Part I: Hardware Components

1-2. Intro

- Computer organization: Operational units / interconnections that realizes architectural specification
- Computer architecture: Attributes of a system visible to a programmer, attributes that have a direct impact on logical execution of a program (e.g. instruction set, 32bit/64bit, IO, memory addressing)

Moore's law: Number of transistors in a chip doubles every 18 months

- Speed of logic gates increases much faster than speed of memory, memory becomes bottleneck
- MIPS: Million instruction per second: Don't know complexity of each operation
- MFLOPS: Million Floating point Operations per second: Scientific operation
- SPECmarks (Standard Performance Evaluation Corporation): Real jobs

3. Logic

- $A(B+C) = AB + AC; A + (BC) = (A+B)(A+C); \overline{A \cdot B} = \bar{A} + \bar{B}; \overline{A+B} = \bar{A} \cdot \bar{B}$
- Any logic expression can be implemented by AND, OR, NOT, can be reduced to NAND/NOR only
- **Sum of product**: Write truth table, collect minterms (all T/F combination), add up "true" minterms

4. Number

Overflow: Magnitude too large to be represented; Underflow: Magnitude too small to be represented
Base r: $d_n \dots d_2 d_1 d_0 . d_{-1} d_{-2} .. d_{-k} = \sum_{i=-k}^{n} d_i \times r^i = (d_n r + d_{n-1})r + \cdots + d_0$ (plus bit and shift)
Integral part: mod r and divide r, obtain LSB; Fractional part: multiply by r and floor, obtain MSB

- **Sign and magnitude**: Leftmost bit is sign, followed by magnitude → Hard to do arithmetic
- **Excess M**: $B = X + M$: Need to $\mp M$ when doing addition/subtraction; $M = 2^{n-1} - 1$
- **One's complement**: $-ve$: $B = \bar{X} = 2^n - 1 - X$; Add 1 iff have carry from leftmost bit; 2 zeros
- **Two's complement**: $-ve$: $B = \bar{X} + 1 = 2^n - X$; Min value: `10000…`; Max value: `01111…`
  - Signed Multiplication: Sign extend after each bit shift except for sign bit. (repeat sign bit at front) Iff sign bit == 1, Add complement of multiplicand and sign extend. Add and ignore carry out

IEEE Floating point: Sign, exponent, magnitude: ±(1.xxxx) * 2^{exp}; Min: 2^{-127}, Max: 2^{128} * (2 - 2^{-23})

- Exponent: Excess-127 (single precision), Excess-1023 (double precision); All 0: 0, All 1: INF w/sgn
- Significand: Remove the first digit. Actual value = $(-1)^S \times 1.M \times 2^{E-M}$
- Addition/Subtraction: Shift M with smaller exponent, set E to larger exponent, add/sub/normalize
- Multiply (divide): Add (subtract) exponent, subtract (add) bias (excess-127), $\times \div M$ and S, Normalize

Adders: Input: [A, B, (Carry in)], Output: [Sum, Carry out]; Chain input/output of multiple 1-bit adders

6. Memory

- Dynamic RAM: Transistors to store electric charges, need refreshing, slower
- Static RAM: Use logic gates to store data, no refreshing needed, faster, used in cache

ROM/Flash: Non-volatile, store the start-up program of the system

- ROM/PROM (cannot change content), EPROM (erase w/ UV), EEPROM (erase w/ electric current)
- Flash memory: Faster than EEPROM in writing, store BIOS, used as SSD, limited # write cycles

- CPU access memory <u>word by word</u>. Read directly from cache (not main memory); 1 word = 4 bytes
  - Memory is byte addressable (each address is 1 byte), One 32-bit word will occupy 4 addresses
  - Access time (RW operation), <u>Memory cycle time (access time + transfer time)</u>
  - Transfer rate: Memory bandwidth (mem. bus size > word size), Memory "burst mode" (consec.)
  - Different blocks: When waiting for one block, start read another block → increase *throughput*

- Physical type and characteristics: Semiconductor, magnetic, optical, magneto-optical
- Memory byte order (of a word): Big Endian (<u>L to R</u>) (mainframes, SPARC) / Little Endian (R to L) (Intel)
- CPU reads memory by giving address, wait for data. Data copied from lower level to higher level
  - Need to add back transfer time from cache to CPU when source is from main memory
- Principle of locality: Next address to be referenced will be close to current address

