IMPLEMENTING A SEQUENTIAL CPU

Unit 1
The Meltdown and Spectre exploits use “speculative execution.” What’s that?

You know the trolley problem? Well, for a while now, CPUs have basically been sending trolleys down both paths, quantum-style, while awaiting your choice. Then the unneeded “phantom” trolley disappears.

That sounds bad. Honestly, I’ve been assuming we were doomed ever since I learned about rolhammer.

What’s that? If you toggle a row of memory cells on and off really fast, you can use electrical interference to flip nearby bits and—do we just suck at...computers?

Yup, especially shared ones.

So you’re saying the cloud is full of phantom trolleys armed with hammers...

...yes, that is exactly right. Okay, I’ll, uh...install updates?

Good idea.
LEARNING GOALS

- Given the notation from the textbook, describe in plain English and at a high level what an instruction does
- Given a Y86 instruction and its description convert that to the short hand form from that describes what the instruction does
- Given the compiler’s assembly output of a C program map between the C and assembly
- Given a small piece of C convert it to Y86
- Convert from x86-64 to Y86
- Describe in plain English what a small piece of x86-64 or Y86 is doing
- Explain the purpose and use of the Y86 assembler directives
- Given the description of a an instruction format for an ISA translate between assembly and machine language byte encodings and vice versa
**SHORT TERM PLAN**

- Look at a new instruction set architecture
  - Goal - understand how it might be implemented

- Study how it is implemented in hardware
  - Goal - Appreciate the implications on program performance and behaviour

- Look at a pipelined implementation
  - Goal - Appreciate the implications on program performance and behaviour
computers implement an Instruction Set Architecture
- ISA: Abstract expression of what a machine does
- Independent of actual implementation
- Multiple implementations of the same ISA

compilers translate high-level language programs into sequences of low-level hardware instructions

CPU hardware executes
- one instruction at a time – one per cycle (or does it?)
- cycle is Fetch then Execute
- instructions are stored in memory
- execution transforms register and memory contents from one state to another
WHAT DOES THE CPU DO?

Fetch next instruction → Execute instruction → Check for interrupts
FETCH/EXECUTE CYCLE

From Computer Systems: A Programmer’s Perspective
Most ISAs have in common:

- Some set of registers to work with
- Rules for accessing memory
- A set of instructions for:
  - Manipulating memory
  - Manipulating registers
  - Controlling the flow of instruction execution
**Y86-64 INTRODUCTION**

- Y86-64: invented by the textbook authors as a teaching tool.
- A simple subset of the x86-64 (Intel) architecture.
  - Less instructions, simpler encoding, simpler addressing modes
- Inspired by RISC (reduced instruction set computer)
- It has
  - 15 general-purpose 64-bit registers.
  - 1 program counter (PC).
  - 3 condition codes zero (ZF), sign (SF), overflow (OF).
    - We may also refer to a carry flag (CF).
  - 12 types of instruction.
Y86-64 ISA

- Instructions are similar to 213 but with different names
  - Basic types are the same
- Conditional jumps use condition-codes instead of registers
- Address where a function call returns to is on the stack and not in a register
- Convention: arguments are passed using registers instead of stack
Y86-64 REGISTERS

- Registers (64 bits each):

<table>
<thead>
<tr>
<th></th>
<th>%rax</th>
<th>%rsp</th>
<th>4</th>
<th>%r8</th>
<th>%r12</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>%rcx</td>
<td>%rbp</td>
<td>5</td>
<td>%r9</td>
<td>%r13</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>%rdx</td>
<td>%rsi</td>
<td>6</td>
<td>%r10</td>
<td>%r14</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>%rbx</td>
<td>%rdi</td>
<td>7</td>
<td>%r11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Instructions that only need one register use F (15) for the second register.
- Additional flags available: overflow flag (OF), zero flag (ZF), sign flag (SF)
  - on or off depending on result of previous operation
  - only accessible indirectly
- Memory contains $2^{64}$ addressable bytes
  - all data accesses load/store 64 bit words aligned on an 8 byte boundary
INSTRUCTION TYPES

