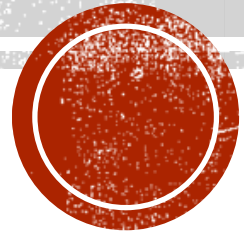
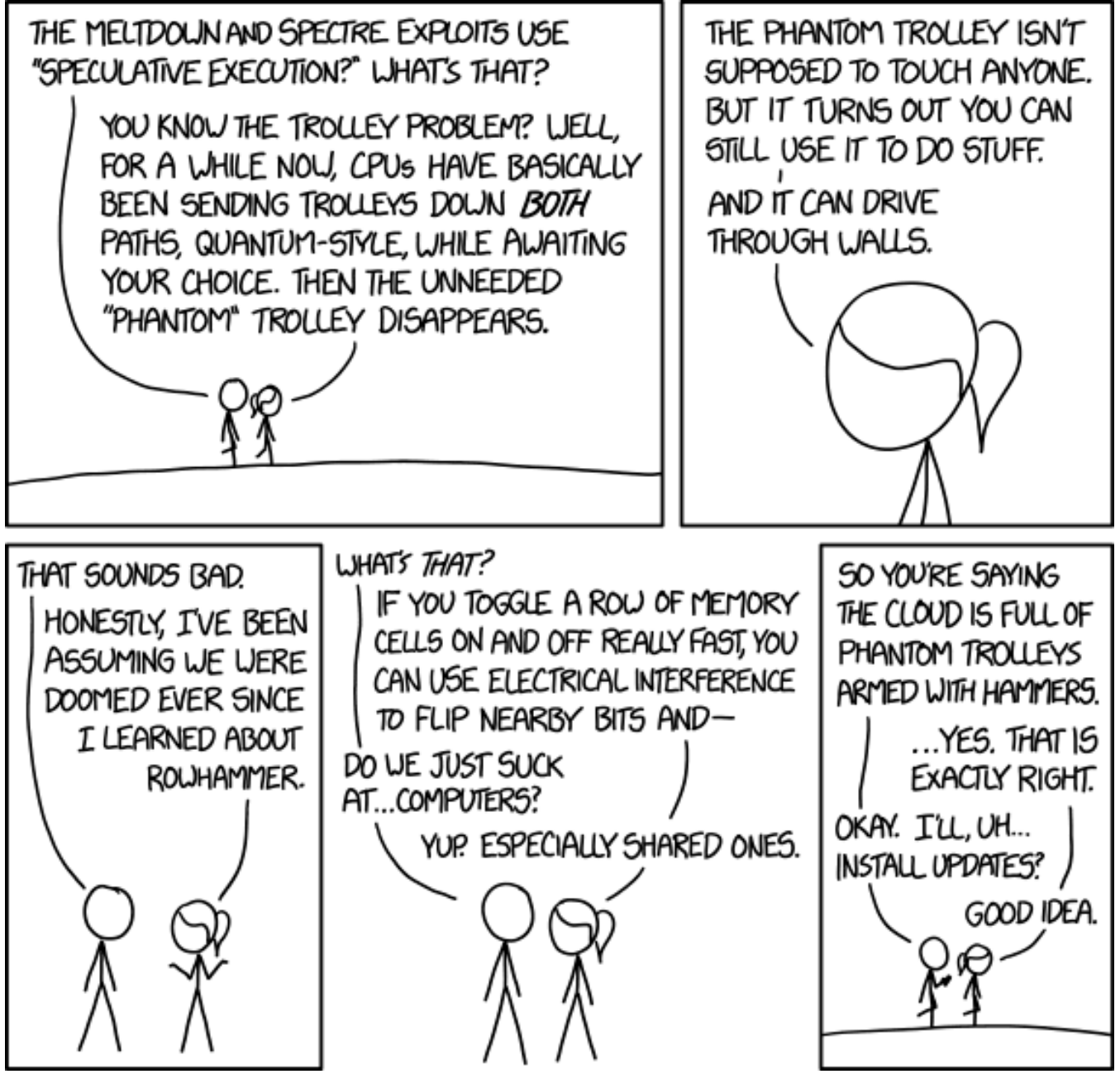
IMPLEMENTING A SEQUENTIAL CPU