13.3. RISC

Reduced instruction set computers: Background

- Advances in computer technology: Fully utilize the large amount of registers
  - Use of large register file: Cache for variables (Re: Register allocation (register-variable mapping))
- Standardized OS/environments (UI): -> can afford to remove compatibility with older instruction sets

Simple instruction set, hardwired (not microprogrammed) control: High clock rate

- Fixed length/format: fetch register operands earlier if they are always at same place in an instruction
- Relatively few operands and addressing modes: Simpler CPU implementation, faster clock rate
  - Instruction pipeline can be more efficient if there are less cases to consider → Low CPI
    - Software and hardware techniques to eliminate pipeline hazards, use optimizing compiler
- Simple instruction: High instruction count, but increment is usually very small
- All instructions are register-register type, except for LD and ST (no Calculate-Operand stage)

Part II: Additional Memory

7. Cache

Cache memory: Unit of transfer is 1 block (4KB), CPU access word by word (from cache). L1 <L2<L3<RAM

- Divide memory and cache into equal-sized blocks (*cache line*). $B_{no} = Addr_{32}/B_{sz}$, $S_{id} = B_{no} \% S_{cnt}$
  - Store $B_{id} = B_{no}/S_{cnt}$ in cache tag, block placed in any 1(direct)-8(way) lines in set (many to one)
  - Direct: adv: quick selection speed; dis: ↑cache miss if program need data mapped to same block
- Memory block is referenced → retrieve block from cache / move entire block from memory to cache
- **Replacement**: Cache set full → select block in cache to be replaced by incoming block (LRU, FIFO)
- **Write through**: Write cache & mem, execute while m.write; **Write back**: Write mem when replaced
- <u>Split cache</u> (instruction, data) vs Unified cache (balance, memory contention problem for pipeline)
- **Cache hit rate** = 1 - #miss/#tot.access; Avg memory access time = hit time + miss rate * miss penalty

12.2. Virtual Memory

Virtual memory: (Re: limited memory); Based on principle of locality (Re: Cache), Transparent to user

- **Logical addressing space**: Addressing space as seen by user programs (e.g. 32-bit address, 0 to 4GB)
  - Each process has its own logical addressing space, each program can start from address 0
  - Not in physical memory, but mapped to different places in the physical addressing space
- **Physical address**: Addresses that are used to actually access the memory (can be smaller, e.g. 1GB)
- Both addressing spaces divided into pages (blocks) of fixed size (4KB)
- Memory Management Unit (MMU) maps logical address (programs) to physical address (memory)
  - Get address line from CPU, pass address line to memory; Protection checking (e.g. segfault)

Paging: Each process will need to keep track of a page table, on how to perform memory mapping

- Each page in logical address space has a corresponding page table entry (PTE) (a row) in page table
  - <u>Valid</u> (page in mem?); <u>Protection</u> (RW); <u>Dirty</u> (page modified?); <u>Phy.frame.no</u> (where in phy.mem)
- Page table cached in the Translation Lookaside Buffer (TLB)
- *Demand paging*: When program starts, nothing is in memory. Only bring in pages when it is needed
  - Advantage: Fast response, because we do not need to bring in the entire program for it to start
  - Disadvantage: Lots of page faults until a stable working set of program is brought into memory;
    - Use prepaging to bring in a number of pages at the beginning to minimize this

Address translation workflow: When program access a memory location via the CPU address bus,

1. Extract the local (logical) page number (most significant n-bits) and offset
2. Find the corresponding PTE from page table, see if the page is in physical memory
   - YES: Get physical frame number, append offset (same as (1)) within the page to it, access mem.
   - NO: Generate a **page fault**. (OS fetch required page from hard disk, put in memory)
     - Find page from harddisk -> Find a free page in physical memory, replace if no free page; If the page is dirty, write back to hard disk. Invalidate the page table entry of that page Write the page to that physical page -> Modify PTE to contain correct info
     - Slow harddisk IO → Suspend current process, switch to another process in the meantime

T6. Cache vs Virtual Memory

Combining cache and virtual memory:

1. Address translation via TLB. If PTE not in TLB, use page table, put the corresponding entry into TLB
2. Get physical address, Use this physical addr. to access cache memory (Re: cache miss handling (ch.7))