- All ISAs have a few types of instructions
  - Memory based operations (load, store)
  - Register based operations (moves, arithmetic operations)
  - Change program counter (jumps, branches, call, return)
  - Conditional operations (allows loops, if/then/else)

- Some ISAs have instructions that might combine a couple of these operations
  - Example: load from memory and add to register
Y86-64 Instruction Types

- register/memory transfer (at most one memory access per instruction)
- arithmetic and register move (no memory access allowed)
- jumps (conditional and unconditional)
- conditional moves (register move only if condition is true)
- stack manipulation
- procedure calls (special kind of jump)
- miscellaneous (halt, nop)
Y86-64 ISA: INSTRUCTIONS

- register/memory transfers:
  - `rmmovq rA, D(rB)  M_8[D + R[rB]] ← R[rA]`
    - Example: `rmmovq %rdx, $0x20(%rsi)`
  - `mrmovq D(rB), rA  R[rA] ← M_8[D + R[rB]]`
    - Example: `mrmovq $0x0A(%rdx), %rsi`
Y86-64 ISA: INSTRUCTIONS

- Register manipulation
  - `rrmovq rA, rB` \( R[rB] \leftarrow R[rA] \)
  - `irmovq V, rB` \( R[rB] \leftarrow V \)

- Arithmetic instructions
  - `addq rA, rB` \( R[rB] \leftarrow R[rB] + R[rA] \)
  - `subq rA, rB` \( R[rB] \leftarrow R[rB] - R[rA] \)
  - `andq rA, rB` \( R[rB] \leftarrow R[rB] \land R[rA] \)
  - `xorq rA, rB` \( R[rB] \leftarrow R[rB] \oplus R[rA] \)
  - `mulq rA, rB` \( R[rB] \leftarrow R[rB] \ast R[rA] \)
  - `divq rA, rB` \( R[rB] \leftarrow R[rB] / R[rA] \)
  - `modq rA, rB` \( R[rB] \leftarrow R[rB] \% R[rA] \)

Extensions, not in the book
Y86-64 ISA: INSTRUCTIONS

- Unconditional jump
  - jmp Dest  PC ← Dest

- Conditional jumps
  - jle Dest  PC ← Dest if last result ≤ 0
  - jl Dest   PC ← Dest if last result < 0
  - je Dest   PC ← Dest if last result = 0
  - jne Dest  PC ← Dest if last result ≠ 0
  - jge Dest  PC ← Dest if last result ≥ 0
  - jg Dest   PC ← Dest if last result > 0
Y86-64 ISA: INSTRUCTIONS

- **Conditional moves**
  - `cmovle rA, rB` $\quad R[rB] \leftarrow R[rA]$ if last result $\leq 0$
  - `cmovl  rA, rB` $\quad R[rB] \leftarrow R[rA]$ if last result $< 0$
  - `cmove  rA, rB` $\quad R[rB] \leftarrow R[rA]$ if last result $= 0$
  - `cmovne rA, rB` $\quad R[rB] \leftarrow R[rA]$ if last result $\neq 0$
  - `cmovge rA, rB` $\quad R[rB] \leftarrow R[rA]$ if last result $\geq 0$
  - `cmovg  rA, rB` $\quad R[rB] \leftarrow R[rA]$ if last result $> 0$
Y86-64 ISA: INSTRUCTIONS

- Stack operations and procedure calls
  - call Dest
    
    \[ R[\%rsp] \leftarrow R[\%rsp] - 8; \]
    \[ M_8[R[\%rsp]] \leftarrow PC; \ PC \leftarrow Dest; \]
  - ret
    
    \[ PC \leftarrow M_8[R[\%rsp]]; \ R[\%rsp] \leftarrow R[\%rsp] + 8 \]
  - pushq rA
    
    \[ R[\%rsp] \leftarrow R[\%rsp] - 8; \ M_8[R[\%rsp]] \leftarrow R[rA] \]
  - popq rA
    
    \[ R[rA] \leftarrow M_8[R[\%rsp]]; \ R[\%rsp] \leftarrow R[\%rsp] + 8 \]