Unit 1





https://xkcd.com/1938/

MODULE 1: SEQUENTIAL Y86-64 IMPLEMENTATION

- Module outline
  - Review: CPU functionality
  - Y86-64 Introduction
  - Y86-64 Instruction Set Architecture (ISA)
  - Y86-64 Implementation

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

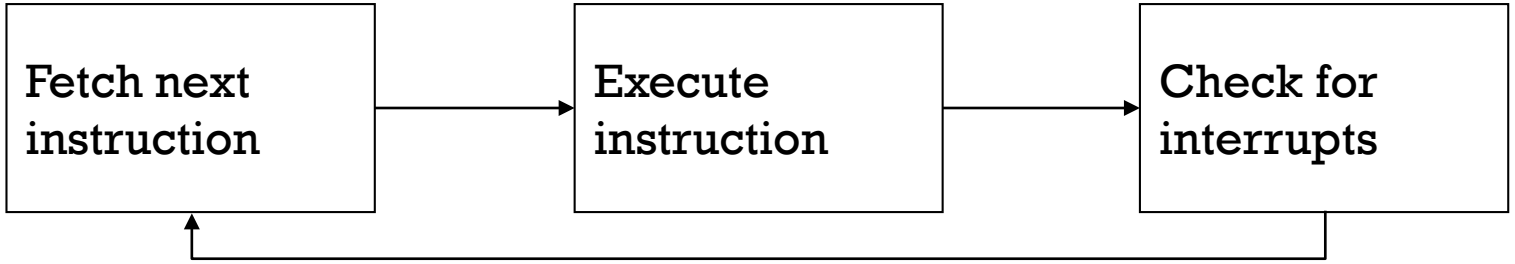
SHORT TERM PLAN

- 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

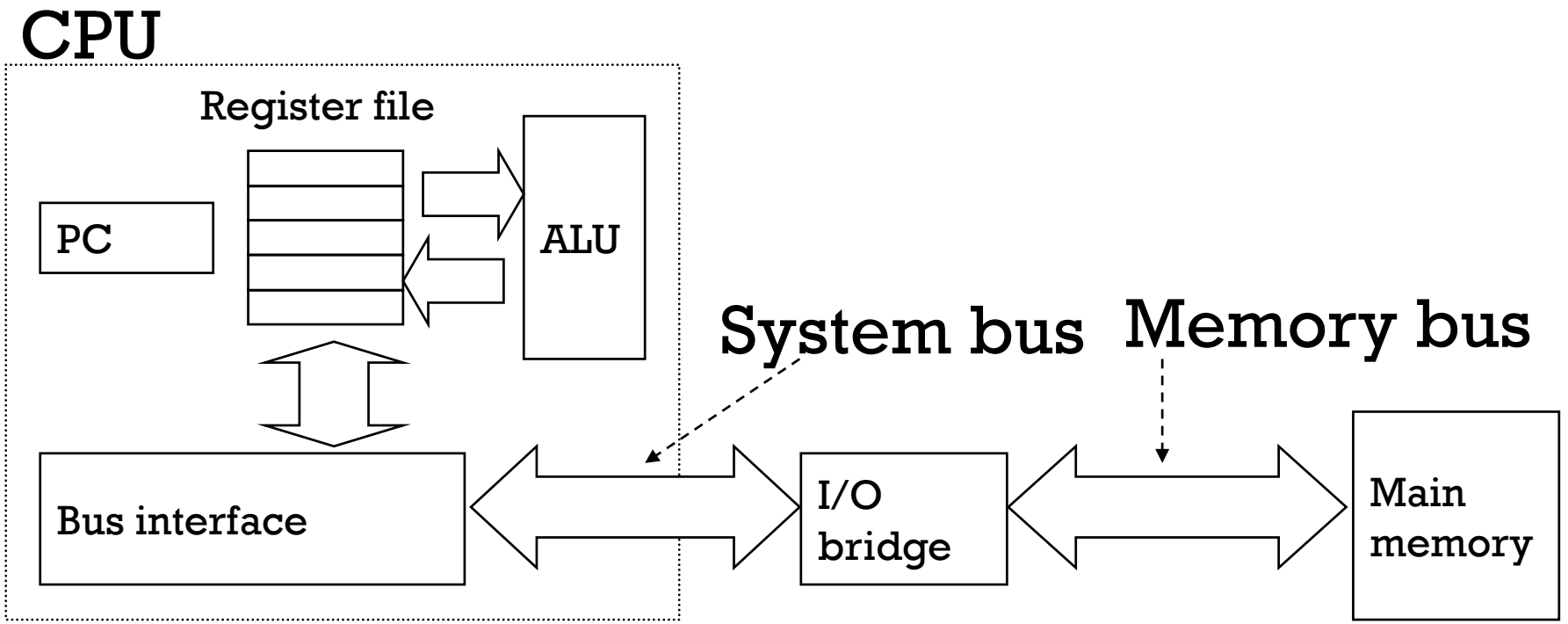
FROM 213 YOU KNOW

- computers implement an Instruction Set Architecture
  - what is an 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/EXECUTE CYCLE