Cache vs virtual memory	Cache (<1 order)	Virtual memory (>3 order)
Placement: Where place block	Only on a particular set	Full associative; Anywhere
Identification: How find block	Tag matching, associative	Table lookup via page table
Replacement: Which block?	FIFO/LRU	Approximate FIFO/LRU
Write back/write through?	Either	Always write back

Improve cache performance:

- Change block size: Larger block → Larger neighborhood brought into cache, #block reduced
- Optimize array access s.t. the elements accessed should be in a way that the next array element should be as close to the current array element as possible
- Write buffer for a write-through system, so that CPU do not need to wait for the write to complete

8. External

Magnetic disks: Each surface divided into circular tracks, each track divided into sectors (512 bytes)
Seek: Move the RW head to target cylinder; Rotational latency: Rotate to required sector, ½ revolution
Data transfer time: < seek + latency; Time to rotate for one sector = time for 1 revolution/#sectors
Total time for n consec. sector = seek time + rotational latency + n * rotation time for 1 sector
RAID0: Consec. Sectors over the drives, efficient for accessing a block; RAID1: Duplicate disk, efficient
RAID3: Extra HDD for parity bit; RAID5: Partiy strip (block level) across HDD; RAID6: Two parity strips

Part III: External Services

9. Input / Output

Devices have large speed variation, cannot be controlled by a single clock (synchronous communication)
Asynchronous communication:

- Processor send a request to the IO module, IO module acknowledge with device status
- If device ready, processor send the data by means of a command to the IO module
- IO module obtain data from device, Data transferred back to processor

Processor/device communication: Command decoding, Data, Status reporting, Address recognition
Data buffering: <u>Store the data in buffer register</u> (because device is slow), transaction can more efficient

- Wait for device to fill in a block, and transmission is then in blocks, instead of words

Error detection and correction: Parity bit

IO ports: CPU controls operation of IO devices by writing/reading data/status/control registers

- These registers can be in dedicated IO ports if IO instructions are provided in the CPU

Memory-mapped IO: Registers are put in the memory map, IO operations performed using mem.RW

Programmed IO: Processor execute a program that gives it direct control of the IO operation

- Processor send command, then repeatedly check device for "ready" status until operation is finished
- Control and Status Register (CSR) is used to control the operation of the device, and report its status
- Waste CPU cycle waiting for the device, especially for slow device, such as printer

Interrupt-driven IO: Processor issue an IO command, then continue to execute other process

- When IO finishes, IO module interrupts the CPU
  - Processor finish current instruction execution, check for interrupt, send interrupt INTA to device
  - Interrupt routine save those registers that have been used by interrupt routine to the stack

Direct memory access: Use intelligent device controller (with IO processor), minimize CPU intervention

- CPU will tell IO processor: *"Read the device and place the data into memory location x to y"*
- IO processor directly write to memory. Prevent CPU and IO from both writing memory: *Steal cycles*
  - IO processor signals CPU to disconnect itself from the buses, let IO device control memory bus
  - CPU will see an elongated clock, CPU wait for end of clock cycle to perform its next action
  - IO processor reads/writes the memory for one cycle
  - IO processor finishes with current word, removes signal, clock return to normal, CPU normal op.
  - CPU will execute at a slower rate. When IO processor finished the entire IO operation, it interrupts the CPU, notify the end of IO operation

12.1. OS Services

OS: A software that controls the execution of programs on a processor, manages processor's resources

- <u>Interface to user</u>: Command line interface (shell), file system manipulation, program execution
- <u>Interface to program</u>: OS calls, File open/close, File RW, shared memory operations, process priority
- <u>Services provided by OS</u>: Program creation (editor/compiler/debugger), Program execution, (read program from hard disk to main memory), IO interface (device sharing), File system (permissions), Error detection and response (report error to programs), Accounting (usage statistics, performance)
  - System access: Control access, Protect system resources/data from unauthorized users
  - E.g. memory protection: segfault when you try to access unauthorized memory

Protection scheme: CPU execute in different modes to facilitate protection (User mode/Kernel mode)

- OS is supposed to be well-tested, while bugs exist in user programs. If (buggy) user programs are allowed to handle system resources, they may hang the system, or hold the resources indefinitely
- Only the OS (in kernel mode) will have the rights to some resources (e.g. IO) or actions
  - System change from user mode to kernel mode to run/use protected resources (via system calls)
  - Users cannot change themselves to kernel mode

Multitasking and time-sharing system: CPU is shared among processes to fully utilize resources

- Process uses up its allocated time slice → stop execution → let next process to be executed
  - When a process cannot proceed (e.g. perform IO), it will also stop execution
- System maintain a priority queue of processes, pick first process in the queue for execution
- Priority depend on: process wait how long? processes used previous time slice? system load?