- Others
  - halt
  - nop
# Using Zero and Sign Flags (Simplified)

<table>
<thead>
<tr>
<th>Condition</th>
<th>Test</th>
<th>Zero flag</th>
<th>Sign Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>&gt; 0</td>
<td>0</td>
<td>and 0</td>
</tr>
<tr>
<td>ge</td>
<td>&gt;= 0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>== 0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ne</td>
<td>!= 0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>le</td>
<td>&lt;= 0</td>
<td>1</td>
<td>or 1</td>
</tr>
<tr>
<td>l</td>
<td>&lt; 0</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
Y86 ASSEMBLY LANGUAGE NOTES

- **Labels**
  - symbolic names for addresses, assigned using label:
  - used anywhere a number can be used (number replaces label)

- **Fixed values**
  - `.byte X` insert the 8-bit number X
  - `.long X` insert the 32-bit number X (in Little Endian)
  - `.quad X` insert the 64-bit number X (in Little Endian)
  - `.byte X, n / .long X, n / .quad X, n` same as above, repeated n times

- **Adjust memory location**
  - `.pos X` set the address of the next instruction (or directive) to X
  - `.align X` ensure that address of next instruction is aligned to X bytes, padding (adding unused space) if necessary
Y86-64 ISA: FUNCTION CALLS AND STACK

- Parameters are passed in registers unless there are too many.
- Stack pointer (%rsp) represents the top of the stack:
  - %rbp is sometimes used for the base of the frame (where stack was when function started).
- Stack is used for:
  - local variables in functions
  - arguments (if too many)
  - returning address
  - register values before a function call that may change them.
STACK STRUCTURE (NO BASE POINTER)

- Local variables
- Local variables
- Return Address
- Argument 7
- Argument 8
- Argument 9
- Caller's Local variables
- ...
- Caller's Return Address
- ...

(%rsp)
16(%rsp)
8(%rsp)
24(%rsp)
## Stack Structure (With Base Pointer)

| (\%rsp) | Local variables | Local variables | 8(\%rsp) or -16(\%rbp) |
| (\%rbp) or 24(\%rsp) | Saved caller's \%rbp | 32(\%rsp) or 8(\%rbp) |
| Return Address | Argument 7 | Argument 8 | Argument 9 |
| Caller's Local variables | ... | Caller's Return Address | ... |
CALLING CONVENTIONS

- Some conventions define register and stack usage when one function calls another
- This permits compiler to create code for a function without knowing where it will be called from
- Calling conventions are:
  - Compiler dependent
  - Architecture dependent
  - Operating system dependent
- For example:
  - 32-bit, x86, gcc-based calling convention is called cdecl (after C)
213 CALLING CONVENTIONS (FOR COMPARISON)

- All parameters passed on the stack
- Parameters are pushed on the stack in reverse order
- All registers can be overwritten by the called function (except r5 and r6)
- Function’s return value is in r0
- Return address for function is passed through r6
Y86-64 Calling Conventions (Based on X86-64)

- Stack pointer: `%rsp`
- Arguments passed in registers, in this order: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`
  - Additional arguments passed in stack
- Returning value passed in `%rax`
- Caller-save registers: `%rax`, `%r10`, `%r11` + arguments
  - Callee can change their values at will
  - Caller responsible for saving before calling other functions
- Callee-save registers: `%rsp`, `%rbx`, `%rbp`, `%r12`, `%r13`, `%r14`
  - Caller assumes they have same value when function returns
  - Callee must restore value before returning
long start[] = { 4, 7, 8, 9, 12, 11 };

long sum_function () {
    long sum = 0;
    long count = 6;
    long *str = start;

    while (count) {
        sum += *str;
        str++;
        count--;
    }
    return sum;
}
Compiled to assembly by gcc 4.9.2 with -O1 -S

long start[] = { 4, 7, 8, 9, 12, 11 };  

long sum_function () {
    long sum = 0;
    long count = 6;
    long *str = start;

    while (count) {
        sum += *str;
        str++;
        count--;
    }

    return sum;
}

sum_function:
    movl $start, %edx  # long *str = start
    xorl %eax, %eax  # long sum = 0
    .L2:
        addq (%rdx), %rax  # sum += *str
        addq $8, %rdx  # str++
        cmpq $start+48, %rdx  # if str != &start[6]
        jne .L2  # loop
    rep ret
ASSEMBLY CODE EXAMPLE