From *Computer Systems: A Programmer's Perspective*

Y86-64 Introduction

- The Y86-64:
  - Invented by the textbook authors as a teaching tool.
  - A simple subset of the x86-64 (Intel) architecture.
    - The x86-64 ISA has
      - Too many instruction with an overly complicated encoding.
      - Too many addressing modes
      - Efficiency over simplicity.
    - So we use a simplified version.
  - 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).
    - 12 types of instruction.

ISA COMMONALITY

- Some number 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 ISA

- Instructions
  - Similar to 213 but with different names
  - Basic types are the same
  - Conditional jumps use condition-codes, which is different
  - 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

MODULE 1.2: Y86-64 INSTRUCTION SET ARCHITECTURE

- Registers (64 bits each):

0	%rax	%rsp	4
1	%rcx	%rbp	5
2	%rdx	%rsi	6
3	%rbx	%rdi	7

8	%r8	%r12	12
9	%r9	%r13	13
10	%r10	%r14	14
11	%r11		

  - Instructions that only need one register use F (15) for the second register.
- Additional flags available: carry flag (CF), zero flag (ZF), sign flag (SF)
  - on or off depending on result of previous operation
- 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
  - Load from memory to a register
  - Store from register to memory
  - Register based operations
    - Move from register to register
    - Arithmetic and binary operations (+, -, *, %, |, &, >>, << etc) that typically work only on register values
  - Change program counter (call, ret, jmp)
  - Conditionally change the program counter (allows loops, if/then/else)
- Some ISAs have instructions that might combine a couple of these operations. For example add and store result to memory

INSTRUCTION TYPES (FIRST 4 OF 7)

- register/memory transfer (at most one memory access per instruction)
  - rmmovq %rdx, $0x20(%rsi) # m [0x20 + r[rsi]] ←r[rdx]
- arithmetic and register move (no memory access allowed)
  - irmovq $5, %rax # r[rax] ←5
  - subq %rax, %rbx # r[rbx] ← r[rbx] - r[rax]
- jumps (conditional and unconditional)
  - jle 0x1000 # pc ← 0x1000 if last arithmetic result <= 0
- conditional moves (a new one)
  - cmovle %rax, %rbx # r[rbx] ← r[rax] if last arithmetic result<=0

INSTRUCTION TYPES (LAST 3 OF 7)

- stack manipulation
  - pushq %rax # m[r[rsp]-8] ← r[rax] and r[rsp] ← r[rsp] - 8
  - popq %rax # r[rax] ← m[r[rsp]] and r[rsp] ← r[rsp] + 8
- procedure calls
  - call 0x1000 # pushq PC and PC ← 0x1000
  - ret # popq PC
- others
  - halt
  - nop
- Complete ISA description in text book
  - Section 4.1.2 (Y86-64 Instructions)
  - a few additional instructions sometimes implemented as part of assignments

Y86-64 ISA: INSTRUCTIONS

- register/memory transfers:
  - rmmovq rA, D(rB) $M_8[D + R[rB]] \leftarrow R[rA]$
    - Example: rmmovq %rdx, $0x20(%rsi)
  - mrmovq D(rB), rA $R[rA] \leftarrow M_8[D + R[rB]]$
    - 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 — $R[rB] \leftarrow R[rB] \wedge R[rA]$
  - xorq rA, rB — $R[rB] \leftarrow R[rB] \oplus R[rA]$
  - mulq rA, rB — $R[rB] \leftarrow R[rB] * R[rA]$
  - divq rA, rB — $R[rB] \leftarrow R[rB] / R[rA]$
  - modq rA, rB — $R[rB] \leftarrow R[rB] \% R[rA]$

