Lec 2019/05/06

Course Objectives:

1. Learn how a typical modern computer is structured.

   architecture, CPU

2. Learn how a computer operates as it executes instructions.

   fetch-execute cycle or fetch-decode-execute cycle

3. Learn how data and instructions are represented internally in a computer.

   signed / unsigned integers
   character and strings
   floating-point numbers
   machine instructions

4. Learn how to write programs in assembly language?

   - Often needed when creating:
     - Embedded system
     - OS Kernel
     - Device drivers
     - Code generator part of a compiler
     - Application (rare today)

5. The course is useful for helping you:

   - Understand a computer's arch
   - Understand the details of OS
   - Write more efficient high-level language
   - Learn C
   - Learn how to mix C and asm

High-level arch

1. A basic computer system consists of:

   CPU(Central Processing Unit)

   System Clock

   Primary Memory(RAM, also called Random Access Memory)

   Secondary memory(Hdd, also called Hard disk drive, or solid state drive)

   Peripheral I/O devices Keyboard, mouse, monitor, scanner, printers, joysticks ...

   Bus(Transit data/instructions between components)

2. CPU(Central Processing Unit):

   - is the brain of any computer system
   - Executes the instructions (program)
   - Controls the transfer of data across the bus(policy)
   - Usually contained on a single microprocessor chip (eg. Intel Core i5)
   - Consists of 3 main parts: CU(Control Unit), ALU(Arithmetic Logic Unit), Registers

     - CU: Directs the execution of instructions:

       - Loads an operation code(opcode) from primary memory into the instruction register(IR)
       - Decodes the opcode to identify the operation
       - If neccessary, transfers data between primary memory and register
       - If neccessary, directs the ALU to operate on data in registers
       - The rules change for registers' writing/overwriting based on the assembly language

     - ALU:

       - performs arithmetic and logical operations on data stored in registers(Eg: Add two numbers: regA: 5, regB: 2, regC: 5 + 2 = 7)
       - For logic operations, ALU would do the same thing to compare conditional cases, but using comparatives such as AND, OR, NOT

3. Registers:

   - Binary storage units within the CPU.
   - May contain: Data, Addresses, Instructions, Status information

- General-Purpose registers(GP): used by a programmer to temporarily hold data and addresses
- Program Counter(PC): A program counter is a register in a computer processor that contains the address (location) of the instruction being executed at the current time. As each instruction gets fetched, the program counter increases its stored value by 1 (point to the next address).
- The Status Register(SR): contains information(flags) about the result of a previous instruction.
  - Eg: Overflow or carry
  - 32-bit 0(flag) 1(flag) 2 ... 31
  - 64-bit 0 1 2 ... 63

4. System Clock:

   - Generates a clock signal to sync the CPU and other clocked devices:
     - is a square wave at a particular freq

   - Devices will coordinate on the rising and falling edges of the square wave, ***not*** on each wave's crest or though.
   - Examples of clock rates: iMac(2016) 3.2GHz, RPi(700MHz)

5. Primary Memory

   - Often called Random Access Memory(RAM)
   - Any byte in memory can be accesses directly if you know its address
   - Primary memory can stores bytes that can be read/written by accessing an address
   - is *volatile*: Data disappears when a computer is powered off
   - is used to store program instructions and program data(variables)
   - Consists of a sequence of addressable memory locations
     - Each location is typically 1 byte(8bits) long
     - Example sizes: iMac(8Gb), Rpi(256MB)
     - 1GB = 1024MB = 1024*1024KB = 1024*1024*1024Bytes = 1024*1024*1024*8Bits
     - 4bits is a nibble

   - In a von neumann arch, RAM contains both data and programs(instructions)

6. Bus:

   - is a set of parallel data/signal lines
   - is used to transfer information between computer components
   - often subdivided into *address bus*, *data bus* and *control bus*(Sorts of buses)

   - Address Bus: Specifies a memory location in RAM(Common or sometimes a memory-mapped IO device)

        - N sizes: 32 and 64bits(match the sizes in CPU)
        - Only transits one-way, from CPU to the Primary Memory

    - Data Bus:

        - Two ways between CPU and the Primary Memory

    - Control Bus: Control or monitor devices: Read/Write signal for RAM

        - Two ways between CPU and the Primary Memory

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

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

7. Secondary Memory:

    - is used to hold a computer's file system: stores files containing programs or data
    - is *non-volatile* read / write memory: its contents persist through a power cycle
    - usually embodied on a HDD/SSD: SSDs are becoming more common

