29, x30, [sp, -(16+12) & -16]!
...
```

- We allocate 32 bytes total
 - There are 4 *pad bytes*, which remain unused
- The stack appears as:
- Stack variable are addressed by adding an *offset* to the *base address* in FP

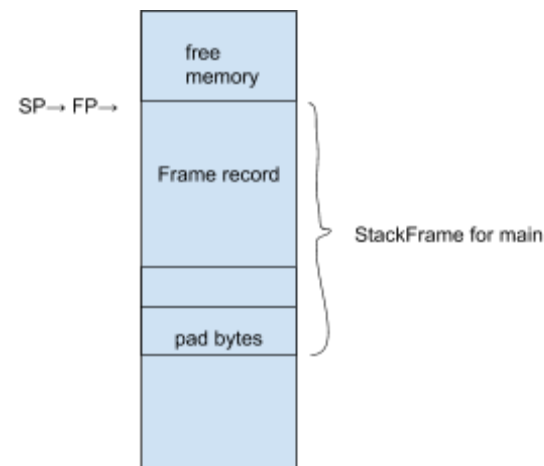
- Eg: FP + 16, FP + 20, FP + 24

- addresses are specified inside [] of load and store instructions
 - Eg: Write 42 to the first stack variable

```
...
mov     w20, 42
str     w20, [x29, 16]
```

- Eg: Read from the second stack variable

```
...
ldr     w21, [x29, 20]
...
```



Memory Alignment

- Many architectures require aligned memory access
 - An n-byte unit must be at the address evenly divisible by n
 - Eg: a 2 byte *short int* must have an address evenly divisible by 2
 - Pad bytes are inserted as needed to ensure alignment
 - Wastes memory!
 - On ARMv8, unaligned memory accesses are generally allowed, with these exceptions
 - addresses in SP must be quadword aligned
 - Frame record must be at an address evenly divisible by 16
 - If not, the program will abort with a *bus error*
 - Machine instructions must be word-aligned
 - To ensure this, use: `.balign 4`
 - *Exclusive load/store* accesses must be aligned
 - Not used in this course

Load Register

- Load Register
 - 64 bit form: `ldr Xt, addr`
 - Xt: register to be loaded
 - addr: expression specifying the address in memory to read from
 - Loads register with 8 bytes read from RAM
 - 32 bit form: `ldr Wt, addr`
 - Loads register with 4 bytes from RAM
- Load byte
 - Form 32-bit only: `ldrb wt, addr`
 - Loads 1 byte from RAM into low order part of Wt, zero extending high order bits
- Load Signed Byte
 - Form(32): `ldrsh Wt, addr`
 - Form(64): `ldrsh Xt, addr`
 - Loads 1 byte from RAM into low order bit part of wt or xt, sign extending high order bits
- Load Halfword
 - Form 32-bit only: `ldrh wt, addr`
 - Loads 2 bytes of RAM into wt, zero extending high order bits
- Load Signed Halfword
 - Form(32): `ldrsh Wt, addr`
 - Form(64): `ldrsh Xt, addr`
 - Loads 2 bytes from RAM into Wt or Xt, sign extending high order bits
- Load Signed Word
 - Form 64-bit only: `ldrsw Xt, addr`
 - Loads 4 bytes from RAM into Xt, sign-extending high order bits
- Store Register
 - Form 64-bit: `str Xt, addr`
 - Xt: register to be stored
 - addr: expression specifying the address in memory to write to
 - Store a doubleword(8 bytes) in Xt to RAM
 - E.g: `str x20, [x29, 56]` ← Count 56 bytes down from FP(x29), move x20 there
 - Stores word(4 bytes) in wt to RAM
- Store Byte
 - Form(32 bit only): `strb wt, addr`
 - Stores low order byte in Wt to RAM
- Store Halfword
 - Form(32-bit only): `strh, Wt, addr`
 - Stores low order halfword(2 bytes) in Wt to RAM

Load/Store addressing modes

- The following are possible for the basic load/store instructions

- Base plus immediate offset
 - Form: [base, #imm]
 - base: an X register or SP
 - #imm: a 9 bit constant(range -256 to 255) or a 12 bit unsigned constant(range: 0 to 4095)
 - E.g: `ldr x20, [x29, 16]`
- Base plus 64-bit offset:
 - Form: [base, #imm]
 - base: an X register or SP
 - Xm: an x register containing an offset
 - E.g: `mov x21, 16`
`ldr x22, [x29, x21]`
 - Optionally, xm can be scaled using LSL #imm
 - E.g: lets add 5x8 to base in x29
 - `mov x21, 5`
 - `ldr x22, [x29, x21, LSL 3]`
- Base plus 32-bit register offset
 - Sign extended form: [base, Wm, SXTW]
 - Zero extended form: [base, Wm, UXTW]
 - base: an X register or SP
 - Wm: a W register containing an offset
 - adds sign or zero extended offset in Wm, to base
 - E.g:
 - `mov w21, -16`
 - `ldr x22, [x29, w21, SXTW]`
- An optional LSL scaling is possible
- E.g: Scale offset in w21 by 4
 - `mov w21, -16`
 - `ldr x22, [x29, w21, SXTW 2]`
- Pre-indexed by immediate offset
 - Form: [base, #imm]!
 - base: an X register or SP
 - #imm: a 9-bit signed constant(range -256 to 255)
 - base is first updated by adding #imm to it
 - Then this address is used for the load/store
 - E.g:
 - ...
 - `add x28, x29, 16 //RAM address is x28 = x29+24`
 - `str w20, [x28, 8]!`
- Pos-indexed by immediate offset
 - Form: [base], #imm
 - base: an X register or SP
 - #imm: a 9-bit signed constant(range -256 to 255)
 - The address in base is used for the load/store
 - Then base is updated by adding #imm to it
 - E.g:
 - `add x28, x29, 16`
 - `str w20, [x28], 8 //RAM address is x28 = x29+24`

Stack variable offset macros:

- M4 macros can be used for offsets to improve readability

- Eg:

```
define(a_s, 16)
define(b_s, 20)
...
str    w20, [x29, a_s]
ldr    w21, [x29, b_s]
```
- Can also be done with assembler *equates*:
 - Eg:

```
a_s = 16
b_s = 20
...
str    w20, [x29, a_s]
ldr    w21, [x29, b_s]
```
- *Register equates* are useful for renaming x29 and x30 to FP and LR
 - Eg:

```
fp     .req x29
lr     .req x30
...
str    w20, [fp, a_s]
ldr    w21, [fp, b_s]
```

Local Variables:

- In c, can be declared in a block of code
 - I.e. in any construct delimited by {...}
 - Eg:

```
int main() {
    int a = 5, b = 7;
    if (a < b) {
        int c;
        C = 10;
        ...
    }
}
```
- Local variables are implemented as stack variables in assembly
 - Those local to the function are allocated when entering the function, as already shown
 - Those local to other blocks of code are:
 - Allocated when entering the block
 - Done by directly decrementing SP
 - Are read/written using FP plus a *negative* offset
 - Deallocated when leaving the block
 - Done by incrementing SP
- Eg: equivalent assembly code:

```
a_s = 16
b_s = 20
c_s = -4
...
main: stp    x29, x30, [sp, -(16+8) & -16]!
      mov    w19, 5                // init a to 5
      str    w19, [x19, a_s]
      mov    w20, 7                // init b to 7
      str    w20, [x29, b_s]
      ..
```



```

        cmp    w19, w20
        B.ge   next
        // start of block of code for if else
        add    sp, sp, -4 & -16 // alloc RAM for c
        mov    w21, 10
        str    w21, [x29, c_s]
        ...
        / end of block of code
        add    sp, sp, 16
Next: ...

```

- After allocating RAM for c, the stack appears as:

SP Arrow points between free memory and pad bytes

FP Arrow points between c and frame record

Everything from pad bytes to pad bytes } = stack frame for main

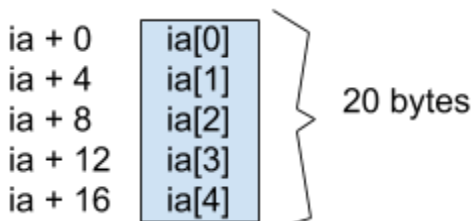
Free memory
Pad bytes
c
Frame record
a
b
Pad bytes

Data structures:

One-dimensional Arrays:

- Store consecutive array elements of the same type in a contiguous block of memory
 - Block size = number of elements x elements size
 - address of element *i* is:
 - Base address + (*i* x element size)
 - Eg:

Int ia[5]; // gives you that:



- Like other local variables, an array is allocated in the function's stack frame
 - Eg C Code:


```

int main(){
    int a, b, ia[5];
    ...
}

```
 - Assembly code:

a_size = 4

- "I'm helping" - Jas

```
    b_size = 4
    ia_size = 5 * 4
    alloc = -(16 + a_size + b_size + ia_size) & -16
    dealloc = -alloc
    a_s = 16
    b_s = 20
    ia_s = 24
    ...
main: stp    x29, x30, [sp, alloc]!
      mov    x29, sp
      ..
      ldp    x29, x30, [sp], dealloc
      ret
```

- Memory is used as follows:
- Ask sim 4 drawing, 2 lazy

One Dimensional Arrays:

- Array elements are accessed using load and store instructions
- E.g: `ia[2] = 13;`