Extensions

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 — R[rB] ← R[rA] if last result ≤ 0
  - cmovl rA, rB — R[rB] ← R[rA] if last result < 0
  - cmove rA, rB — R[rB] ← R[rA] if last result = 0
  - cmovne rA, rB — R[rB] ← R[rA] if last result ≠ 0
  - cmovge rA, rB — R[rB] ← R[rA] if last result ≥ 0
  - cmovg rA, rB — R[rB] ← R[rA] if last result > 0

USING ZERO AND SIGN FLAGS (SIMPLIFIED)

Condition	Test	Zero flag		Sign Flag
g	> 0	0	and	0
ge	>= 0			0
e	== 0	1		
ne	!= 0	0		
le	<= 0	1	or	1
l	< 0			1

Y86-64 ISA: INSTRUCTIONS

- Procedure calls and return support
  - call Dest R[%rsp]←R[%rsp] - 8; M_8[R[%rsp]]←PC; PC←Dest;
  - ret PC←M_8[R[%rsp]]; R[%rsp]←R[%rsp] + 8
  - pushq rA R[%rsp]←R[%rsp] - 8; M_8[R[%rsp]]←R[rA]
  - popq rA R[rA]←M_8[R[%rsp]]; R[%rsp]←R[%rsp] + 8
- Others
  - halt
  - nop

Y86 Assembly Language Notes

- labels
  - symbolic names for addresses
  - assigned using label:
  - used anywhere a number can be used (number replaces label)
- .pos X
  - set the address of the next instruction (or directive) to X
- .long X
  - insert the 32-bit number X (in Little Endian) and advance address by 4
- .long X, n
  - insert n 32-bit numbers with value X and advance address by 4n

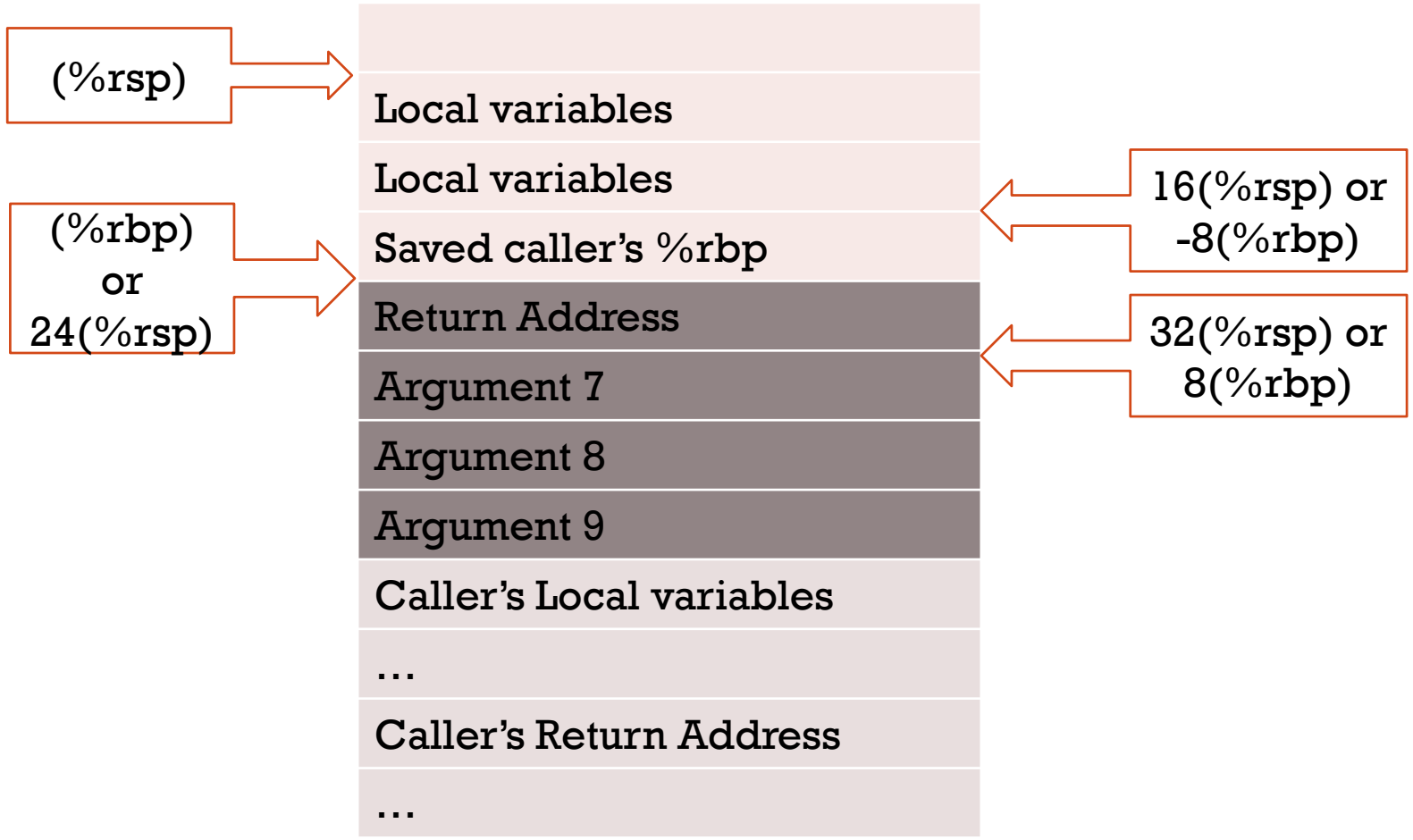
Y86 ASSEMBLY LANGUAGE NOTES (CONT.)