    - Peripheral I/O Devices:

        - Allow communication between the computer and the external environment
        - Example input devices: Keyboard, Pointing devices, Microphone
        - Example output devices: Monitor, Printer, Speakers
        - Example input/output devices: Hard disk drive, Modem, Connections to networks, SSD

8. Basic CPU Arch:

    - Operands for an instruction come from the ACC(accumulator register) and from a single location in RAM
    - ALU results are always put into the ACC
    - The ACC can be loaded from or stored to RAM
    - Only *load* and *store* instruction can access RAM
    - Other instructions operate on specified registers in the *register file*, not on RAM(Since registers are more quickly accessed than RAM)

9. RISC & CISC

    - Both are instruction sets, the main difference between the two being that RISC is 'one-cycle', where CISC requires multiple cycles.

- A 'cycle' corresponds to each vertical edge of a system clock's generated square wave.

- RISC: Reduced Instruction Set Computer

    - Uses only simple instruction that can be executed in one machine cycle
    - Enables faster clock rates, thus faster overall execution
    - But makes programs larger, more complex
    - Multiplication done using repeated add-shift operations
    - Machine instructions are always the same size
    - Makes decoding simpler and faster
    - ARMv8 instructions are always 32-bit wide

- CISC: Complex Instruction Set Computer(Most modern computers)

    - May have instructions that make many cycles to execute
    - Are provided for programmer convenience
    - 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(imm):
    - Makes decoding difficult and slow
    - Intel x86: Can be as short as 1 byte long, but 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

        1. Fetch the next instruction from memory into the instruction register(IR), while PC register contains its address
        2. Increment Program Counter to point to the next instruction
        3. Decode the instruction
        4. If the instruction uses an operand in RAM, calculate its address
        5. Fetch the operand
        6. Execute the instruction
        7. If the instruction produces a result that is gonna store in RAM, then calculate its address and store the result

    - 4-7 repeat if neccessary

10. Asm Programs examples:

```
add x20, x20, x21 // x20 = x20 + x21

/* Each statement consists of an opcode and a variable number of operands */
/* add is opcode, x20, x21 are operands */
```

Corresponds to:
1000 1011 0001 0101 0000 0010 1001 0100
Or in hex:
0x8b150294

- Items are stored sequentially in memory.

```
/* _start is a label, can prefix any statement */
_start: add x20, x20, x21 // x20 = x20 + x21
```

- A label is a symbol whose value is the address of the machine instruction.
- May be used as a target for a branch instruction.

```
.global _start
/* .global is an assembler directive */
_start: add x20, x20, x21 // x20 = x20 + x21
```

- assembler directives do **not** generate machine instructions, but give the assembler extra information.

1. Assemblers:

    - Translate asm source code into machine code
    - In this course we will use the GNU assembler(as)
    - To assemble armv8 source code:

    ```
    m4 myprog.asm > myprog.s
    gcc myprog.s -o myprog
    #gcc calls assember 'as', then links the code, producing an executable called myp
    rog)
    ```

    - Many assemblers support macros, although GNU as has limited support for macros
    - Macro proprocessors example:

```
define(var_name, value)
define(coef, 23)
define(z_r, x18)
add x19, z_r, coef
```

- Use m4(macro processor) converts asm files to sources files, before calling gcc(Read Chapter1 on the textbook for more)

Lec 2019/05/08

ARMv8-A Architecture

The course uses Applied Micro X-Gene X-C1 servers

- Its CPU is APM883208-X1 X-Gene Multi-Core 64bit processor
- An implementation of the ARMv8-A
- specification(Advanced RISC Machine)
- OS is Linux

The ARMv8-A Arch

- Is a RISC
- Is a Load/Store machine
  - Register file contains 31 64-bit registers
  - Most instructions manipulate 64-bit or 32-bit data stored in these registers

- Has two execution states: AArch64, AArch32

  - AArch64
    - Uses the A64 instruction set and 64-bit registers
    - Used exclusively in this course

  - AArch32
    - Uses the A32 or T32 instruction sets
    - Provide for computability with the older ARM and THUMB instruction sets using 32-bit registers
    - ***Not used in this course***

```
add x, y, z // size of the instruction: 32(AArch32) / 64(AArch64)
```

- has *4 exception levels*:

- EL0: for normal user applications with limited privileges
  - Restricted access to a limited set of instructions and registers
  - Most programs work at this level

- EL1: for the OS Kernel
  - Accessed indirectly by user programs using *system calls*

- EL2: for a *Hypervisor*

  - Supports *virtualization*

- EL3: for low-level firmware
  - Includes the *Secure Monitor*

ARMv8 Registers

AArch64, has 31 64bit-wide General-Purpose registers (GP)

- Numbered from 0-30
- When using all 64bits, use 'x' or 'X' before the number(stands for extended: eg. x0, x30)
- When using only the low-order 32 bits of the registers, use 'w' or 'W'(word, eg. w2 W29)
- Many of these registers have special uses:

  - x0-x7 used to pass arguments into procedure(function), and return results

```
mov x0, 13
mov x1, 11
mov x2, 15
b sum
sum: /* definition of sum */
```

  - x8 indirect result location register, returning large things back to code
  - x9-x15 temporary registers
  - x16, x17 *intra-procedure-call* temporary registers(IP0, IP1)
  - x18 platform registers

```
mov x9, 100
mov x0, 13
mov x1, 10
mov x2, 15
// push x9's value to stack
b sum
// pop x9's value from stack
```

- x29 *frame pointer* (FP) register
- x30 procedure *link register* (LR)

- Use registers *x19-x28* for most of your work (for 'int' type, use *w19-w28*)
  - are *callee-saved* registers
    - Value is preserved by any function you want

- 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 runtime stack
  - Zero Register
    - xzr 64bit wide
    - wzr 32bit wide
    - Gives 0 value when read from
    - Discards value when written to
  - Program Counter
  - Program Status Register
  - Exception Link Register

```
mov xzr, 10 // wont work since 10 will be discarded
```

- Program Counter
  - PC: 64bits 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 loads/stores

  - Can be accessed by *gdb* by $pc

```
xyz: mov x22, 15
mov x19, 22
add x19, x19, x20
bl xyz
add x19, x19, x21
// xyz will be executed at least 1 time
```

```
mov x19, 22
add x19, x19, x20
bl xyz
add x19, x19, x21
xyz: mov x22, 15
// the count of execution of xyz depends on how many 'bl xyz'
```

AArch64, has 32 128-bit-wide *floating-point* registers (*not* FP register!)

- Has numerous *system registers*

  - Most are accessed in EL1.

A64 Assembly Language

- Consists of statements with **1** opcode and **0** to **4** operands

  - The allowed operands depend on the particular instruction.
  - In general:
    - The **1st** operand is a *destination register*.
    - The others are *source registers*.
    - Eg: add x19, x20, x21 // operand destination, source0, source1

  - An *immediate value* (a constant) may be used as the final source operand for some instructions.

    - Eg: add x19, x20, 42 // 42 is an immediate value
    - A # symbol can prefix the immediate, but is optional when using gcc
    - Eg: add x19, x20, #42 // 42 is an immediate value
    - The allowable range of constants depends on the particular instruction
      - Depends on the number of *available bits* within the machine instruction

    - Only **1** constant(imm value) is allowed in one instruction

- *Immediate values* are assumed to be decimal numbers unless *prefixed* a follows:
    - Hex: 0x
        - Eg: 0x6f
    - Octal: 0
        - Eg: 0777
    - Bin: 0b
        - Eg: 0b10
- 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 value(32-bit):

    ```
    mov Wd, #imm32      // imm32: -2^31 to 2^32 - 1
    mov Xd, #imm64      // imm64: -2^63 to 2^64 - 1
    ```

    - Move 32bit reg:

    ```
    mov Wd, Wm // alias orr Wd, wzr, Wm, equivalent to Wd = Wm
    ```

    - Move 64bit reg:

    ```
    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 function call
    - Return value is in *x0*