```
define(ia_base_r, x19)
define(index_r, x20)
define(offset_r, x21)

...

add    ia_base_r, x29, ia_s           //calculate array base address
mov     index_r, 2                    //Set index to 2
lsl     offset_r, index_r, 2          //offset = i*elem. size

mov     w22, 13
str     w22, [ia_base_r, offset_r]    //ia[2] = 13
```

Can be optimized by doing the LSL in the STR instruction:

```
mov     index_r, 2                    //set index 2
mov     w22, 13
str     w22, [ia_base_r, offset_r, LSL 2]
```

If index is a w register, we must sign extend

```
define(index_r, w20)
...

str     w22, [ia_base_r, index_r, SXTW 2]    //ia[2] = 13
```

Multidimensional Arrays

- Use 2 or more indices to access individual array elements

- Most languages use row major order when storing arrays in RAM
- i.e. first all elements of row 0, then row 1
- The block size for an n dimensional array is: (some symbol thing) $\text{dim} \times \text{element_size}$ - where
- dim is the number of elements in dimension r
- Eg: `int ia[2][3][4];`
- Block size = $(2 \times 3 \times 3) \times 4 = 72$ bytes
- The offset for a given element is:
- $[\sum_{r=1 \rightarrow n-1} [\text{index}_r \times \prod_{s=r+1 \rightarrow n} \text{dim}_s] + \text{index}_{\text{subscript } n}] \times \text{element size}$ (where index is the index for the element in dimension r)
- Eg: a 3-dimensional array would be declared with: `type array[dim1][dim2][dim3];`
Its offset for element `array[index1][index2][index3]` is: $[(\text{index1} \times \text{dim1} \times \text{dim3}) + (\text{index2} \times \text{dim3}) + \text{index3}] \times \text{element size}$
If the array is declared as:

```
int ia[2][3][4];
Its offset for ia[i][j][k] would be:
    [(i x 3 x 4) + (j x 4) + k] x 4
```

Eg: C code

```
Int main()
{
    Int ia[2][3];
    register int i, j;
    ...
    Ia[i][j] = 13;
    ...
}
```

Assembly code:

```
define(ia_base_r, x19)
define(offset_r, w20)
define(i_r, w21)
define(j_r, w22)

dim1 = 2
dim2 = 3
ia_size = dim1 + dim2 * 4
alloc = -(16 + ia_size) & -16
```

Well meme'd, my men

```
dealloc = -alloc
ia_s = 16

...
main: stp    x29, x30, [sp, alloc]!
mov   x29, sp
...
add   ia_base_r, x29, ia_s    //calculate array base address

mov   w23, dim2
mul   offset_r, i_r, w23 //offset = i * dim2
add   offset_r, offset_r, j_r    //offset = (i * dim2) + j
lsl   offset_r, offset_r, 2      //offset = ((i * dim2) + j) * 4

mov   w24, 13
str   w24[ia_base_r, offset_r, SXTW]    //ia[i][j] = 13
...
```

NOTE: can be optimized by doing LSL in STR instruction

structures:

- Contains fields, which may be of different types
- Are allocated in a single block of memory on the stack
 - Fields are accessed using offsets
-
-
- E.g:

```
struct rec{
    int a;
    char b;
    short c;
};
```

MEMORY	OFFSET
[a]	rec_a=0
[b]	rec_b = 4
[c]	rec_c = 5

```
ldr    w20, [x19, rec_a]
ldrsh  w21, [x19, rec_b]
ldrsh  w22, [x19, rec_c]
```

E.g: C code:

```
struct rec{
    int a;
    char b;
    short c;
};

int main()
{
    <- Defines a custom data type
```

Well meme'd, my men

```
    struct rec s1; <- allocates memory, struct rec being data type
    ...
    s1.a = 42;
    s1.b = 23;
    s1.c = 13;
}
```

NOW THE ASSEMBLY CODE:

```
define(s1_base_r, x19)
    rec_a=0
    rec_b = 4
    rec_c = 5

    s1_size = 7 //4+2+1
    alloc = -(16+s1_size) & -16
    dealloc = -alloc
    s1_s = 16

main: stp    x29, x30, [sp, alloc]!
      mov    x29, sp
      ...
      add    s1_base_r, x29, s1_s
      mov    w20, 42
      str     w20, [s1_base_r, rec_a] //s1.a = 42
      mov    w20, 23
      strb    w20, [s1_base_r, rec_b] //s1.b = 23
      mov    w20, 13
      strh    w20, [s1_base_r, rec_c] //s1.c = 13
```

Nested structure:

- A structure may contain a field whose type is another structure
 - The field's *subfields* are accessed using a 2nd set of offsets
- E.g C code:

```
struct date{
    unsigned char, day, month;
    unsigned short year;
};

struct employee{
    int id;
    struct date start; <- nested struct
};

int main()
{
    struct employee joe;                //Allocated memory for joe
    ...
    joe.id = 4001;
    joe.start.day = 1;
    joe.start.month = 6;
    joe.start.year = 1999;
    ...
}
```

Well meme'd, my men

Offset	Nested Offset	Memory	
0		[4001]	joe.id
4	0	[1]	joe.start.day
	1	[6]	joe.start.month
	2	[1999]	joe.start.year

Assembly:

```
define(joe_base_r, x19)

date_day = 0
date_month = 1          <- Offsets
date_year = 2

employee_id = 0
employee_start = 4
joe_size = 8
alloc = -(16+joe_size) & -16
dealloc = -alloc
joe_s = 16
...
main: stp x29, x30, [sp, alloc]!
      mov x29, sp
      ...
      add joe_base_r, x29, joe_s
      mov w20, 4001
      str w20, [joe_base_r, employee_id]           //joe.id =400

      mov w20, 1
      strb w20, [joe_base_r, employee_start +date_day]    //joe.start.day = 1

      mov w20, 6
      strb w20, [joe_base_r, employee_start + date_month]//joe.start.month=6

      mov w20, 1999
      strb w20, [joe_base_r, employee_start + date_year] //joe.start.year=1999
```

November 4, 2016

Subroutines:

- Use the ret instruction to return from a subroutine back to the calling code
 - o Transfers control to the address stored in the link register (x30)
 - \$ i.e. jumps to the instruction immediately following the original bl in calling code
- Eg: C code

```
int main()
{
    ...
    func1();
    ...
}
```

Well meme'd, my men

```
void func1()
{
    ...
    func2();
    ...
}

void func2()
{
    ...
}
```

○ Assembly Code:

```
main:      stp x29, x30, [sp, -16]!
           mov x29, sp
           ...
           bl func1
           ...
           ldp x29, x30, [sp], 16
           ret
func1:     stp x29, x30, [sp, -16]!
           mov x29, sp
           ...
           bl func2
           ...
           ldp x29, x30, [sp], 16
           ret
func2:     stp x29, x30, [sp, -16]!
           mov x29, sp
           ...
           ldp x29, x30, [sp], 16
           ret
```

§ The stp instructions create a frame record in each function's stack frame

· Safely stores the LR (x30), in case it is changed by a bl in the body of the function

○ Is restored by the ldp instruction, just before the ret

- The FP and the stored FP values in the frame records form a linked list

○ Eg: the stack while func2() is executing

- A called function must save/restore the state of the calling code

○ If it uses any of the registers x19-x28, it must save their data to the stack at the beginning of the function

§ Are "callee-saved registers"

○ The function must restore the data in these registers just before it returns

- Eg:

```
x19_size = 8
alloc = -(16 + x19_size) & -16
dealloc = -alloc
x19_save = 16
func2: stp x29, x30, [sp, alloc]!
       mov x29, sp
       str x19, [x29, x19_save]           // save x19
       ...
       mov x19, 13                       // use x19
       ...
       ldr x19, [x29, x19_save]           // restore x19
       ldp x29, x30, [sp], dealloc
       ret
```

- Note that the callee can also use registers x9 – x15

- By convention, these registers are not saved/restored by the called function
 - § Thus are only safe to use in calling code between function calls
- The calling code can save these registers to the stack, if it is necessary to preserve their value over a function call
 - § Are “caller-saved registers”

Arguments to Subroutines

- 8 or fewer arguments can be passed into a function using registers x0-x7

- ints, short int, and chars use w0-w7
- long ints use x0-x7

- Eg: C code

```
void sum(int a, int b)
{
    register int i;
    i = a + b;
    ...
}
int main()
{
    sum(3, 4);
}
```

- Assembly code:

```
define(i_r, w9)
sum:    stp x29, x30, [sp, -16]!
        mov x29, sp

        add i_r, w0, w1
        ...
        ldp x29, x30, [sp], 16
        ret
main:   stp x29, x30, [sp, -16]!
        mov x29, sp
```


Well meme'd, my men

```
    mov w0, 3      // set up 1st arg
    mov w1, 4      // set up 2nd arg
    bl sum
    ...
    ldp x29, x30, [sp], 16
    ret
```

○ Note that the subroutine is free to overwrite registers x0 – x7 as it executes
i.e. register contents are not preserved over a function call

Pointer arguments

- In calling code, the address of a variable is passed to the subroutine
- Implies that the variable must be in RAM, not in a register

e.g C code:

```
int main()
{
    int a = 5, b=7;
    swap(&a, &b);
}
void swap(int *x, *y)
{
    register int temp;

    temp = *x;
    *x = *y;
    *y = temp;
}
```

Assembly:

```
    a_size = 4
    b_size = 4
    alloc = -(16+a_size+b_size)& -16
    dealloc = -alloc
    a_s = 16
    b_s = 20

    ...
main: stp x29, x30, [sp, alloc]!
      mov x29, sp

      mov w19, 5          //init a to 5
      str w19, [x29, a_s]
      mov w20, 7          //init b to 7
      str w20, [x29, b_s]

      add x0, x29, a_s     //set up 1st arg
      add x1, x29, b_s     //set up 2nd arg
      bl swap
      ...
define(temp_r, w9)
```

Well meme'd, my men

```
swap: stp x29, x30, [sp, -16]!
      mov x29, sp

      ldr temp_r, [x0]          // temp = *x
      ldr w10, [x1]             // w10 = *y
      str w10, [x0]             // *x = w10
      str temp_r, [x1]          // *y = temp
      ldrp x29, x30, [sp], 16
      ret
```

Returning Integers:

- A function returns:
 - Long ints in x0
 - ints, short ints, and chars in w0
- E.g: Cube in function in C

```
int main(){
    register int result;
    result = cube(3);
}
```

```
int cube(int x){
    return x*x*x;
}
```

Assembly:

```
define(result_r, w19)

main: stp x29, x30, [sp, -16]!
      mov x29, sp
      mov w0, 3
      bl cube
      mov result_r, w0
      ...
cube: stp x29, x30, [sp, -16]!
      mov x20, sp

      mul w9, w0, w0
      mul w0, w9, w0

      ldp x29, x30, [sp], 16
      ret
```

Returning structures

- In C, a function may return a struct by value
- E.g:

```
struct mystruct{
    long int i;
    long int j;
}
```

```
struct mystruct init(){
    struct mystruct lvar;
    lvar.i = 0;
    lvar.j = 0;
```

Well meme'd, my men

```
    return lvar;
}
```

```
int main(){
    struct mystruct b;

    b = init();
}
```

- Usually a struct is too big to return in x0 or w0
 - Thus another return mechanism is needed
- the calling code provides memory on the stack to store the returned result
 - The address of this memory is put into **x8** prior to the function call
 - **x8 is the “indirect result location register”**
 - The called subroutine writes to memory at this address, using x8 as a pointer to it
- Assembly:
mystruct_i = 0
mystruct_j = 8

```
b_size = 16
alloc = -(16 + b_size) & -16
dealloc = -alloc
b_s = 16

...
```

```
main: stp x29, x30, [sp, alloc] !
      mov x29, sp
      add x8, x29, b_s
      bl init
      ...
```

```
define(lvar_base_r, x9)
```

```
    lvar_size = 16
?>?>
    alloc = -(16 + lvar_size) & -16
    dealloc = -alloc
    lvar_s = 16
```

```
init: stp x29, x30, [sp, alloc]!
      mov x29, sp
```

```
//calc lvar struct base address
add lvar_base_r, x29, lvar_s
```

```
str xzr, [lvar_base_r, mystruct_i] //lvar.i = 0
str xzr, [lvar_base_r, mystruct_j] //lvar.j = 0
```

```
ldr x10, [lvar_base_r, mystruct_i]
str x10, [x8, mystruct_i]
ldr x10, [lvar_base_r, mystruct_j]
str x10, [x8, mystruct_j]
```

```
ldp x29, x30, [sp], dealloc
Ret
```

Optimizing Leaf Subroutines

- Leaf subroutines do not call any other subroutines
 - I.e. are leaf nodes on a structure diagram
- A frame record is not pushed onto the stack
 - Since the routine does not do a bl, LR won't change
 - Since the routine does not call a subroutine, FP won't change
 - Thus can eliminate the usual stp/ldp instructions
- If one uses only the registers x0-x7 and x9-x15, the na stack frame is not pushed at all
- Eg: optimized cube function

```
Cube: mul    w9, w0, w0    // w9 = x * x
      mul    w0, w9, w0    // w0 = x*x*x
      ret                    // result is in w0
```

Subroutines with 9 or more arguments:

- Arguments beyond the 9th are passed on the stack
 - The calling code allocated memory at the top of the stack, and writes “spilled” argument values there
 - By convention, each argument is allocated 8 bytes
 - The callee reads this memory using the appropriate offset

- Eg: c code

```
int sum(int a1, int a2, int a3, int a4, int a5, int a6, int a7,
        int a8, int a9, int a10);

Int main() {
    register int result;
    Result = sum(10, 20, 30, 40, 50, 60, 70, 80, 90, 100);
}

int sum(int a1, int a2, int a3, int a4, int a5, int a6, int a7,
        int a8, int a9, int a10) {
    Return a1+ a2+ a3+ a4+ a5+ a6+ a7+ a8+ a9+ a10;
}
```

- Eg: ASM code

```
define(result_r, w19)
    Spilled_mem_size = 16
.global main
Main: stp    x29, x30, [sp, -16]!
      mov    x29, sp

      // Set up first 8 args
      mov    w0, 10
      mov    w1, 20
      mov    w2, 30
      mov    w3, 40
      mov    w4, 50
      mov    w5, 60
      mov    w6, 70
      mov    w7, 80
```

Well meme'd, my men

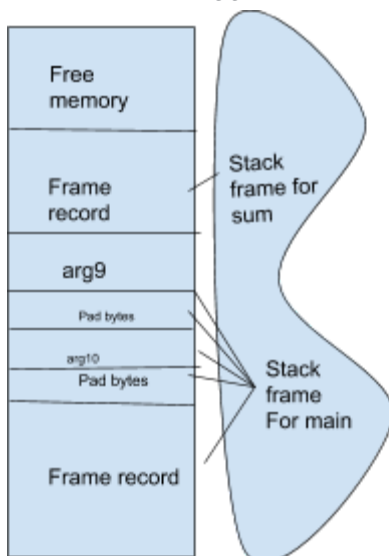
```
// allocate memory for args 9 and 10
add    sp, sp, -Spilled_mem_size & -16

// write spilled arguments to top of stack
mov     w20, 90
str     w20, [sp, 0] // setup arg 9
mov     w20, 100
str     w20, [sp, 8] // setup arg 10
bl      sum          // call the sum function
mov     result_r, w0 // result is in w0

//deallocate memory for spilled arguments
add     sp, sp, spilled_mem_size
...
arg9_s = 16
arg10_s = 24
Sum:    stp     x29, x30, [sp, -16]!
        mov     x29, sp
        add     w0, w0, w1
        add     w0, w0, w2
        add     w0, w0, w3
        add     w0, w0, w4

        add     w0, w0, w5
        add     w0, w0, w6
        add     w0, w0, w7

// add the 9th and 10th arg
ldr     w20, [x29, arg9_s]
add     w0, w0, w20
ldr     w20, [x29, arg10_s]
add     w0, w0, w20          // result in w0
Ldp     x29, x30, [sp], 16
Ret
```



i am an ARTIST

Chris 4 prez tbh

External data and text

Introduction:

- In the c language, local (automatic) variables are always allocated in the stack frame for a function
 - Scope: local to block code where declared
 - Lifetime: life of the block of code
 - It persists from call to call of the function
 - eg:

```
Int count() {
    Static int value = 0; // static local var (value)
    Return ++value;
}
```
 - Cannot be stored on the stack frame for the function
 - Stack memory may be reused by other functions
 - Are stored in a separate section of ram

Nov 14th

- Global variables:
 - Scope: global(from declaring onwards)
 - Lifetime: life of program
 - Are stored in a separate section of ram
 - Eg: c code:

```
int val; // global variable
main() {
    val = 3;
    ...
}
int f() {
    int a;
    a = val;
    ...
}
```
- *Static global variables*
 - Scope: local to file (from declaration onwards)
 - Lifetime: life of program
 - Stored in a separate section of RAM
 - Eg: c code:

```
static int val; // global variable
main() {
    val = 3;
    ...
}
```

The text, data, and .bss sections:

- Programs may allocate 3 sections of memory
 - *Text*
 - Contains:
 - Program text (machine code)
 - Read-only, programmer-initialized data
 - Is read-only memory
 - Attempts to write to this memory causes a *segmentation fault*
 - *Data*
 - Contains programmer-initialized data

- Is read/write memory
- *Bss* (block starting symbol)
 - Contains zero-initialized data
 - Is read/write memory
- These sections are located in low memory, just after the section reserved for the OS kernel
- A picture. Its one of those columns with 7 rows. Top left says “low” and an arrow called PC pointing to the 2nd row. 1st row is OS, 2nd is text, 3rd is data, 4th is bss, (2,3,4 are all in one row separated in 3 sub-rows), 5th is heap with an arrow pointing down, 6th is free memory, and last is 7th called stack with arrow pointing up. Bottom left says ‘high’, SP points between free memory and stack, FP points to stack
- Pseudo-ops are used to indicate that what follows goes into a particular section
 - `.text`
 - Is the default section when assembling
 - `.data` anything following this goes into that section
 - `.bss` anything following this goes into that section
- The assembler uses a location counter for each section
 - Starts at 0, and increases as instructions and data are processed
 - The final step of assembly gathers all code and data into the appropriate sections
- When the OS loads the program into RAM
 - The text and data sections are loaded first
 - The bss section is then zeroed

External variables:

- How we implement static global or global variables
- Are non-local variables, allocated in the data (usually data) or bss section
 - Are used to implement C language *global* and *static local variables*
- Can be allocated and initialized using the pseudo-ops: `.dword .word .hword .byte`
 - General form:

```
Label:pseudo-op    value1[, value2, . . .]
```

// Anything in square brackets is optional
- Eg:

```
        .data
a_m:    .hword 23
b_m:    .word  (11 * 4) - 2
c_m:    .dword 0
ar_m:   .byte 10, 20, 30    // array, size 3
```

 - Allocates 17 bytes in data section:
 - `A_m = 23, b_m = 42, c_m = 0, ar_m = 10, 20, 30`
- The labels represent 64-bit addresses
 - Use `adrp` and `add` to put the address into a register
 - Then use `ldr` or `str` to access the variable
 - Eg: c code:

```
int i = 2, j = 12, k = 0;
int main() {
    K = i+j;           // k = 12 + 2
```
 - Eg: asm

```
        .data
i_m:    .word 2
j_m:    .word 12
k_m:    .word 0

        .text
```

```
        .balign 4
        .global main
Main: stp    x29, x30, [sp, -16]!
      mov    x29, sp
      adrp   x19, i_m
      add    x19, x19, :lo12:i_m      //low 12 bits of address i_m
      ldr     w20, [x19]              // w20 = i
      adrp   x19, j_m
      add    x19, x19, :lo12:j_m
      ldr     x21, [x19]              // w21 = j
      add    w22, w20, w21            // w22 = i + j
      adrp   x19, k_m
      add    x19, x19, :lo12:k_m
      str     w22, [x19]              // k = w22
```

- Uninitialized space can be allocated with the .skip pseudo-op

- Eg: 10 element int array

```
myarray: .skip 10 * 4
```

- Use .global, if the variable is to be made available to other compilation units

- Eg:

```
        .global      myvar_m
myvar_m: .word 13      // 13 is just initial value
```

- The bss section usually only uses the .skip pseudo-op

- All bss memory is zeroed before program execution

- Initializing memory to non-zero values (with .word, .hword, etc.) doesn't make sense

- Eg:

```
        .bss
array_m: .skip 10 * 4      // int array
c_m:     .skip 1           // char
h_m:     .skip 2           // short int = half word = 2
bytes
```

- Programmer-initialized constants are put into the text section

- Must be before or between functions

- Eg:

```
        .text
        .balign 4
func1: stp    ...
      ...
      ret
const_m: .hword42          // cannot be overwritten
        .balign 4          // stp must be wrd align
Func2: stp    ...
      ...
      ret
```

The ASCII Character Set

- American standard code for information interchange

Dec	Hx	Oct	Char	Dec	Hx	Oct	Htmi	Chr	Dec	Hx	Oct	Htmi	Chr	Dec	Hx	Oct	Htmi	Chr
0	0	000	NUL	(null)	32	20	040	Space	64	40	100	64	4	96	60	140	96	6
1	1	001	SOH	(start of heading)	33	21	041	!	65	41	101	65	5	97	61	141	97	7
2	2	002	STX	(start of text)	34	22	042	"	66	42	102	66	6	98	62	142	98	8
3	3	003	ETX	(end of text)	35	23	043	#	67	43	103	67	7	99	63	143	99	9
4	4	004	EOT	(end of transmission)	36	24	044	\$	68	44	104	68	8	100	64	144	100	10
5	5	005	ENQ	(enquiry)	37	25	045	%	69	45	105	69	9	101	65	145	101	11
6	6	006	ACK	(acknowledge)	38	26	046	&	70	46	106	70	10	102	66	146	102	12
7	7	007	BEL	(bell)	39	27	047	'	71	47	107	71	11	103	67	147	103	13
8	8	010	BS	(backspace)	40	28	050	(72	48	110	72	12	104	68	150	104	14
9	9	011	TAB	(horizontal tab)	41	29	051	{	73	49	111	73	13	105	69	151	105	15
10	A	012	LF	(NL line feed, new line)	42	2A	052	*	74	4A	112	74	14	106	6A	152	106	16
11	B	013	VT	(vertical tab)	43	2B	053	+	75	4B	113	75	15	107	6B	153	107	17
12	C	014	FF	(NP form feed, new page)	44	2C	054	,	76	4C	114	76	16	108	6C	154	108	18
13	D	015	CR	(carriage return)	45	2D	055	-	77	4D	115	77	17	109	6D	155	109	19
14	E	016	SO	(shift out)	46	2E	056	.	78	4E	116	78	18	110	6E	156	110	20
15	F	017	SI	(shift in)	47	2F	057	/	79	4F	117	79	19	111	6F	157	111	21
16	10	020	DLE	(data link escape)	48	30	060	0	80	50	120	80	10	112	70	160	112	22
17	11	021	DC1	(device control 1)	49	31	061	1	81	51	121	81	11	113	71	161	113	23
18	12	022	DC2	(device control 2)	50	32	062	2	82	52	122	82	12	114	72	162	114	24
19	13	023	DC3	(device control 3)	51	33	063	3	83	53	123	83	13	115	73	163	115	25
20	14	024	DC4	(device control 4)	52	34	064	4	84	54	124	84	14	116	74	164	116	26
21	15	025	NAK	(negative acknowledge)	53	35	065	5	85	55	125	85	15	117	75	165	117	27
22	16	026	SYN	(synchronous idle)	54	36	066	6	86	56	126	86	16	118	76	166	118	28
23	17	027	ETB	(end of trans. block)	55	37	067	7	87	57	127	87	17	119	77	167	119	29
24	18	030	CAN	(cancel)	56	38	070	8	88	58	130	88	18	120	78	170	120	30
25	19	031	EM	(end of medium)	57	39	071	9	89	59	131	89	19	121	79	171	121	31
26	1A	032	SUB	(substitute)	58	3A	072	:	90	5A	132	90	20	122	7A	172	122	32
27	1B	033	ESC	(escape)	59	3B	073	;	91	5B	133	91	21	123	7B	173	123	33
28	1C	034	FS	(file separator)	60	3C	074	<	92	5C	134	92	22	124	7C	174	124	34
29	1D	035	GS	(group separator)	61	3D	075	=	93	5D	135	93	23	125	7D	175	125	35
30	1E	036	RS	(record separator)	62	3E	076	>	94	5E	136	94	24	126	7E	176	126	36
31	1F	037	US	(unit separator)	63	3F	077	?	95	5F	137	95	25	127	7F	177	127	37

- Encodes characters using 7 bits, stored in a byte:
- 128 characters listed here.
- In assembly, character constants can be denoted with:
 - The hex code:
 - Eg: `mov w19, 0x5A`
 - The character in single quotes
 - Eg: `mov w19, 'Z'`
 - Note: may interfere with m4
 - Note: in gdb, use `p/c $w19` to print register contents as a character (say you wanted to print out the actual number in a register)

Creating and addressing string literals:

- A string is an array of characters
- Count be initialized in memory one byte at a time
 - Eg: want to make the string "cheers"


```
.byte 'c', 'h', 'e', 'e', 'r', 's' // allocates single byte for each letter (6 total allocated)
```
- But the .ascii pseudo-op is more convenient
 - Eg: `.ascii "cheers"`
- In C, strings are null terminated
 - Could be done using two pseudo-ops:


```
.ascii "cheers"
.byte 0
```
 - But more conveniently with .asciz or .string:


```
.string "cheers" // 7 bytes
```
- A **string literal** is a read-only array of characters. Allocated in the text section
 - In c code, is delimited with `"..."`
 - Eg:


```
int main() {
    printf("Hello, world!\n"); //String literal inside " "
}
```
- The literal usually has a label, which represents the address of the first character in the array
 - This address can be passed as a pointer argument into a function using an x register
 - Eg: equivalent assembly code:


```
.text
fmt: .string "Hello, world\n" // string literal
.balign 4 // evenly divisible by 4
```

```

        .global main
main: stp    x29, x30, [sp, -16]!
      mov    x29, sp

      adrp   x0, fmt                // address of 'H'
      add    x0, x0, :lol2:fmt      // now in x0
      bl     printf
      . . .

```

External arrays of pointers

- Are created with a list of labels
- Eg: c code

```

#include <stdio.h>
// Array of pointers to string literals
char *season[4] = {"spring", "summer", "fall", "winter"};
int main() {
    register int i
    for (i = 0; i < 4; i++) {
        printf("Season[%d] = %s\n", i, season[i]);
    }
    return 0;
}

```

- Assembly code:

```

define(i_r, w19)
define(base_r, x20)
        .text
fmt:      .string      "season[%d] = %s\n"

        spr_m:.string   "spring"
        sum_m:.string   "summer"
        fal_m:.string   "fall"
        win_m:.string   "winter"