- .quad X / .quad X, n
  - similar to long, but with 64-bit number
- .align X
  - ensure that address of next instruction is aligned to X bytes
  - "pads" memory (adds 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



CALLING CONVENTIONS

- Some conventions defines register and stack usage when one function calls another
- This permits compiler to compile 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)

CDECL OLD CALLING CONVENTIONS (32-BIT VERSION OF GCC)

- All parameters passed on the stack
- Parameters are pushed on the stack in reverse order
- Registers eax, ecx, edx can be over written by the called function
- Function's return value is in eax
- ebp (frame pointer) is pushed onto the stack upon function entry
- ebp is then set to point to the top of stack. Parameters and local variables are then addressed relative to ebp
- Additional rules for floating point registers

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 (caller's responsibility to save before calling other functions): %rax, %r10, %r11 + arguments
- Callee-save registers (function must restore value before returning if changed): %rbx, %rbp, %r12, %r13, %r14
  - In x86-64: %r15*

ASSEMBLY CODE EXAMPLE

- C example:

```
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;
}
```

ASSEMBLY CODE EXAMPLE

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

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

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

Y86 – GCD CODE

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

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

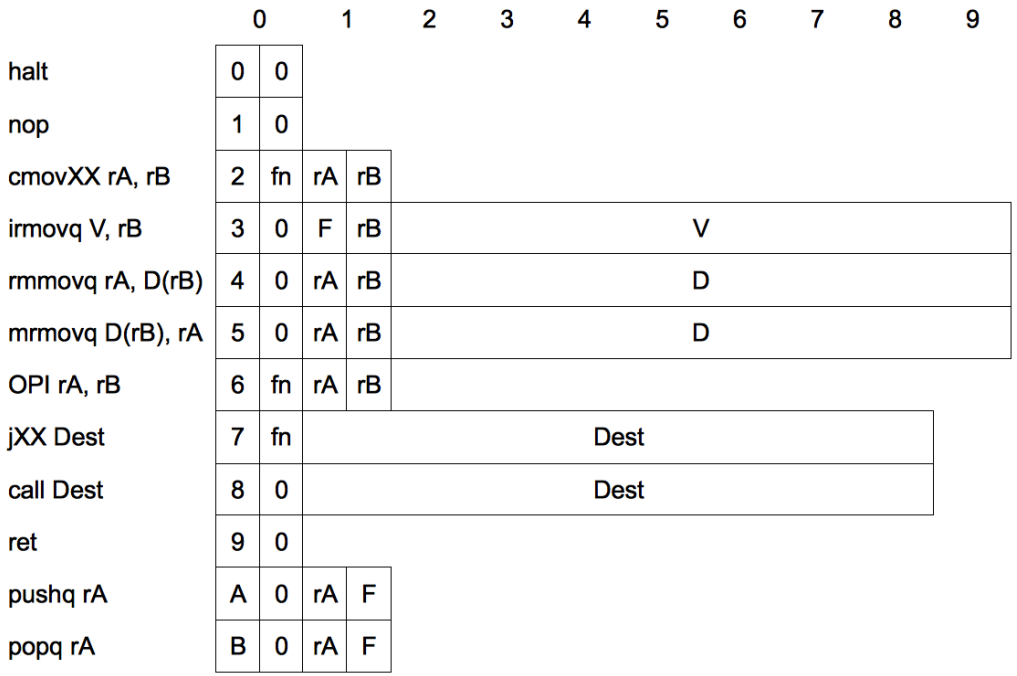
```
.pos 0x100
main:   xorq %rdx, %rdx          # %rdx = 0
        mrmovq a(%rdx), %rax     # %rax = a
        mrmovq b(%rdx), %rbx     # %rbx = b
loop:   subq %rdx, %rbx          # is b <= 0?
        jle     end              # if so, done
        modq %rbx, %rax          # %rax gets remainder
        rrmovq %rax, %rcx        # save remainder
        rrmovq %rbx, %rax        # a = b
        rrmovq %rcx, %rbx        # b = remainder
        jmp     loop
end:    rmmovq %rax, gcd(%rdx) # gcd = a
        rmmovq %rax, a(%rdx)     # update a
        rmmovq %rbx, b(%rdx)     # update b
        halt
.pos 0x1000
a:      .quad 1830
b:      .quad 1131
gcd:    .quad 0
```