Basic Program Structure

- Code Template

```
.global main

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

        // your custom code goes here

        // Set up return value of zero from from main()
        mov  w0, 0

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

- .global main: Makes the label 'main' visible to the linker
  - The main() routine is where execution always start
- [sp, -16]!
  - allocates 16bytes in stack memory(in RAM)
  - does so by pre-incrementing the SP register by -16(size of 2 registers, x29, x30)
- stp x29, x30
  - store the contents of the pair of registers to the stack
  - x29: FP, x30: 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 from the current sp
  - fp may be used as a base address in the routine
- ldp x29, x30, [sp], 16
  - loads the pair of registers (x29, x30) from RAM
  - SP points to the location in RAM where we read from
  - Restores the state of the FP and LR registers
  - [sp], 16
    - Deallocate 16 bytes of stack memory

        - Does so by post-incrementing SP by +16

- ret
    - Returns control to calling code(in OS)
    - Uses the address in LR

Basic Arithmetic Instructions

- Addition
    - Uses 1 destination and 2 source operands
    - Register (64bit and 32bit):
        - Eg: add x19, x20, x21 // x19 = x20 + x21
        - Eg: add w19, w19, w20 // w19 = w19 + w20
    - Imm (64bit and 32bit):
        - Eg: add x20, x20, 1 // x20 = x20 + 1
        - Eg: add w27, w19, 4 // w27 = w19 + 4
        - Eg: add w27, w19, -4 // w27 = w19 - 4
- Subtraction
    - Uses 1 destination and 2 source operands
    - Register (64bit and 32bit):
        - Eg: sub x19, x20, x21 // x19 = x20 - x21
        - Eg: sub w19, w19, w20 // w19 = w19 - w20
    - Imm (64bit and 32bit):
        - Eg: sub x20, x20, 1 // x20 = x20 - 1
        - Eg: sub w27, w19, 4 // w27 = w19 - 4
- Multiplication
    - Uses 1 destination and 2 source operands
    - No imm allowed
    - From(32-bit): mul Wd, Wn, Wm
        - Calculate: Wd = Wn * Wm
        - Alias for: madd Wd, Wn, Wm, wzr
        - Eg: mul w0, w1, w2
    - The 64bit form is similar

        - Eg: mul x19, x20, x20 // x19 = x20 * x20

- Signed Multiply Long
    - smull Xd, Wn, Wm
    - Signed Multiply Long: Xd = Wn * Wm
- Multiply-Add
    - Cannot use imm
    - Form(32bit): madd Wd, Wn, Wm, Wa
        - Calculates: Wd = Wa + Wn * Wm
        - Eg: madd w20, w21, w22, w23 // w20 = w23 + w21 * w22
    - 64bit form is similar
        - Eg: madd x20, x0, x1, x20 // x20 = x20 + x0 * x1
- Multiply-Subtract
    - Form(32bit): msub Wd, Wn, Wm, Wa
        - Calculates: Wd = Wa - Wn * Wm
- Multiply-Negate
    - Form(32bit): mneg Wd, Wn, Wm // Wd = - (Wn * Wm)
- Other variants see ARM doc
- Divison
    - Uses 1 destination and 2 source operands
    - Immediate-values are **NOT** allowed
    - Signed form(32bit): sdiv Wd, Wn, Wm // all are signed numbers(0 will be handled as +0)
        - Operands are signed integers
        - Calculates: Wd = Wn / Wm
    - The *udiv* variants use unsigned integer operands:
        - Eg: udiv w0, w1, w2
    - These instructions do *integer division*
        - The calculated quotient is an integer, and any remainder is discarded
        - Eg: 14/3 is 4

        - To get the remainder:

```
mov x19, num
mov x20, quo
mov x21, den
msub x22, x20, x21, x19 // numerator - (quotient * denominator)
```

    - Dividing by 0 will **NOT** generate an exception(a trap), instead, it will write 0 to destination register.

- Print to standard output

    - Is done by calling printf()
    - Is a standard function in the C library
    - Invoked with 1 or more arguments

```
#include <stdio.h>
int main() {
    printf("Meaning of life = %d\n", 42);   // %d is a placeholder for signed int
eger
    return 0;
}
```

equvialent asm code:

```
fmt:    .string     "Meaning of life = %d\n"     // creates the format string
        .balign     4                            // ensure instructions are prope
rly aligned

