CS 261
Fall 2016

Mike Lam, Professor

x86-64 Misc Topics

Topics

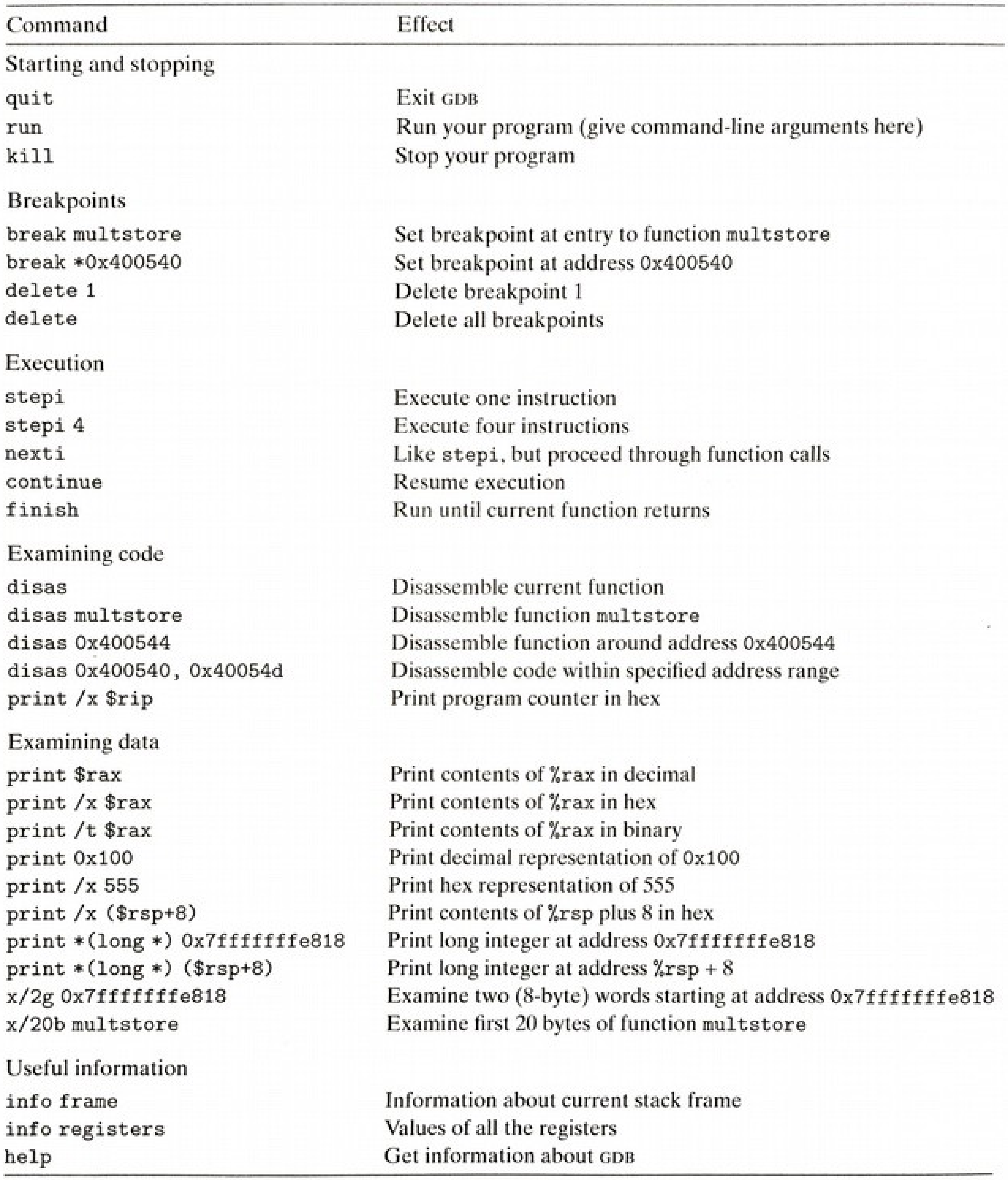
- Pointer wrap-up
- Buffer overflows and mitigation
  - Stack randomization
  - Corruption detection
  - Read-only code regions
- Floating-point code
- Conclusion