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



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

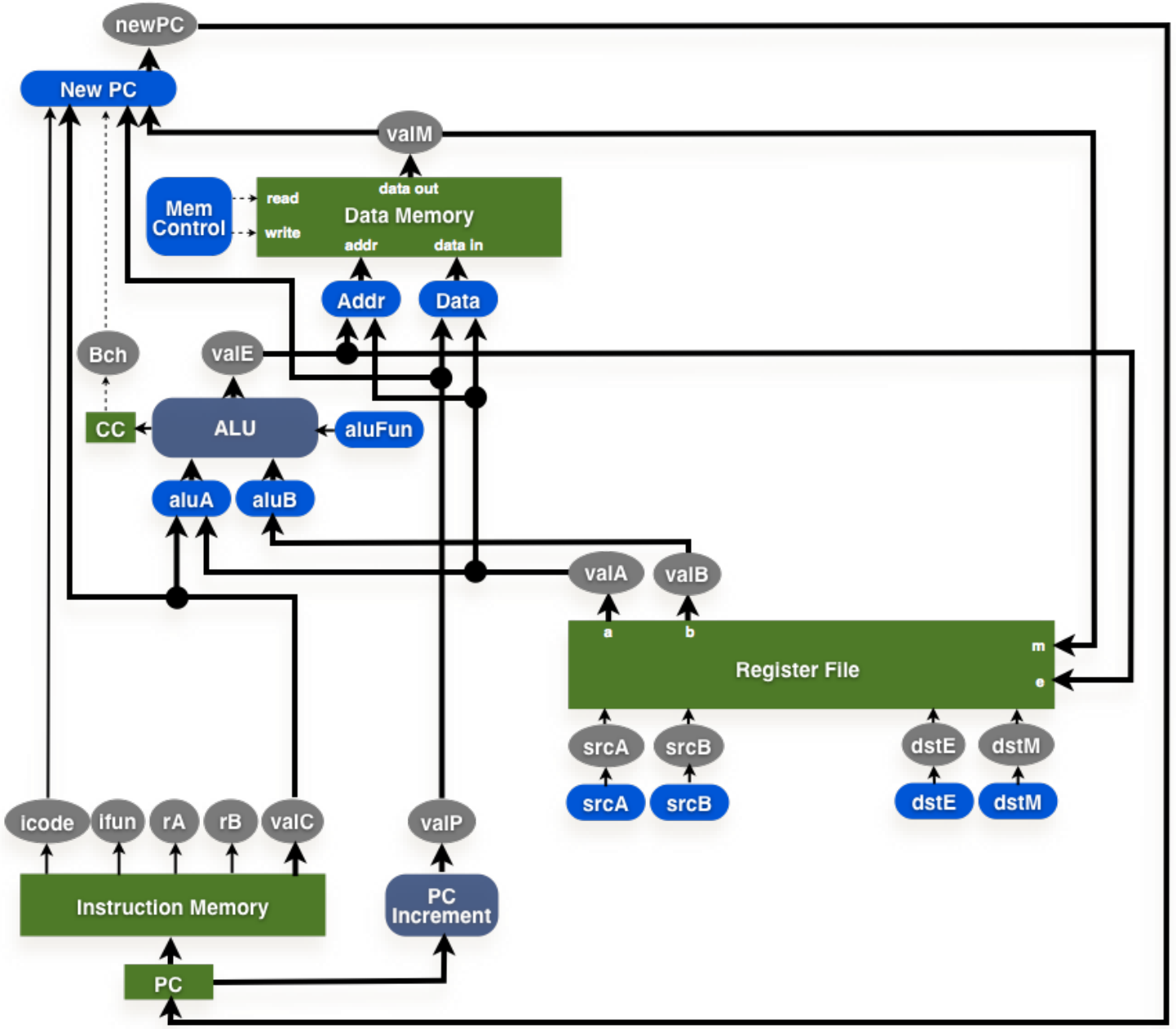
PART 2: ISA IMPLEMENTATION

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
  - use Java to describe instruction semantics
    - book uses C-like HCL for the same purpose
  - real chips use hardware description languages such as Verilog, VHDL


newPC
New PC
valM
Mem Control
read
write
data out
Data Memory
addr
data in
Addr
Data
Bch
valE
CC
ALU
aluFun
aluA
aluB
valA
valB
a
b
Register File
m
e
srcA
srcB
dstE
dstM
srcA
srcB
dstE
dstM