.global main

main:
        adrp    x0, fmt              // set format string high bits
        add     x0, x0, :lo12:fmt   // set format string low 12 bits
        mov     w1, 42               // set 42 be the second argument of printf
        bl      printf               // call printf
```

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

    - Like a *goto* in the C language
    - PC register will not be incremented as usual, but is set to the computed address of the instruction, which means the value of the label

- Ab *Unconditonal branch* is always taken

- Form: bl *label*
- Eg: bl exampleLabel
- LR = address of instruction after bl
- PC = address of exampleLabel

- RET: return from subroutine

  - Eg: ret
  - PC = LR

- *Condition flags* may be used to store information about the result of an instruction

  - Are single-bit units in the CPU
    - Record *process state* (pstate) information
    - 0 means false, 1 means true

  - There are 4 flags:
    - Z: true if result is **z**ero
    - N: true if result is **n**egative
    - V: true if result is o**v**erflows
    - C: true if result generates a **c**arry out

- Condition flags are set by instructions that end in 's' (short for set flags)

  - Eg: subs, adds
  - *subs* may be used to compare two registers

    - Eg: subs x0, x1, x2

  - But **cmp** is more intuitive:

    - From(64bit): cmp Xn, Xm // alias for: subs xzr, Xn, Xm

- Condition flags instructions use the condition 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 label

  - Otherwise, control 'drops through' to the following instruction (means it will be ignored)
  - Eg: b.eq top // branch top if Z == 1 is true

```
mov x19, 10
mov x20, 12
cmp x19, x20 // Z == 0
b.eq xyz // wont branch since Z == 0

xyz: //....
```

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

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

```
define(x_r, x19)
        mov x_r, 1
top:
        // statements forming
        // loop body
        add x_r, x_r, 1
test:   cmp x_r, 10
        b.le top // branch if x_r <= 10
```

        - while: is a *pre-test* loop

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

```
define(x_r, x19)
        mov     x_r, 0

top:    cmp     x_r, 10
                b.ge done

        // statements forming
        // loop body
        add     x_r, x_r, 1
        b       top

done:   // other instructions
```

- Other example

```
int i = 0;
while(i < 20) {
    printf("i = %d\n", i);
    ++i;
}
```

```
/* loop_test at the bottom of loop_top */
            mov     x19, 0
            b loop_test
loop_top:
            ldr     x0, =printf_string  // "i = %d\n"
            mov     x1, x19             // set the 2nd argument to x19
            bl      printf              // printf("i = %d\n", x19);
            add     x19, x19, 1         // ++i
loop_test:
            cmp     x19, 20
            b.lt    loop_top
```

```
/* loop_test at the bottom of loop */
            mov     x19, 0
loop_test:
            cmp     x19, 20
            b.ge    done
loop:
            ldr     x0, =printf_string  // "i = %d\n"
            mov     x1, x19             // set the 2nd argument to x19
            bl      printf              // printf("i = %d\n", x19);
            add     x19, x19, 1         // ++i
            b       loop_test
done:   // other instructions
```

- for loop: can be converted to equvialent while loop, so it is also a *pre-test* loop

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

is the same as:

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

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

        mov     i_r, 10

loop:   cmp     i_r, 20
        b.ge    next
        add     x_r, x_r, i_r
        add     i_r, i_r, 1
        b       loop
next:
```

- if: is formed by branching over the statement body if the condition is *not* true
  - Must use the *logical complement*:

        - b.lt <----> b.ge
        - b.le <----> b.gt
        - b.eq <----> b.ne

    - Example C Code:

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

```
define(a_r, x19)
define(b_r, x20)
define(c_r, x21)
define(d_r, x22)

        cmp     a_r, b_r
        b.le    next

        add     c_r, a_r, b_r
        add     d_r, c_r, 5
:next
```

- if-else:

    - Example C Code:

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

```
define(a_r, x19)
define(b_r, x20)
define(c_r, x21)
define(d_r, x22)

        cmp     a_r, b_r
        b.le    else

        add     c_r, a_r, b_r
        add     d_r, c_r, 5

        b       next
:else
        sub     c_r, a_r, b_r
        sub     d_r, c_r, 5
:next
```

Introduction to the GDB debugger

- In order to preserve debugging info, use "gcc -g" to compile
- To start a prog under debugger control, use: gdb myprog
- To set a breakpoint, type: b label Eg: b main // or break *main
- Breakpoint at a specific line Eg: b line# (line number depends on *l*)
- Use *r* to run the prog: Will stop at the first breakpoint
- Use *r argv0 argv1 argv2* to run the prog with arguments, arguments will be used in char ** argv in C
- Use *l* list 10 lines of source code
- Use *l line_number* list 10 lines of source code around line_number
- Use *c* to continue to the next breakpoint
    - Or to the end of the program, if no other breakpoints
- Use *i r* to print the values in all registers
- Use *i r $reg_name* to print the value in reg_name
- Use *i b* to print all breakpoints
- Use *shell cmd* to execute commands in inner shell
- *info functions* Print functions in program
- *info stack* Print backtrace of the stack
- *info frame* Print information about the current stack frame
- Use *clear label_name* to clear the breakpoint at the label
- Use *clear line_number* to clear the breakpoint at the line#
- Use *delete* to clear all breakpoints