Converting to y86-64

sum_function:

```
  movl $start, %edx
  xorl %eax, %eax
.L2:
  addq (%rdx), %rax
  addq $8, %rdx
  cmpq $start+48, %rdx
  jne .L2
  rep ret
```

sum_function:

```
  irmovq start, %rdx # long *str = start
  xorq %rax, %rax # long sum = 0
.L2:  mrmovq (%rdx), %rbx
  addq %rbx, %rax # sum += *str
  irmovq $8, %rbx
  addq %rbx, %rdx # str++
  irmovq end, %rbx
  subq %rdx, %rbx # if str != &start[6]
  jne .L2             # loop
  ret
```
EXERCISE: TRANSLATE TO Y86

```c
int a = 1830;
int b = 1131;
int gcd;

int main() {
    int r;
    while (b > 0) {
        r = a % b;
        a = b;
        b = r;
    }
    gcd = a;
}
```
int a = 1830;
int b = 1131;
int gcd;

int main() {
    int r;
    while (b > 0) {
        r = a % b;
        a = b;
        b = r;
    }
    gcd = a;
}

CPSC 313 2018 WT1 © 2018 Jonatan Schroeder
WHAT DO WE KNOW SO FAR

- Y86-64 instructions
- Way to specify what an instruction does
- Translation of C to Y86 for simple problems

Now that we know what an instruction does we want to know how an instruction is represented in memory so that the CPU can retrieve it and then execute it.
### Y86-64 Memory Representation

<table>
<thead>
<tr>
<th>Command</th>
<th>Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>00</td>
</tr>
<tr>
<td>nop</td>
<td>10</td>
</tr>
<tr>
<td>cmovXX rA, rB</td>
<td>2fn</td>
</tr>
<tr>
<td>irmovq V, rB</td>
<td>30F</td>
</tr>
<tr>
<td>rmmovq rA, D(rB)</td>
<td>40rA</td>
</tr>
<tr>
<td>rmmovq D(rB), rA</td>
<td>50rA</td>
</tr>
<tr>
<td>OPI rA, rB</td>
<td>6fn</td>
</tr>
<tr>
<td>jXX Dest</td>
<td>7fn</td>
</tr>
<tr>
<td>call Dest</td>
<td>80</td>
</tr>
<tr>
<td>ret</td>
<td>90</td>
</tr>
<tr>
<td>pushq rA</td>
<td>A0</td>
</tr>
<tr>
<td>popq rA</td>
<td>B0</td>
</tr>
</tbody>
</table>
**Y86-64 MEMORY REPRESENTATION**

- **Instructions format:**
  - **Arithmetic instructions:**
    - `addq` → fn = 0
    - `subq` → fn = 1
    - `andq` → fn = 2
    - `xorq` → fn = 3
    - `mulq` → fn = 4
    - `divq` → fn = 5
    - `modq` → fn = 6

- **Conditional jumps and moves:**
  - `jmp` → fn = 0
  - `jle` → fn = 1
  - `jl` → fn = 2
  - `je` → fn = 3
  - `jne` → fn = 4
  - `jge` → fn = 5
  - `jg` → fn = 6
Y86-64 MEMORY REPRESENTATION

- Translate the following into machine language
  - `mrmovq 0x2000(%rax), %rdx`
  - `xorq %rsi, %rbx`
  - `jne $0x1234`
  - `irmovq $0x376, %rax`

- Translate the following into assembly language
  - `0x25 0x42`
  - `0x80 0x12 0x34 0x56 0x78 0x00 0x00 0x00 0x00`
PART 2: ISA IMPLEMENTATION
LEARNING GOALS