        .data          //create array of pointers
        .balign 8      //must be double word aligned because they are
                        pointers
season_m:  .dword spr_m, sum_m, fal_m, win_m //references are always 8 bytes

        .text
        .balign 4
        .global main
main:     stp x29, x30, [sp, -16]
        mov x29, sp

        mov i_r, 0
        b test
top:      adrp x0, fmt
        add x0, x0, :lol2:fmt
        mov w1, i_r                //setting up 1st arg

        adrp base_r, season_m      //calculate array base address

```

```
        add base_r, base_r, :lol12:season_m

        ldr x2, [base_r, i_r, SXTW 3] //set up 3rd arg
        bl printf

test:    add i_r, i_r, 1
        cmp i_r, 4
        b.lt top

        ldp x29, x30, [sp], 16
        ret
```

Command-Line Arguments:

- Allow you to pass values from the shell into your program
- In C: `main(int argc, char *argv[])`
 - `argc`: the number of arguments (counted from `argv[]`)
 - `argv[]`: an array of pointers to the arguments (represented as a string)
 - C code:

```
int main(int argc, char *argv[])
{
    register int i;

    for(i=0; i<argc, i++){
        printf("%s\n", argv[i]);
    }
    return 0
}
```

Samples Run:

```
prompt> ./myecho one two
./myecho
one
two
prompt>
```

Assembly Code:

- Note: `argc` in `w0`, and `argv[]` is in `x1` for `main()`

```
define(i_r, w19)
define(argc_r, w20)
define(argv_r, x21)

fmt: .string "%s\n"

        .balign 4
```

Well meme'd, my men

```
.global main
main: stp x29, x30, [sp, -16]!
      mov x29, sp
      mov argc_r, w0          //copy argc
      mov argv_r, x1          //copy argv
      mov i_r, 0
      b test
top:   adrp x0,fmt              //set up 1st arg
      add x0, x0, :lol2:fmt

      ldr x1, [argv_r, i_r, SXTW 3] //set up 2nd arg
      bl printf

      add i_r, i_r, 1
test:  cmp i_r, argc_r
      b.lt top

      ldp x29, x30, [sp], 16
      ret
```

Separate Compilation:

- Source code is often divided into several *modules*(compilation units)
 - ie. separate .c or .s files
 - Makes development of large projects easier
- Modules can be compiled into *relocatable object code*
 - Will be put into corresponding .o files
 - E.g: gcc -c myfile1.c produced myfile1.o
- Object code is linked together to create an executable
 - Is done by *ld*(loader), which is usually invoked by gcc as the final step in the compilation process
 - E.g: gcc myfile1.o myfile2.o -o myexec
 - *ld* resolves any references to external data or functions in other modules
 - Code and data are relocated in memory as necessary to form contiguous text, data, and bss sections

Separate Compilation and Linking:

- Assembly code: first.s

```
.balign 4
.global main
main: stp x29, x30, [sp, -16]!
      mov x29, sp
      adrp x19, a_m ← refers to objects in other modules

      ldr w0, [x19]
      bl myfunc ← refers to objects in other modules

      ldp x29, x30, 16
      ret
```

- Assembly code: second.s

```
.global a_m
a_m: .word 44          //global constant
```

```
.balign 4
.global myfunc
myfunc: stp x29, x30, [sp, -16]!
        mov x29, sp
        sub w0, w0, 1
        ldp x29, x30, [sp], 16
        ret
```

- To Assemble and link:

```
as first.s -o first.o
as second.s -o second.o
gcc first.o second.o -o myexec
```

- Note: this can be done using a *makefile*

C code can call functions written in assembly

- Can be useful when optimizing sections of code

- E.g

- C code: mymain.c

```
#include <stdio.h>
int sum (int, int); //function prototype: need to define for assembler to know
```

```
int main()
{
    int i=5, j=10, result;
    result = sum(i, j);
    printf("result = %d\n", result);
    return 0;
}
```

- Assembly code: sum.s

```
.balign4
.global sum

sum:  stp x29, x30, [sp, -16]!
      mov x29, sp

      add w0, w0, w1

      ldp x29, x30, [sp], 16
      ret
```

- To compile and link:

```
gcc -c mymain.c
as sum.s -o sum.o
gcc mymain.o sum.o -o myprog
```

- To execute:

```
./myprog
result = 15
```

Assembly code can call functions written in C

- E.g:

- C code: sum.c

```
int sum(int a, int b){
    return a+b
}
```

- Assembly code: mymain.s

```
i_size= 4
j_size= 4
result_size = 4
alloc = -(16+i_size+j_size_result_size)&-16
dealloc = -alloc
i_s = 16
j_s = 20
result_m = 24          //starting addresses

fmt.  .string "result = %d\n"
      .balign 4
main: stp x29, x30, [sp, alloc]!
      mov x29, sp

      mov w19, 5
      str w19, [x29, i_s]
      mov w19, 10
      str w19, [x29, j_s]

      ldr w0, [x29, i_s]
      ldr w1, [x29, j_s]
      bl sum      <- function defined in sum.c

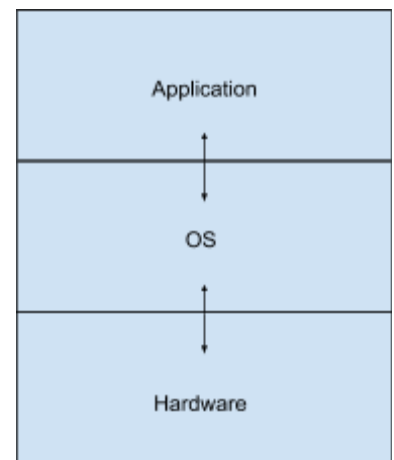
      mov w1, w0
      adrp x0, fmt
      add x0, x0, :lol2:fmt
      bl printf <- defined in the library

      ldp x29, x30, [sp], dealloc
      ret
- To compile and link:
gcc -c sum.c
as mymain.s -o mymain.o
gcc mymain.o sum.o -o myprog
- To execute:
./myprog
result = 15
```

Input and Output:

- A computer may do input and output in a variety of ways:
 - Interrupt-Driven I/O
 - Memory-mapped I/O
 - Port I/O
 - System I/O
- Since our ARM servers are running a linux OS, only system I/O is available
 - user -level programs communicates with external devices through the OS
 - Prevents malicious or accidental interactions that may damage the device

System I/O



- A program running in exception level 0 (EL0) does i/o by making a *system call*
 - I.e. it generates an *exception* using the `svc` instruction
 - Like a subroutine all
 - Control is transferred to a predefined system function
 - The address for this is stored in a table in the OS
 - The system code executes at exception level 1 (EL1)
 - Is able to interact directly with I/O devices
 - Once finished, control returns to the calling code
 - Changes back to EL0
- The type of system call is determined by the number put into `x8` before invoking `svc`
 - i/o service requests:

x8	Service request
56	opeanat (open file or open connection to device)
57	close
63	read
64	write

Note: are listed in a header file in `/usr/include/asm-generic/unistd.h`

- Eg:

```
mov    x8, 57
svc    0
```
- Arguments to system calls are put into registers `x0-x5`
- Any return value is in `x0`
- In unix (linux), all peripheral devices are represented as *files*
 - Provides a uniform interface for I/O
 - Typical pattern:
 - Open the file
 - I.e. connect to the device, get its *file descriptor*
 - Read from or write to the file
 - I.e. do device i/o, transferring bytes
 - Close the file
 - I.e. disconnect from the device
 - File i/o involves interacting with secondary memory (usually a disk or tape device)
 - Standard i/o involves the keyboard and screen devices

File I/O:

- Opening a file:
 - Equivalent C function:

```
int fd = openat(int dirfd, const char *pathname, int flags, mode_t mode) ;
```
 - Parameters in c code:
 - *dirfd*: directory file descriptor
 - Is used only if *pathname* is a relative path
 - Can be set to `AT_FDCWD` (the value -100) to indicate that the *pathname* is relative to the program's current working directory
 - *pathname*: absolute or relative pathname of a file
 - *flags*: combination of constants (using `|`) indicating what will be done to the file's data
 - Choose only one of:

- | | | | |
|--|----------|----|-------------------|
| | O_RDONLY | 00 | Read-only access |
| | O_WRONLY | 01 | Write-only access |
| | O_RDWR | 02 | Read/write access |
- Optional flags:

	O_CREAT	0100	Create file if it doesn't exist
	O_EXCL	0200	Fail if file exists *
	O_TRUNC	01000	Truncate an existing file gets rid of **
	O_APPEND	02000	Append access

* - with o_creat at same time
 ** - gets rid of anything in the file
 Note: flags are defined in /usr/include/bits/fcntl-linux.h
 - *mode*: optional arguments that specified UNIX file permissions
 - (There's permission for users, groups, and others. For each of these we can specify read, write, and execute. Shortened as 'RWX' permissions. Check out: <https://www.freebsd.org/doc/handbook/permissions.html>)
 - Required only when creating a new file (using O_CREAT)
 - Specified in octal
 - Eg: 0700 specifies read/write/execute permission for file owner, no permissions for group and others
 - *fd*: the returned file descriptor
 - Is -1 on error? (if it is, it didnt open the file)
 - Eg: opening an existing file called myfile.bin