Part IV: Instruction Execution Cycle

5. Execution

Two basic operation in CPU: <u>Data Movement via system buses, Data Transformation via ALU</u>

- PC: Stores the address containing the instruction to be executed; IR: current instruction, mem[PC/4]
  - PC is automatically increased by the size of the instruction (4 bytes) during the execution cycle
  - PC is changed if you execute an instruction involving transfer of control (goto, function call)
- MAR: Holds the address for the main memory, directly connected to external address bus to memory
- MBR: Contains data returned from external memory, or data to be written to the external memory
- System bus: Provide data transfer, source register put data on bus, destination read from bus
- Registers: CPU storage; [User-visible: GPR, index, base pointers], [Hidden: PC/IR, MAR/MBR, PSW]
- General purpose registers: Access by number, fed to register file for reading register content

Instruction: 32 bit word, encoding the operations to be performed by CPU

- Instruction fetch: `PC→MAR, mem[MAR / 4]→IR, PC+=4  # fetch memory from PC address`
- Instruction decode: CU will decode the operations to be performed and set up necessary operations
  - Identify the instructions by the instruction code, read registers and put in ALU source operands
- Operand fetch: PC points to operand: PC→MAR, mem[MAR]→MBR, MBR→MAR, mem[MAR]→MBR

Interrupt: IO devices will generate an interrupt signal to the CPU

- Check for interrupt after we finish the current instruction, no need to save all temporary registers
  - Save: Flag register, Program counter, Register file (only those used in interrupt program, stack)
  - Save PC in the stack, pop stack when finish interrupt program (resume execution)

13.2: Pipeline

Pipeline: Achieve lower clock per instruction; Ideal pipeline: 1 instruction/clock cycle

- Divide single instruction into multiple stages in instruction execution cycle (FI, DI, CO/FO/EI/WO)
- Different stages from different instructions can be overlapped (while doing current DI, do next FI)

Pipeline hazards: Prevent the next instruction from entering the pipeline

- <u>Resource hazard</u>: Resource conflict (e.g. PC increment and ADD operation both use ALU)
  - Hardware used by each stage should not overlap, otherwise duplicate resources are needed:
    - E.g. dedicated incrementer for PC, separate read port for data and instruction (FI/FO), multiple internal buses instead of single bus
- <u>Data hazard</u>: Data dependency on previous instructions (e.g. ADD operand depends on prev result)
  - Need to wait for previous Write-Out before current Fetch-Operand
  - Solution: Re-arrange instructions s.t. WO run before FO (e.g. insert 2+ instructions in between)
  - Hardware solution: Forward the result of ALU to the input of ALU (*data forwarding*)
  - Types of data hazard: Read-After-Write, Write-After-Read, Write-After-Write
- <u>Control hazard</u>: Don't know what is the next instruction after a branch/function call
  - FI stage of the next instruction cannot start until the branch is resolved (don't know which to FI)
  - Solution: Branch prediction: Continue execution along one of the two paths (don't do WO)
    - Dynamic branch prediction: Change our prediction after two consecutive wrong predictions

Processor performance: Execution time = Instruction_count * Clock_per_instruction * Clock_cycle_time

- Effective pipelining (CPI↓), Multiple instruction execution unit, Large register file (↓#memory access), Simplified instruction set (reduced need for microprograms)

13.1. Control Signals

Data movement is controlled by control signals: e.g. MAR <- PC

- E.g. MAR <- PC: [1. Signal tell PC to put its content on bus; 2. Signal tell MAR to get value from bus]
- Multiple registers can get data from bus at same time
- Other operations may require more control signals: e.g. telling ALU what operation to perform

How to generate the control signals?