- Identify CPU stages and the order an instruction goes through them in our Y86 sequential processor
- Describe/explain the general functionality of each instruction execution stage
- Describe/explain, in plain English, what happens in each stage as an instruction is executed
- Use the notation from the text to describe what happens in each stage of the processor as the instruction is executed
BASIC IDEA

- The parts
  - register file
  - PC and instruction registers
  - main memory
  - combinatorial logic (and gates, or gates, etc.)
  - clock

- Describing the combinational logic
  - We will use C/Java-like statements to describe instruction semantics
  - book uses C-like HCL for the same purpose
  - real chips use hardware description languages such as Verilog, VHDL
**PROCESSOR COMPONENTS**

- Memory (green boxes)
  - register file, main memory, program counter (PC) and condition codes (CC)

- Multiplexers (blue boxes)
  - control input selects output from one of several data inputs
  - similar to switch statement in Java

- Special-Purpose Registers (grey boxes)
  - connect memory and multiplexers
  - named to indicate their function
  - later these will store data between CPU stages
  - in Java these are instance variables of stage that contains them
Modularize by dividing instruction execution into 6 stages:
- **Fetch**: read instruction and decide on new PC value
- **Decode**: read from registers
- **Execute**: use the ALU to perform computations
- **Memory**: read data from or write data to memory
- **Write-back**: store value(s) into register(s)
- **PC update**: store the new PC value
IN MORE DETAILS

- **Fetch**
  - Use PC to read next instruction from memory into instruction register
  - Determine instruction length and extract pieces of instruction
  - Advance the PC to the address of next instruction in sequence

- **Decode**
  - Read values from register file

- **Execute**
  - Perform ALU operations AND determine whether jumps are taken

- **Memory**
  - Read data from or write data to main memory

- **Write back**
  - Store values back to registers, if needed

- **Update PC**
  - Store the new PC value
The things the programmer sees:
- Registers -> %rax, %rbx, %rsp …
- Program counter -> PC
- Memory -> M₁[addr], M₈[addr] …
- Status register bits ZF, SF, OF
  - Set when certain instructions execute
  - Indirectly used by certain instructions
  - Sometimes referred to as condition codes (CC) or program status word (PSW)
CAVEAT - PURPOSE OF INSTRUCTIONS

- An instruction precisely describes the actions made on the current programmer visible state to get to the “next” state
  - Example: rrmovq rA, rB
    - R[rB] <- R[rA]
    - PC <- PC + 2
  - The underlying hardware implementation to achieve the required outcome may be different provided the result conforms to the instruction/architecture specification
SPECIAL PURPOSE REGISTERS

- Fetch:
  - icode: the opcode for the current instruction (0 to F).
  - ifun: the arithmetic logical operation, or the branch/move condition.
  - rA, rB: register numbers contained in the instruction.
  - valC: constant, displacement, or target of a jXX or call instruction.
  - valP: the address of the instruction after the current one.
SPECIAL PURPOSE REGISTERS

- **Decode:**
  - srcA: first register to read from (often but not always the same as rA).
  - srcB: second register to read from (often but not always the same as rB).
  - valA: the value read from register srcA.
  - valB: the value read from register srcB.

- **Execute:**
  - valE: value computed by the ALU.
  - cnd (aka bch): whether or not the branch condition was satisfied.
SPECIAL PURPOSE REGISTERS

- **Memory:**
  - `valM`: value read from memory.

- **Write back:**
  - `dstE`: register to which value `valE` should be written.
  - `dstM`: register to which value `valM` should be written.