- Use *x* to print the address of current instruction
- Use *x/i* to print the address and contents of current instruction

  - *x/d $fp+16* Examine the (4-byte) word at the address which is 16 bytes from the frame pointer.
  - *x/g $fp+16* Examine the (8-byte) word at the address which is 16 bytes from the frame pointer.
  - *x/xg $fp+16* x/g fp+16 with hex output
  - *x/s 0xbfff08ff* Examine a string stored at 0xbfff08ff
  - *x/20i main* Examine 20 instructions of the main() function

- Use *kill* to stop execution of current running program

- To *Single step* through your program, use:

  - *si* // executes the next instruction, if a function is call, it will step into function calls
  - *si 4* // execute the next 4 instructions like *si*
  - *ni* // also executes the next instruction, but if a function is call, it will not step into function calls
  - *display/i $pc* to set GDB display info of PC after every command
  - *undisplay 1* undo display at slot1
  -

- Use *p $reg* to print the contents of a register:

  - Eg: p $x19
  - Can append a format character: p/x(hex), p/t(bin), p/d(dec), p/o(oct), p/a(address)

- Use *q* to quit the gdb

Difference between 'si' and 's' (or 'ni' and 'n')

- 'si' Execute one instruction
- 's' Execute one C statement

Other ARMv8 instructions

- ldr: load register (ldr Xn, *address*)

  - ldr x19, =label // load the address of label, x19 = address of label
  - ldr x19, [x20 + 4] // load the value at memory whose address is x20 + 4, x19 = *[x20 + 4]

- str: store register(str Xn, *address*)

  - str xn, [xd] // write the value in xn to memory [xd], *xd = xn

- sxtw: sign extension

```
int a;
long int b = (int)a;
```

```
sxtw    b, a // Sign extend product and put it in temp1
```

The Decimal Representation

- For the n-bits binary number:

  $$a_n a_{n-1} a_{n-2} \ldots a_2 a_1 a_0$$

  It's equivalent to the decimal number $\sum_{i=0}^{n-1} b_i 2^i$

Addition in Binary

$c_7 c_6 c_5 c_4 c_3 c_2 c_1 c_0$
$b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$
$a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0$

$$r_7 r_6 r_5 r_4 r_3 r_2 r_1 r_0$$

Where $r_i = a_i XOR b_i XOR c_i$

- How to optimize 'if (product & 0x1)'?

```
ANDS wzr, product, 0x1    // ANDS wzr, Wn, Wm
TST  product, 0x1         // TST is an alias for ANDS wzr, Wn, Wm.
                          // Thus in the case it means ANDS wzr, product, 0x1
b.eq  branchWhenProductEndsWithZero // ANDS results 0 when the bit-0 of product i
s 0
// or you can use b.ne branchWhenProductEndsWithOne
```

Instructions to read from and write to memory

- **str** store a register in memory
- **strh** store the lower 16-bits of a register in memory, a halfword
- **strb** store the lower 8-bits of a register in memory, a byte
- **ldr** load a register from memory
- **ldrh** load the lower 16-bits of a register in memory

- **ldrb** load the lower 8-bits of a register in memory
- **ldrsw** load a word from the memory, sign-extended it, and put into destination register
- **ldrsh ldrsb**

- Load immediate 32-bit values to a register using LDR Rd, =const

Assembly Equates (EQU) directive

- Similar to m4 macros or register equates
- Form:

```
define(SIZE, 1231)

.equ    myVal, (-127811123 + SIZE)  // two ways to write equ
myVal2  = 723911452394111

fmt:    .string "%d %ld\n"
        .align  4
        .global main
main:
        stp x29, x30, [sp, -16]!
        mov x29, sp

        ldr x0, =fmt
         ldr w1, =myVal    // Note the register type should match the imm type
         ldr x2, =myVal2
        bl  printf

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

Note

- Program sequence of load/store machines loads registers from memory, execute instructions, store result to memory

- A character of the stack: it must be must be quadword aligned (.balign 4)

- RAM in harvard architecture: separates memory for data and instructions