- **Hardwired control**: Control signals generated by logic gates (use truth table and logic gates)
  - Faster, but more complicated design
  - Modern processor tends to use hardwired control with simple instruction set
- **Microprogrammed control**: Control signals/truth tbl stored in mem., mem. addr input: microcodes
  - Even though the control store is inside the CPU, memory access is still slower
  - Simple design and implementation, easy to modify/debug

A machine uses ***a 32-bit word*** to represent single-precision floating point numbers as follows:

S (1-bit)	8-bit Exponent (E)	23-bit Significand (M)

The value presented is given by $(-1)^S\ 1.M \times 2^{E-127}$

Mapping between main memory and cache memory. Given a 32-bit address:

Block id	Cache Set No.	Offset in block

$\underbrace{\text{Block id} \quad \text{Cache Set No.}}_{\text{block no. in main memory}}$

COMP2120: The Final Note Jeffrey Lee
Version 1.0 May 2019

10.1. Instruction set

- Elements of a machine instruction: Opcode (1 byte), Parameters (Operands/addressing mode)
  - Operation code: Specifies the operation to be performed (e.g. ADD, SUB)
  - Operands: Specifies the location of the source and destination operand
    - For most instructions, always 3 operands: 2 source, 1 destination
    - For less than 3 operands, the other operands are implied:
  - Next instruction: Specifies the location of the next instruction call (Branch/procedure call)
  - Arithmetic operation: Treats operands as numbers (shift right extends sign)
  - Logical operation: Treats operands as bit patterns (shift right adds 0 regardless)
- Memory models: Memory in unit of bytes, machine access memory in words (4/8 bytes)
- General purpose registers: Can be used freely
- Special purpose registers: Have dedicated purpose (e.g. point to strings, floating point operands)
  - PSW (processor status word): Flag register, define condition codes
- Dedicated purpose registers: Program counter (PC), Stack pointer (SP), MAR, MBR
- Data types: 8/16/32/64 bytes: Numeric (int, float), Non-numeric (char: ASCII/Unicode, bitmap)

10.2. Addressing Mode: Calculate the effective address of an operand

- Built into the instruction set of a machine, no need to use extra instructions to calculate address
  - Reduce program code size, more complicated hardware
- Immediate addressing: Instruction provides actual operand value (in instruction word/following word)
  - `LD #0xFF, R6`: Load the value 255 to R6
- Direct addressing: Operand field contains the address of the operand
  - `MOV A, R1`: Move the content at address A to R1
- Indirect addressing: Data read is read at the address of the actual data
  - `MOV (A), R1`: Move the content pointed by the value at A to R1 (A contains address to operand)
  - Register indirect addressing: Address of operand is in the specified register (e.g. ADD R1, (R2), R1)
- Register addressing: All Operands are in registers (Re: Modern CPU/RISC, high speed of register access)
  - `MOV R1, R2`: Move the content of R1 to R2
- Displacement addressing: Address of memory is given by: Register + an offset; (Re: Array access, stack)
  - `MOV A(R2), R2`: Move the content at (A + R2) to R4
  - Used to access local variables in function calls (stored in stack frame: base pointer + offset)
- Stack addressing: Stack pointers point to top or empty space above top of stack (not cached!)
  - `PUSH R4: SUB SP, #4, SP;  MOV R4, (SP)`
  - `POP  R3: MOV (SP), R3;    ADD SP, #4, SP`

2120 Instruction Set

Arithmetic operations:
ADD R1, R2, R3; R3 <- R1 + R2
SUB R1, R2, R3; R3 <- R1 − R2
ADD A, B; B <- A + B
ADD A; AC <- AC + A (accumulator)
ADD; Pop two from stack, ADD, push
 	
AND R1, R2, R3; R3 <- R1 and R2
OR R1, R2, R3; R3 <- R1 or R2
NOT R1, R3; R3 <- not R1
 	
MOV R1, R3; R3 <- R1
 	
LD A, R3; R3 <- A, A is in memory
ST R1, A; A <- R1, A is in memory
 	
PUSH, POP
	Conditional branch: Based on the result of a previous ALU instrction (flag reg.)
	