icode
ifun
rA
rB
valC
valP
Instruction Memory
PC Increment
PC

PROCESSOR COMPONENTS

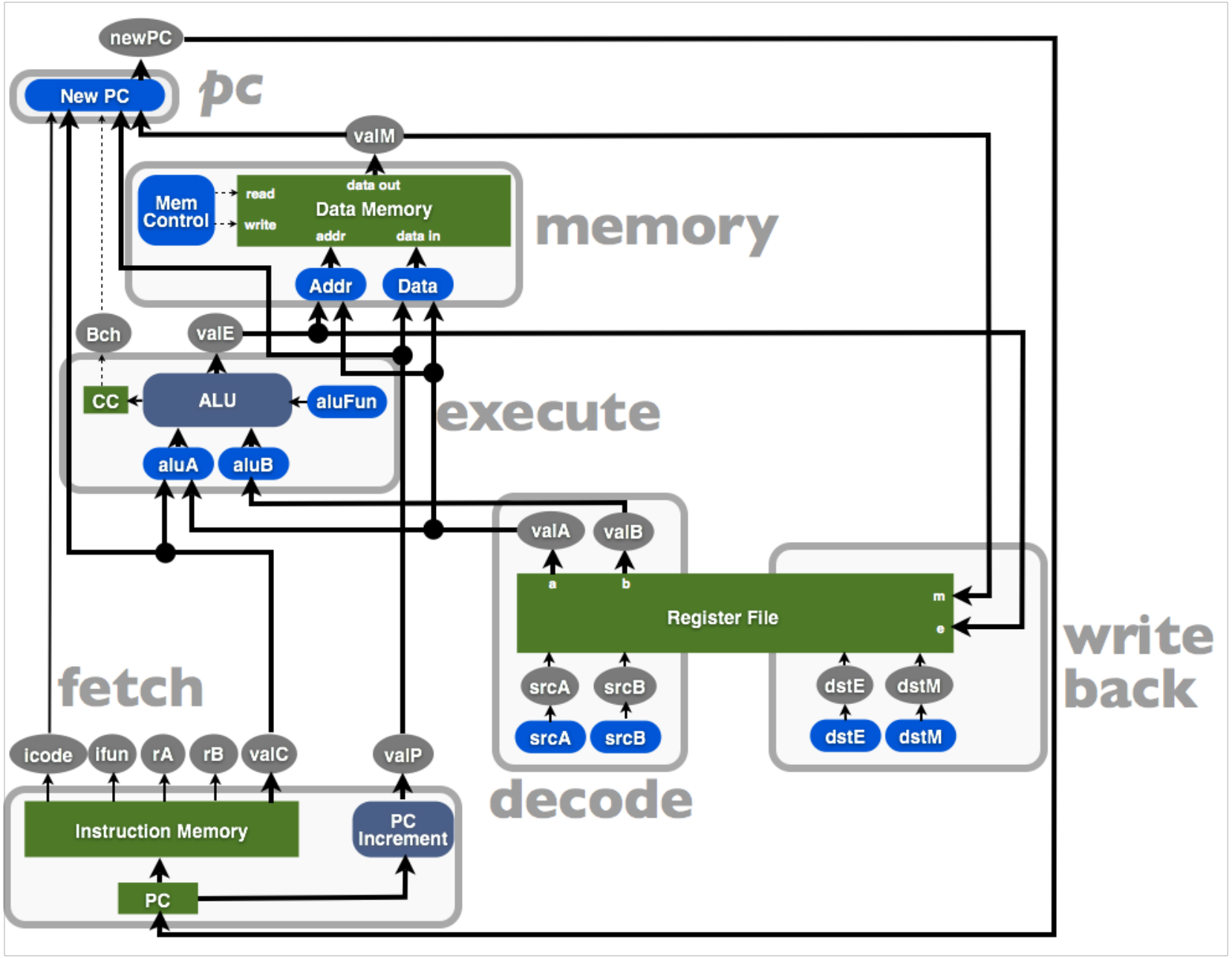
- Memory (green boxes)
  - register file, main memory, program counter (PC) and condition codes (CC)
- Multiplexers (blue boxes)
  - control input selects output from one of several data inputs
  - similar to switch statement in Java
- Special-Purpose Registers (grey boxes)
  - connect memory and multiplexers
  - named to indicate their function
  - later these will store data between CPU stages
  - in Java these are instance variables of stage that contains them