Pointers

- Every pointer has a type and a value
  - Casting changes type but not value
- Pointer values are simply addresses in memory
  - NOT the same as the pointer's address
- Pointers are created with '&' and dereferenced with '*'
  - **Declaration != creation!**
  - Addresses of variables aren't stored explicitly until a pointer is created
- Arrays and pointers are closely related in C
  - Array variable = pointer to first element
- In assembly, indirect addressing modes are similar to pointers
  - Register name vs. register value vs. indirect memory value
  - Pointer name vs. pointer value vs. dereferenced value

GDB

- Learn it!



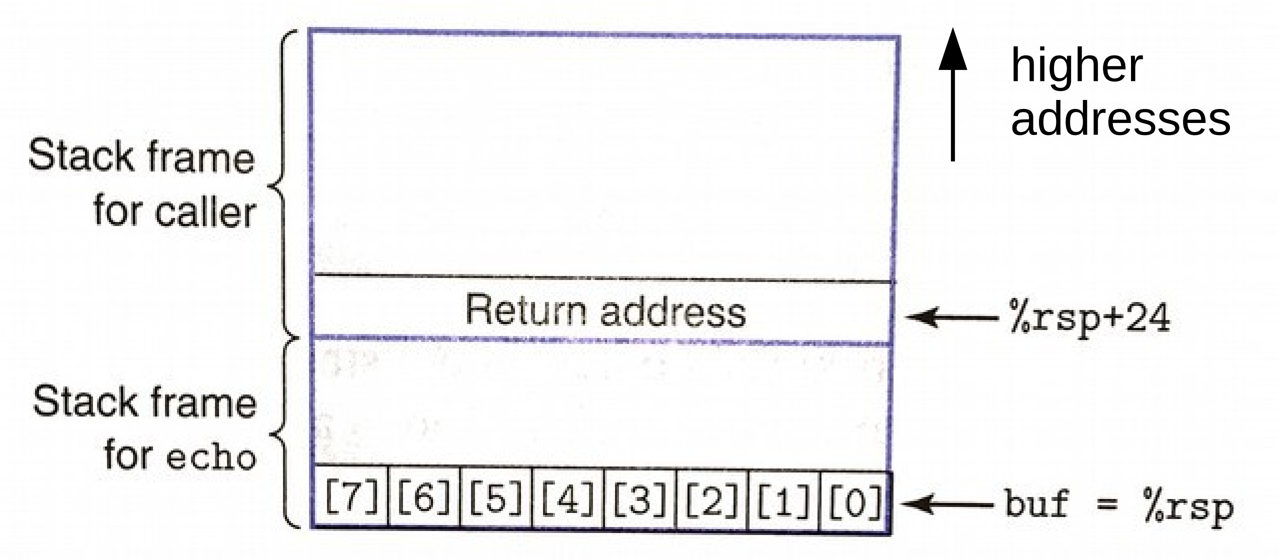
Command	Effect
Starting and stopping	
`quit`	Exit GDB
`run`	Run your program (give command-line arguments here)
`kill`	Stop your program
Breakpoints	
`break multstore`	Set breakpoint at entry to function `multstore`
`break *0x400540`	Set breakpoint at address `0x400540`
`delete 1`	Delete breakpoint 1
`delete`	Delete all breakpoints
Execution	
`stepi`	Execute one instruction
`stepi 4`	Execute four instructions
`nexti`	Like `stepi`, but proceed through function calls
`continue`	Resume execution
`finish`	Run until current function returns
Examining code	
`disas`	Disassemble current function
`disas multstore`	Disassemble function `multstore`
`disas 0x400544`	Disassemble function around address `0x400544`
`disas 0x400540, 0x40054d`	Disassemble code within specified address range
`print /x $rip`	Print program counter in hex
Examining data	
`print $rax`	Print contents of `%rax` in decimal
`print /x $rax`	Print contents of `%rax` in hex
`print /t $rax`	Print contents of `%rax` in binary
`print 0x100`	Print decimal representation of `0x100`
`print /x 555`	Print hex representation of 555
`print /x ($rsp+8)`	Print contents of `%rsp` plus 8 in hex
`print *(long *) 0x7fffffffe818`	Print long integer at address `0x7fffffffe818`
`print *(long *) ($rsp+8)`	Print long integer at address `%rsp` + 8
`x/2g 0x7fffffffe818`	Examine two (8-byte) words starting at address `0x7fffffffe818`
`x/20b multstore`	Examine first 20 bytes of function `multstore`
Useful information	
`info frame`	Information about current stack frame
`info registers`	Values of all the registers
`help`	Get information about GDB