	BR L; Unconditional, Always goto L
	BZ L; Branch if zero flag is set
	BNZ L; Branch if zero flag is NOT set
	BGT R9, R10, L; if (R9 > R10) goto L
	BLT R9, R10, L; if (R9 < R10) goto L
	BNE R9, R10, L; if (R9 != R10) goto L
	BEQ R9, R10, L; if (R9 == R10) goto L
	
	HLT; Stop the program
	RET; Return to OS / End function call
	
	Call L; function call, special branch
	Store address of next instr (return addr) in system stack

Programmed IO

```
Control register RFCSR
     Bit 0: Ready bit, the meter is ready
     Bit 1: Value has been read and stored in RFBR
     Bit 2: Set this bit to start reading the rainfall meter
Buffer register RFBR

Ready:   LD RFCSR, R2        # read device status
         AND R2, #1, R3      # check bit 0
         BEQ Ready           # wait if not ready
         MOV #0x4, R1        # bit pattern 00...00100
         ST R1, RFCSR        # start reading
Loop:    LD RFCSR, R2        # read device status
         AND R2, #0x2, R3    # check bit 1
         BEQ Loop            # Rainfall data read?
         LD RFBR, R0
         CALL Convert
         ST R2, LEDBR1
         ST R1, LEDBR2
```

Displacement addressing mode: `LD DISP (R1), R3: Word 1 [LD|R1|--|R3], Word 2 [DISP]`

```
Read register file   # R1 now in RFOUT1
MAR <- PC            # move PC to MAR, perform memory read
MBR <- mem[MAR]      # read memory, DISP now in MBR
PC <- PC + 4         # increment PC to next instruction
A <- RFOUT1          # move R1 to A
B <- MBR             # move DISP to B
C <- A + B           # address of operand
MAR <- C             # move address of operand to MAR
MBR <- mem[MAR]      # memory read, now mem[R1 + DISP] is in MBR
RFIN <- MBR          # move MBR to RFIN
Write register file
```

ST R4, P

```
Read register file   # R4 now in RFOUT1
MAR <- PC            # PC now points to a memory location storing the saving address
MBR <- mem[MAR]      # Memory read, the target address is in MBR
PC <- PC + 4         # PC points to next instruction
MAR <- MBR           # Move the target address to MAR
MBR <- RFOUT1        # Move the value to be saved to MBR
Write memory
```

ADD

```
Read register port 1, register no. given in source operand 1 field
Result will be put in RFOUT1
Read register port 2, register no. given in source operand 2 field
Result will be put in RFOUT2
Move from RFOUT1 to A input of ALU
Move from RFOUT2 to B input of ALU
Perform ADD in ALU, result in C
Move from C to RFIN
Write register file, writing RFIN to register given in destination operand field
```

BNZ L

```
MAR <- PC            # PC now points to the memory location storing the branch address
MBR <- mem[MAR]      # Get the branch address
PC <- PC + 4         # PC now points to next instruction if no branch
MAR <- MBR           # Store the branch address in MAR
Get condition code   # cc = IR / 65536 % 256
Check if we need to branch. If no branch, do nothing
PC <- mem[MAR]       # Move branch address to PC
```

`if (a[0] > a[1]) x = a[0]; else x = a[1];`

```
.data                         # data segment
a:    .word 1                 # create storage containing 1
      .word 3                 # create storage continaing 3
x:    .word 4                 # create storage contining 4
.text                         # program segment
main:
      ld #a, r8               # r8 = address of a (#a)
      ld 0(r8), r9            # r9 = a[0], i.e. 1
      ld 4(r8), r10           # r10 = a[1], i.e. 3
      bgt r9, r10, f1         # if (r9 > r10), goto f1
      st r10 x                # x = r10
      br f2                   # goto f2
f1: st r9, x                  # x = r9
f2: ret                       # return to OS
```

`temp = 0, a = 1; while (temp < 100) {temp += a;  a++;}`

```
      sub r8, r8, r8      # r8 = 0
      ld #1, r9           # r9 = 1
      mv r9, r10          # r10 = 1
      ld #0x64, r11       # r11 = 100
      jmp c               # while loop: check condition first
f1:   add r8, r9, r8
      add r8, r10, r9
c:    blt r8, r11, f1
```