```
pn:      .string      "myfile.bin"    //pn = pathname
...
mov      w0, -100      // 1st arg (use cwd)
adrp     x1, pn        // 2nd arg (pathname)
add      x1, x1, :lo12:pn
mov      w2, 0         // 3rd arg (read only)
mov      w3, 0         // 4th arg (not used)
mov      x8, 56        // openat I/O request
svc      0             // call system function
cmp      w0, 0         // error check
b.ge     open_ok
...
open_ok:
...
// error handling code
// fd is in w0
```

Read from a file

- Equivalent C function: `long n_read = read(int fd, void *buf, unsigned long n)`
 - *fd*: file descriptor: previously set by openat(), or 0 for stdin
 - *buf*: buffer where the bytes read will be stored - usually a local variable
 - *n*: number of bytes read (must be <= buffer size)
 - *n_read*: number of bytes actually read: -1 is returned on error
 - Assembly Code:

```
buf_size = 8
alloc = -(16+buf_size) &-16
dealloc = -alloc
buf_s = 16
...
main: stp x29, x30, [sp, alloc]!
      stp x29, sp
```



```
...                //open a file
...                //put fd in w19
mov w0, w19        //1st arg(fd)
add x1, x29, buf_s //2nd arg(pointer to buf)
mov x2, 8          //3rd arg(n)
mov x8, 63         //read I/O request aka code: 63
svc 0              //call sys function
```

Writing to a file

- Equivalent C function: `long n_written = write(int fd, void *buf, unsigned long n)`
 - *fd*: file descriptor: previously set by `openat()`, or 1 for `stdout`
 - *buf*: buffer where the bytes to write will be stored - usually a local variable
 - *n*: number of bytes to write (must be \leq buffer size)
 - *n_written*: number of bytes actually written: -1 is returned on error
 - Assembly Code:

```
buf_size = 8
alloc = -(16+buf_size) &-16
dealloc = -alloc
buf_s = 16
...
main: stp x29, x30, [sp, alloc]!
      mov x29, sp
      ...                //open a file
      ...                //put fd in w19
      ...                //fill buf with data
mov w0, w19        //1st arg(fd)
add x1, x29, buf_s //2nd arg(pointer to the beginning of the buffer)
mov x2, 8          //3rd arg(n)
mov x8, 64         //write I/O request aka code: 64
svc 0              //call sys function
...                //n_written is in x0
```

Closing a file:

- Equivalent C function:
 - `int status = close(int fd)`
- *fd*: file descriptor
- *status*: 0 if successful, -1 if failed
- Assembly Code:

```
mov w0, w19
mov x8, 57
svc 0
...
```

Standard I/O:

- Are always available
 - i.e are not opened or closed
- Standard input (*stdin*)
 - Represents input coming from the keyboard
 - File descriptor: 0
- Standard output (*stdout*)
 - Represents output to monitor screen
 - File descriptor: 1
- Standard error(*stderr*)

- Also represents output to the monitor screen
 - Normally used to output error messages
- File descriptor: 2
- C Code for copy stdin to stdout:

```
#include<unistd.h>
#define BUFSIZE 32

int main()
{
    char buf[BUFSIZE];          //byte array
    register int n;

    while((n = read(n, buf, BUFSIZE))>0) //0 =stdin
        write(1, buf, n);           //1= stdout

    return 0;
}
```

Assembly:

```
    buf_size = 32
    alloc = -(16+buf_size) & -16
    dealloc = -alloc
    buf_s = 16
    ...
main: stp x29, x30, [sp, alloc]!
      mov x29, sp

      add x1, x29, buf_s          //2nd arg ( pointer to buffer )

top:  mov w0, 0                   //1st arg(stdin)
      mov x2, buf_size           //3rd arg(BUFSIZE)
      mov x8, 63                 //read I/O request
      svc 0

      cmp x0, 0                  //Exit loop if
      b.le exit                  //n_read<=0

      mov x2, x0                 //3rd arg (n)
      mov w0, 1                  //1st arg (stdout)
      mov x8, 64                 //write I/O request
      svc 0                      //call sys function
      b top

exit: ldp x29, x30, [sp], dealloc
      ret
```

Floating point-numbers:

Introduction:

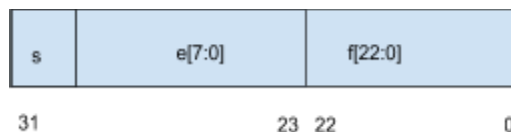
- Most modern architectures support floating point representation for fractional quantities
 - Usually implement the IEEE 754 standard
- Most CPUs now include floating-point units (FPUs)
 - Have instructions to do floating-point arithmetic
 - Very quick
 - If missing, FPU operation is simulated in software
 - Very slow
- A floating point number stored in a fixed size register may only *approximate* a real value
 - Beware: error may accumulate when doing repeated FP calculations

Fixed-point Numbers

- With integers, the binary point is assumed to be to the right of the LSB
 - Eg: 4-bit register
0101.
 - Given the value 5.0:
 $0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5$
 $0 + 4 + 0 + 1 = 5$
- Fractional quantities can be represented by moving the binary point to a new position
 - I.e. scaling the integer by a power of 2
 - Eg: 01.11
 - Given the value 1.75:
 $0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 1.75$
 $0 + 1 + 0.5 + 0.25 = 1.75$
- The position of the point is established *by convention*
- Fixed point is used for the *significand* in floating point representations

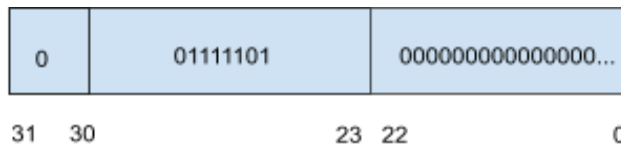
Floating-Point Single Format

- Is analogous to scientific notation
 - The differences are:
 - Base 2 instead of base 10
 - The *significand* (*mantissa*) and *exponent* are in binary
 - The number is *normalized* so that: $1.0 \leq \text{significand} < 2.0$
 - The significand is stored in fixed point format
 - Since the MSB is always 1, it is not stored
 - Provides one more bit of precision
 - The exponent is *biased* by adding the constant 127
 - Ensures it will always be stored as a positive integer
 - Uses a sign bit
- IEEE 754 Standard:



- Uses 4 bytes
- Number represented is: $N = (-1)^s * 1.f * 2^{(e-127)}$
 - s: sign bit (0 or 1)
 - e: biased exponent
 - Created by adding 127 (0x7f) to the unbiased exponent
 - f: fractional part of the significand
- Largest biased exponent allowed is 254 (0xfe)

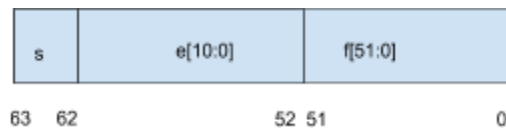
- 255 (0xff) is reserved, used to represent quantities that are not numbers (so-called NaNs)
- Thus, the largest possible unbiased exponent is 127
- The smallest biased exponent allowed is 1
 - 0 is reserved, used for *subnormal numbers* (tiny fractional quantities)
 - Thus, the smallest possible unbiased exponent is -126
- Range of magnitudes: $1.0 * 2^{-126}$ to $(2.0 - \epsilon) * 2^{127}$
 - $1.17549435e-38$ to $3.40282346e+38$
- Total range:
 - $-3.4028234e+38$ to $-1.17549435e-38$
 - Gap to 0.0 gap to (subnormal numbers)
 - $+1.17549435e-38$ to $+3.4028234e+38$
- Has about 7 decimal digits of precision
- Eg: 0x3e800000



- f is .0
- e is 125
- s is 0
- $(-1)^0 * (1.0) * (2^{125-127}) = 1 * 1.0 * 2^{-2} = +0.25$

Floating-Point Double Format

- IEEE 754 Standard:



- Uses 8 bytes
- Number represented is: $N = (-1)^s * (1.f) * (2^{e-1023})$
 - s: sign bit
 - e: biased exponent
 - Created by adding 1023 (0x3ff) to the unbiased exponent
 - f: fractional part of the significand
- Range of magnitudes: $\sim 2.2e-308$ to $\sim 1.8e+308$
- Has about 17 decimal digits of precision
- NaNs are represented using a biased exponent of 2047 (0x7ff)

Floating Point NaNs

- A NaN ("Not a Number") is an entity that cannot be represented using conventional numbers
- in single format, NaNs use a biased exponent of 0xff
 - positive (infinity): 0xf800000
 - negative (infinity) 0xff800000
 - sq(-1): 0x7fc00000
- In double format, NaNs use a biased exponent of 0x7ff
 - positive (infinity): 0x7ff00000 00000000
 - negative (infinity): 0xfff00000 00000000

- `sq(-1)`: `0x7ff8000000000000` //Manzara mentioned to Chris that infinity should yield the same thing as well
- // He also mentioned that all complex numbers yield `0x7ff8000000000000` in double format, might wanna google this though.
- Result from dividing by 0 or taking the square root of a negative number
- Using a NaN as an operand to most instructions causes an exception
 - However, $\pm\infty$ can be compared to a conventional number using `fcmp`
- May also be used as a valid argument to some functions
Eg: `atan(∞)` returns $\Pi/2$
- Armv8 has 32 128-bit FP registers
 - S registers use the low-order 32 bits to hold single precision FP numbers:
s0 - s31
 - D registers use the low-order 64 bits to hold double precision FP numbers:
d0 - d31
 - The high-order 64 bits are only used when using SIMD instructions
 - Not covered in this course
- These registers are loaded or stored like the general-purpose registers
 - Eg: `ldr s0, [base_r, offset_r]` //read 4 bytes
 - Eg: `str d1, [x29, 16]` //write 8 bytes
- Use the `.single` or `.double` pseudo-ops to allocate and initialize memory for a FP number
 - Use the prefix `0r` to specify a “real” value
 - Can specify exponents using E notation

-Eg:

```
.data
a_m:    .single    0r5.0 //set aside 4 bytes and come up with a certain bit pattern and put that bit
pattern in 4 bytes
b_m:    .double    0r5.33e-18
array_m: .single    0r2.5, 0r3.5, 0r4.5 //set aside 4 bytes for each of the elements
listed thus 12 bytes set aside
```

-In gdb:

-Use `p/f` to print the contents of a FP register

-Eg: `p/f $d0`

-Eg: `p/f $s30`

-Use `x/wf` to examine a single in memory

-Eg: `x/wf <a_m address>` shows 5

-`x/wx` shows the hex representation

-Eg: shows `0x40a00000`

-Use `x/gf` or `x/gx` to examine a double in memory

-Eg: `x/gf <b_m address>` shows
`5.3300000000000001e-18`

Basic Floating-Point Instructions

-Arithmetic instructions

-Use registers only (no immediates)

-Have two versions:

- Single precision: use S registers
- Double precision: use D registers

Note: you cannot mix singles with a double when doing arithmetic calculations

-Addition

-fadd Sd, Sn, Sm
 $Sd = Sn + Sm$

-fadd Dd, Dn, Dm
 $Dd = Dn + Dm$

-Subtraction

-fsub Sd, Sn, Sm
 $Sd = Sn - Sm$

-fsub Dd, Dn, Dm
 $Dd = Dn - Dm$

-Multiplication

-fmul Sd, Sn, Sm
 $Sd = Sn * Sm$

-fmul Dd, Dn, Dm
 $Dd = Dn * Dm$

-Multiply-negate

-fmul Sd, Sn, Sm
 $Sd = -(Sn * Sm)$

-fmul Dd, Dn, Dm
 $Dd = -(Dn * Dm)$

-Division

-fdiv Sd, Sm, Sn
 $Sd = Sn / Sm$

-fdiv Dd, Dm, Dn
 $Dd = Dm / Dn$

-Multiply-add

-fmadd Sd, Sn, Sm, Sa
 $Sd = Sa + (Sn * Sm)$

-fmadd Dd, Dn, Dm, Da
 $Dd = Da + (Dn * Dm)$

-Multiply-subtract

-fmsub Sd, Sn, Sm, Sa
 $Sd = Sa - (Sn * Sm)$

-fmsub Dd, Dn, Dm, Da
 $Dd = Da - (Dn * Dm)$

-Absolute value

-fabs Sd, Sn
- Sd = abs(Sn)
-fabs Dd, Dn
- Dd = abs(Dn)

-Negation

-fneg Sd, Sn
- Sd = -Sn

-fneg Dd, Dn
- Dd = -Dn

-Move instructions

-Register

-fmov Sd, Sn
-Moves 32 bits from Sn to Sd
-fmov Dd, Dn
-Moves 64 bits from Dn to Dd
-Variants exist for moving data between general-purpose and FP registers

-Immediate

-Can move a limited set of FP numbers into a register

-Form:

-fmov Sd, #*fpimm*
-fmov Dd, #*fpimm*

-#*fpimm*:

-Encoded with 1 sign bit, 4 bits of fraction, and a 3-bit exponent
-Must be expressible as: $\pm n / 16 \times 2^r$
-n in range: 16 to 31
-r in range: -3 to +4
-Eg: fmov s0, 0.25

- Conversion instructions

- fcvt Dd, Sn
 - Converts single-precision FP in Sn to double-precision FP in Dd
- fcvt Sd, Dn
 - Converts double in Dn to single in Sd, rounding as necessary
 - Precision may be lost since we use fewer bits to encode Sd
- fcvt_{ns} wd, sn // fcvt_{ns}: floating point convert nearest signed integer
- fcvt_{ns} xd, sn
 - Converts single in Sn to nearest signed 32-bit or 64-bit signed integer in Wd or Xd
- fcvt_{ns} wd, dn
- fcvt_{ns} xd, dn
 - Converts double to nearest signed 32-bit or 64-bit signed integer
- fcvt_{nu} converts to unsigned integer
- scvtf sd, wn
- scvtf sd, xn
 - Converts signed 32-bit or 64-bit signed integer in wn or xn to a single
 - You don't lose any data
- scvtf Dd, Wn // sign convert to float
- scvtf Dd, Xn
 - Converts signed integer to a double

- `ucvtf` converts unsigned integers to floats
- Compare instructions:
 - Forms:
 - `fcmp Sn, Sm` // Note, all singles
 - `fcmp Sn, 0.0` // Compare what is in Sn with 0.0
 - `fcmp Dn, Dm` // Variant: All doubles
 - `fcmp Dn, 0.0` // ^
 - Like the *integer* '`cmp`' instruction, these set the condition flags (NZCV)
 - Normally followed by a conditional branch
- eg: assembly code to divide 7.5 by 2.0


```

.data
x_m: .double    0r7.5      // 8 bytes
y_m: .double    0r2.0      // 8 bytes
z_m: .double    0r0.0      // 8 bytes

.text
.balign 4
main: stp    x29, x30, [sp, -16]!
      mov    x29, sp

      adrp   x19, x_m        // get address of x
      add    x19, x19, :lol2:x_m
      ldr    d0, [x19]       // Load x into d0

      adrp   x19, y_m        // get address of y
      add    x19, x19, :lol2:y_
      ldr    d1, [x19]       // load y into d1

      fdiv   d2, d0, d1      // d2 = x / y

      adrp   x19, z_m        // get address of z
      add    x19, x19, :lol2:z_m
      str    d2, [x19]       // Store result in z

end:   nop                  // breakpoint
      // Note: nop = no operation
      ldp    x29, x30, [sp], 16
      ret

```
- GDB output:


```

(gdb) x/15i main          // Note 15 = 15 cycles?>
Make a 'b end' then run it 'r'
p/f $d0 // should print out 7.5, the value of d0
p/f $d1 // should print out 2.0, the value of d1
p/f $d2 // should print out result, 3.75, the value of d2
x/gf <address of x> // Giant (8 bytes) Floating point number:
Prints out the value of x, 7.5
x/gf <address of y> // Giant Floating number: Prints out y, 2.0
(Note: purpose of this is to show that the address of x holds the
same value as the register in which we store it in, d0)

```

Floating-Point Arguments

- Are passed into a subroutine using registers d0-d7 (for doubles) and/or s0-s7 (for singles)
 - Are in addition to the x (w) registers used to pass in integers or pointers
 - Integer = w0, double = d0, float = s1, *char = x1
 - Stack memory is used if there are more than 8 FP arguments
 - The subroutine is free to overwrite these registers
- Registers d8 - d15 are *callee-saved registers*
 - Is used in a subroutine, the subroutine must save and restore their values on the stack
 - Only the bottom 64- bits of the 128- bit register need to be preserved
- Registers d0 - d7 and d16 - d31 can be overwritten by a subroutine
 - The caller is responsible for saving/restoring these if they need to be preserved over a subroutine call

Floating-Point Return Values

- A double FP number is returned from a subroutine in d0
- A single is returned in s0
- E.g.:

- C Code:

```
#include <stdio.h>
double power(double base, unsigned int exp)
{
    register double value;
    register unsigned int i;

    if(exp == 0)
        return 1.0;
    value = base;
    for (i =2; i <= exp; i++){
        Value *= base;
    }
    return value;
}

int main()
{
    register double result;

    result = power(5.5, 4);
    printf("result = %f\n", result);

    return 0;
}
```

- Assembly Code:
 - Note: base is in d0, exp in is w0, return value in d0

```
define(value_r, d16)
define(i_r, w19)
define(result_r, d17)