- **PC update:**
  - `nextPC`: the address of the next instruction (taking `jXX`, call, ret into account).
The “variables” are the names of signals

- \( \text{icode:iFun} \leq M_1[\text{PC}] \)
- \( \text{rA:rB} \leq M_1[\text{PC}+1] \)
- \( \text{valC} \leq M_8[\text{PC} + ?] \)
- \( \text{valP} \leq \text{Current PC} + ? \)
**DECODE STAGE**

- Read up to 2 operands from the register file:
  - srcA, srcB – from rA, rB, or %rsp
  - valA – value read from R[srcA] (valA = R[rA])
  - valB – value read from R[srcB] (valB = R[rb])

- Also setup signals for writing back to registers
  - dstM – from rA, or %rsp
  - dstE – from rB, or %rsp
EXECUTE STAGE

- Depending upon the instruction, produces valE – the value of the computation performed at the execute stage.

- This could be:
  - Result of operation (+, -, /, * etc) specified by ifun
  - Effective address for a memory reference
  - Increment or decrement of the stack pointer

- Condition Codes (CC) (i.e. ZF, OF, SF) could be set if it is an arithmetic operation

- Conditional instructions (jXX, cmovXX): evaluates result of corresponding condition
  - Translates previous CC to bch (flag) based on ifun
MEMORY, WRITE BACK, PC UPDATE

- Memory
  - valM, but only if something is read from memory

- Write back
  - Writes results to registers, would be valM or valE

- PC update
  - PC – set to the next instruction to execute
  - will come from one of valM, valC, valP
At each stage we ask: Which of the above needs to updated and what inputs need to be used to accomplish the update? *What needs to be updated is a function of the instruction.*
Let's implement this instruction
- general form: rmmovq rA, D(rB)
- example: rmmovq %rax, 0x100(%rbx)

Fetch
- icode:iFun <= M_1[PC]
- rA:rB <= M_1[PC+1]
- valC <= M_8[PC+2]
- valP <= PC + 10
DESIGN EXAMPLE

- **decode**
  - srcA  $\leftarrow rA$
  - srcB  $\leftarrow rB$
  - dstE  $\leftarrow F$ (value meaning no reg)
  - dstM  $\leftarrow F$
  - valA  $\leftarrow R[srcA]$
  - valB  $\leftarrow R[srcB]$
**DESIGN EXAMPLE**

- **Execute**
  - \( \text{valE} \leftarrow \text{valB} + \text{valC} \)

- **Memory**
  - \( M_8[\text{valE}] \leftarrow \text{valA} \)

- **write back**
  - Not active, no registers to update

- **pc update**
  - \( \text{PC} \leftarrow \text{valP} \)
**DESIGN EXAMPLE**

- Lets implement this instruction
  - general form: `pushq` rA
  - example: `pushq` `%rax`

- First what does the instruction do?
  - `{R[%rsp]} <- {R[%rsp]} - 8`
  - `{M_8[R[%rsp]]} <- {R[rA]}`
  - `PC <- PC + 2`
**FETCH — PUSHQ %RA**

- \( R[\%rsp] \leftarrow R[\%rsp] - 8 \)
- \( M_8[R[\%rsp]] \leftarrow R[\%rA] \)
- \( PC \leftarrow PC + 2 \)
DECODE -- PUSHQ %RA

- \( R[\%rsp] \leftarrow R[\%rsp] - 8 \)
- \( M_8[R[\%rsp]] \leftarrow R[\%rA] \)
- \( PC \leftarrow PC + 2 \)
EXECUTE — PUSHQ %RA

- \( R[\%rsp] \leftarrow R[\%rsp] - 8 \)
- \( M_8[R[\%rsp]] \leftarrow R[\%rA] \)
- \( PC \leftarrow PC + 2 \)
MEMORY— PUSHQ %RA

- R[%rsp] <- R[%rsp] – 8
- M8[R[%rsp]] <- R[%rA]
- PC <- PC + 2
WRITE-BACK — PUSHQ %RA

- \texttt{R[\%rsp]} <- \texttt{R[\%rsp]} - 8
- \texttt{M_8[R[\%rsp]]} <- \texttt{R[\%rA]}
- \texttt{PC} <- \texttt{PC} + 2
PCUPDATE—PUSHQ %RA

- R[%rsp] <- R[%rsp] – 8
- M8[R[%rsp]] <- R[%rA]
- PC <- PC + 2