Convert all characters into uppercase

```
.data
a:    .asciiz "This is a test"
      # zero-terminated string

.text
main:      sub r9, r9, r9
loop:      lb a(r9), r10         # load byte
           beq r10, #10, exit    # r10 == 0? end of string
           call capitalize
           sb r10, a(r9)         # store byte
           add r9, 1, r9         # r9++, next char (byte)
           br loop
exit:      ret

capitalize:
# input is r10, output is r10
# if r10 is lowercase, change to uppercase
           push r8
           push r9
           ld #0x61, r8          # r8 = 'a'
           ld #0x7a, r9          # r9 = 'z'
           blt r10, r8, ret1
           bgt r10, r9, ret1
           sub r10, #0x20, r10   # 0x20 = 'a' - 'A'
ret1:      pop r9                # reverse order
           pop r8
           ret
```

CALL

```
MAR <- PC          # PC stores the memory location storing the call fuction address
MBR <- mem[MAR]    # Memory read, MBR stores call function address
MAR <- MBR         # Store function at TEMP (connected to MAR)
TEMP <- MAR
PC <- PC + 4       # PC points to next instruction in main program
SP <- SP − 4       # SP points to address of new top element (decreasing)
MAR <- SP          # Move stack top address to MAR
MBR <- PC          # Move PC to MBR, prepare for memory write
Write memory       # PC is now at the top of stack (pointed by SP)
MAR <- TEMP        # Replace MAR with the function address
PC <- MAR          # Replace PC with the function address
```

RET

```
MAR <- SP          # Move stack top address to MAR
SP <- SP + 4       # Pop stack address
MBR <- mem[MAR]    # Read memory, main function instruction address in MBR
PC <- MBR          # Replace PC with address of next instruction in main
```

Floating point representation

```
Write down the bit pattern corresponding to the value 7.375:
0.375 * 2 = 0.75; 0.75 * 2 = 1.5; 0.5 * 2 = 1:  7.375 = 111.011_2 = 1.11011 * 2^2
Sign = 0
Exponent = 1023 + 2 = 1025 = 100 0000 0001
Significand = 1101 1000 0000 0000 0000 0000
2's complement: 401d80000
```

Binary representation proofs

```
2's complement sign extension (expanding from m-bits to n-bits), -ve case
     Bit pattern for m-bit representation: 2^m - |A|
     After sign extend: Bit pattern = 2^m - |A| + 2^(n-1) + 2^(n-2) + … + 2^m
                                    = 2^m - |A| + 2^n − 2^m = 2^n - |A|, represent -|A|

2's complement addition
     +ve, -ve: WLOG, N1 > 0, -N2 < 0
          Add two bit patterns together: 2^n + N1 − N2
          If (N2 > N1), result is 2^n − (N2 − N1), which is −(N2 − N1)
          If (N2 < N1), then ignoring carry (remove 2^n), we get (N1 − N2)
          If (N2 == N1), then ignoring carry, we get 0

     -ve, -ve: -N1 < 0, N2 < 0
          Add two bit patterns together: 2^n + 2^n − N1 − N2 = 2^n − (N1 + N2)

1's complement addition
     +ve, -ve: WLOG, N1 > 0, -N2 < 0: 2^n − 1 − N2
          Add two bit patterns together: 2^n − 1 + N1 − N2
          If (N2 > N1), result is 2^n − (N2 − N1), which is −(N2 − N1)
          If (N2 < N1), then ignoring carry (remove 2^n), we get (N1 − N2 − 1). Add 1
          If (N2 == N1), then ignoring carry, we get 2^n − 1, which is -0

     -ve, -ve: -N1 < 0, N2 < 0
          Add two bit patterns together: 2^n -1 2^n -1 − (N1 + N2)
                                       = 2^n − 2 −(N1 + N2): discard carry out, Add 1

Excess M: B = X + M
     Addition: B1 + B2 = X1 + M + X2 + M = (X1 + X2) + 2M: Minus M
     Subtract: B1 - B2 = X1 + M - X2 - M = (X1 + X2): Add M

Multiplication of signed operand (2's complement)
     Consider A * -B, where
     Let C = the part of -B without sign bit = 2^n−B−2^(n-1) = -B+2^(n-1), -B=C-2^(n-1)
     A * -B = A (C − 2^(n-1)) = AC − A * 2^(n-1)
     AC: multiplier without sign bit * multiplicand
     -A * 2^(n-1): sign-bit * multiplicand. Take 2's complement −ve, sign extend
```