BREAKING THINGS DOWN

- 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


newPC
New PC
pc
valM
Mem Control
read
write
data out
Data Memory
addr
data in
memory
Addr
Data
Bch
valE
CC
ALU
aluFun
execute
aluA
aluB
valA
valB
a
b
Register File
m
e
write back
srcA
srcB
srcA
srcB
dstE
dstM
dstE
dstM
fetch
icode
ifun
rA
rB
valC
valP
Instruction Memory
PC Increment
PC
decode

Y86 PROGRAMMER VISIBLE STATE

- The things the programmer sees:
  - Registers -> %rax, %rbx, %rsp …
  - Program counter -> PC
  - Memory -> M_1[addr], M_8[addr] …
  - Status register bits ZF, SF, OF
    - Set when certain instructions execute
    - Read 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 regsiter srcB.
- Execute:
  - valE: value computed by the ALU.
  - cnd: whether or not the branch condition was satisfied.

Module 1.3: Y86-64 Implementation – 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:
  - newPC: the address of the next instruction (taking jXX, call, ret into account).

FETCH STAGE

- The "variables" are the names of signals
  - icode:iFun <= M_1[PC]
  - rA:rB <= M_1[PC+1]
  - valC <= M_8[PC + ?]
  - valP <= 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

- 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

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

QUICK SUMMARY

Stage	Signals (i.e. ``Variables``)
Fetch	icode, ifun, rA, rB, valC, ValP
Decode	srcA, srcB, dstM, dstE, valA, valB
Execute	valE, CC
Memory	valM
Write back	--
PC Update	PC

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*.

DESIGN EXAMPLE

- Lets 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 ←rA
  - srcB ←rB
  - dstE ←F (value meaning no reg)
  - dstM ←F
  - valA ←R[srcA]
  - valB ←R[srcB]

DESIGN EXAMPLE

- Execute
  - valE ← valB + valC
- Memory
  - M_8[valE] ← valA
- write back
  - Not active, no registers to update
- pc update
  - PC ← 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] <- R[%rsp] – 8
- M_8[R[%rsp]] <- R[%rA]
- PC <- PC + 2

DECODE – PUSHQ %RA

- R[%rsp] <- R[%rsp] – 8
- M_8[R[%rsp]] <- R[%rA]
- PC <- PC + 2

EXECUTE – PUSHQ %RA

- R[%rsp] <- R[%rsp] – 8
- M_8[R[%rsp]] <- R[%rA]
- PC <- PC + 2

MEMORY– PUSHQ %RA

- R[%rsp] <- R[%rsp] – 8
- M_8[R[%rsp]] <- R[%rA]
- PC <- PC + 2

WRITE-BACK – PUSHQ %RA

- R[%rsp] <- R[%rsp] – 8
- M_8[R[%rsp]] <- R[%rA]
- PC <- PC + 2

PCUPDATE– PUSHQ %RA

- R[%rsp] <- R[%rsp] – 8
- M_8[R[%rsp]] <- R[%rA]
- PC <- PC + 2