.text
fmt: .string "result = %f\n"
```

```

init_m:.double 5.5

        .balign 4
power:stp    x29, x30, [sp, -16]!
        mov    x29, sp

        cmp    w0, 0                //Check if exp == 0
        b.ne   next                //branch over if not

        fmov   d0, 1.0              //return 1.0 in d0
        b      exit

next: fmov   value_r, d0              //value = base

        mov    i_r, 2                //init i to 2
        b      test                //branch to loop test

top:  fmul    value_r, value_r, d0    // value *= base

        add    i_r, i_r, 1            //i++
test: cmp    i_r, w0                //loop while
        b.lsl  top                  // i <= exp

        fmov   d0, value_r            //return value

exit: ldp    x29, x30, [sp], 16
        ret

.global main
main: stp    x29, x30, [sp, -16]!
        mov    x29, sp

        adrp   x19, init_m            //get address of constant
        add    x19, x19, :lol2:init_m
        ldr    d0, [x19]              //1st argument (base)
        mov    w0, 4                  //2nd argument (exp)
        bl     power                  //call power() function
        fmov   result_r, d0           //store return value in result

        adrp   x0, fmt                //1st argument
        add    x0, x0, :lol2:fmt
        fmov   d0, result_r           //2nd argument
        bl     printf                 //call printf function

        mov    w0, 0
        ldp    x29, x30, [sp], 16
        ret

```

Readings and Exercise:

- P & H: Section 3.5
- ARM v8 Instruction set Overview
 - Section 5.7
- ARM Procedure Call Standard
 - Section 5.1.2

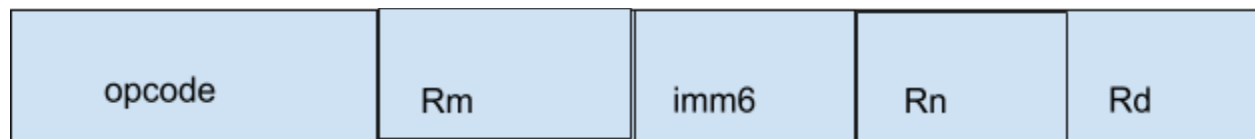
Machine Instructions

Introduction

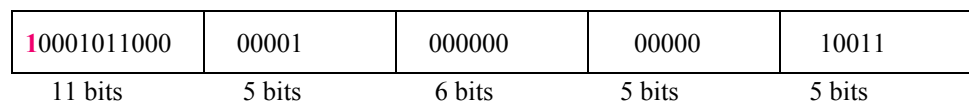
- All ARMv8 instructions are 32 bits in size
 - Encodes things such as:
 - The opcode
 - Eg: add, ldr
 - The destination and source registers
 - 1 bit: determines whether 32-bit or 64-bit registers
 - 5 bits for each register: determines register number
 - remember : $2^5 = 32$
 - An immediate value (if used)
 - An address (if used)
 - The ARMv8 has several different instruction formats
 - Each uses particular fields to encode the needed information
 - Some fields are further divided into *subfields*
- Form:

11 bits 5 bits 6 bits 5 bits 5 bits

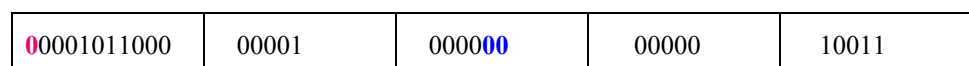
1. R-Type Format:



- Form:
 - *Opcode*: specifies the particular instructions to execute
 - *Rd*: destination register
 - 5 bits encode the unsigned integers 0-31
 - *Rn*: first source register
 - *Rm*: second source register
 - *imm6*: optional *shift amount*
 - Is 000000 if not used
- Eg: add x19, x0, x1
 - Machine instruction is: 0x8b010013
 - In binary: 1000 1011 0000 0001 0000 0000 0001 0011



- Eg: add w19, w0, w1
 - Machine instructions: 0x0b010013



- The left most bit in the *opcode* is the *size format (sf)* subfield
 - 0 indicates 32-bit registers
 - 1 indicates 64-bit registers
- Eg: `add w19, w0, w1, LSL 3`
 - Machine instruction is: 0x0b010c13
 -

00001011000	00001	000011	00000	10011
11 bits	5 bits	6 bits	5 bits	5 bits

2. I-Type Format

- Form:

<i>opcode</i>	<i>imm12</i>	<i>Rn</i>	<i>Rd</i>
---------------	--------------	-----------	-----------

- *opcode*: the particular instruction to execute
- *Rd*: destination register
- *Rn*: source register
- *imm12*: immediate value
 - Is a 12-bit unsigned integer
 - Range: 0 to 4095

- Eg: `add x19, x0, 4095`
 - Machine instruction: 0x113ffc13

0001000100	111111111111	00000	10011
10 bits	12 bits	5 bits	5 bits

3. D-Type Format

Form:

<i>opcode</i>	<i>imm9</i>	<i>op2</i>	<i>Rm</i>	<i>Rt</i>
11 bits	9 bits	2 bits	5 bits	5 bits

- *opcode*: instruction to execute
- *Rt*: source/destination register for store/load
- *Rn*: base register(always an x register)
- *imm9*: immediate value specifying the offset
 - Is a 9-bit signed integer
 - Range -256 to +255
- *op2*: set to 00 for unscaled loads/stores
- E.g `str x19, [x20, 5]`
 - Machine instruction: 0xf8005293

11111000000	000000101	00	10100	10011
-------------	-----------	----	-------	-------

4. B-Type Format

Form:

<i>opcode</i>	<i>imm26</i>
6 bits	26 bits

- *opcode*: the type of unconditional branch
 - b: 000101
 - bl: 100101
- *imm26*: 26-bit signed integer specifying the branch offset in words
 - Range in words: -2^{25} to $+2^{25}-1$
 - Range in bytes: -2^{27} to $+2^{27}-1$ ($\pm 128\text{MB}$)
- The offset is added to the current address in the PC register, to specify the address to jump to
 - Is called *PC-relative addressing*
 - Since absolute addresses are not used, the machine instructions are **relocatable** in memory
- Eg:


```
top:  add x1, x1, 1
      cmp:  x1, 10
      b.ge next
      b     top //branch instruction
next:  ...
```
- Machine instruction for b top: 0x17ffffd

000101	11 1111 1111 1111 1111 1111 1101
6 bits	26 bits

- The offset is -3 words(-12 bytes)

CBI-Type

- Form: opcode, imm19, 0, cond
 - 8 bits, 19 bits, 1 bit, 4 bits
 - *opcode*: 01010100 for conditional branch
 - *Imm19*: 19-bit signed integer specifying the branch offset in *words*
 - Range in words: -2^{18} to $+2^{18}-1$
 - Range in bytes: -2^{20} to $+2^{20}-1$ ($\pm 1\text{MB}$)
 - *Cond*: 4-bit code specifying the particular conditional branch instruction
- (Note: check out the ARM architecture reference manual: p. C1-122)
 - Eg: Asm code:


```
top:  add    x1, x1, 1
      cmp    x1, 10
      b.ge   next // cond. Branch
      b     top
next:  . . .
```
 - Machine instruction for b.ge next: 0x5400004a
 - 01010100, 0000000000000000010, 0, 1010
 - 8 bits, 19 bits, 1 bit, 4 bits

01010100	0000 0000 0000 010	0	1010
----------	--------------------	---	------

- The offset is +2 words (+8 bytes)
- Dank diagram by sim

What to expect for the final:

- Saturday Dec 17th @ 12:00 noon
- MFH 162

- 2 Hour long exam “Sex only lasts 15 minutes, getting fucked by manzara last 2 hours” - Sim

- Will provide reference information
- There may be 3 or 4 coding questions: (Half weight of exam) He will give you C code and ask you to translate it to the equivalent assembly code
- Make sure to follow convention for while loops
- 3 or 4 short answer questions
- Is cumulative, emphasis on 2nd half of course (post-midterm, and assignments 4, 5, & 6)
- Here's what to emphasize:
 - De-emphasize Macro Preprocessors
 - Emphasize ARMv8-A Architecture section
 - De-emphasize Introduction to the gdb debugger
 - OCTAL NUMBERS!
 - BITWISE OPERATIONS-
 - SOMEWHAT DE-EMPHASIZE sign and zero extend, and bitfield operations, only somewhat!
 - De-emphasize unsigned arithmetic, but only somewhat
 - De-emphasize division in binary arithmetic, only somewhat
- YOU NEED TO KNOW THE STACK INSIDE OUT
- MAKE SURE YOU KNOW HOW TO CREATE AND DEAL WITH LOCAL VARIABLES
- MEMORY ALIGNMENT IS IMPORTANT
- MAKE SURE YOU KNOW LOAD AND STORE AND HOW TO USE ASSEMBLER EQUATES
- MAKE SURE TO EXPECT A CODING QUESTION TO TEST ONE DIMENSIONAL ARRAYS!
- YOU SHOULD EXPECT A CODING QUESTION THAT TESTS NESTED STRUCTS
- SUBROUTINES IS A SUPER IMPORTANT CHAPTER! SUPER SUPER IMPORTANT
- MAKE SURE YOU UNDERSTAND LINK REGISTERS
- SAVING AND RESTORING REGISTER AND HOW TO DO THAT, YOU NEED TO KNOW THAT
- KNOW THE ARGUMENTS TO SUBROUTINE VERY WELL
- POINTER ARGUMENTS, RETURNING INTEGERS
- RETURNING STRUCTURES BY VALUE ! !!! IMPORTANT !!
- External data and text, creating string literals
- Maybe a coding question that covers External Array pointers and command line arguments
- Separate compilation ! There will definitely be a question on it!
- File I/O might be on a coding question
- De-emphasize Floating point NaNs
- Expect a coding question modeled on assignment 6, uses topics from floating point number section

Midterm - Cut off on data structures - nested structures. No coding on two dimension arrays and nested structures. Coding on one dimension array.