Buffer overflows

- Major C/x86-64 security issue
  - C does not check for out-of-bounds array accesses
  - x86-64 stores return addresses and data on the same stack
  - Out-of-bound writes to local variables may overwrite other stack frames
  - Allows attackers to change control flow just by providing the right "data"
  - Many historical exploits (including Morris worm)

```
void echo ()
{
    char buf[8];
    gets(buf);
    printf(buf);
}
```

DO NOT WRITE CODE LIKE THIS!

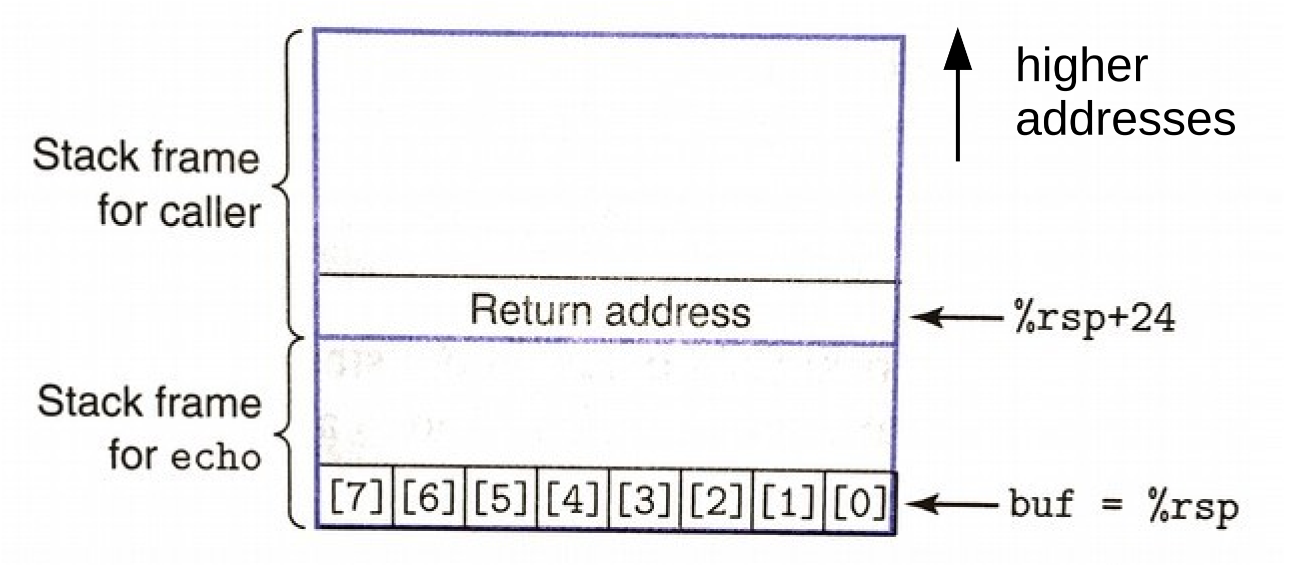


Buffer overflows

- Shellcode (exploit code)
  - Pre-compiled snippets of code that exploit a buffer overflow

```
char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

Complication: Must pad the shellcode with address of the buffer (guess and/or use a NOP-sled)



Mitigating buffer overflows

- Stack randomization
  - Randomize starting location of stack
  - Makes it more difficult to guess buffer address
  - In Linux: address-space layout randomization
- Corruption detection
  - Insert a canary (guard value) after each array
  - Check canary before returning from function
- Read-only code regions
  - Mark stack memory as "no-execute"
  - Hinders just-in-time compilation and instrumentation

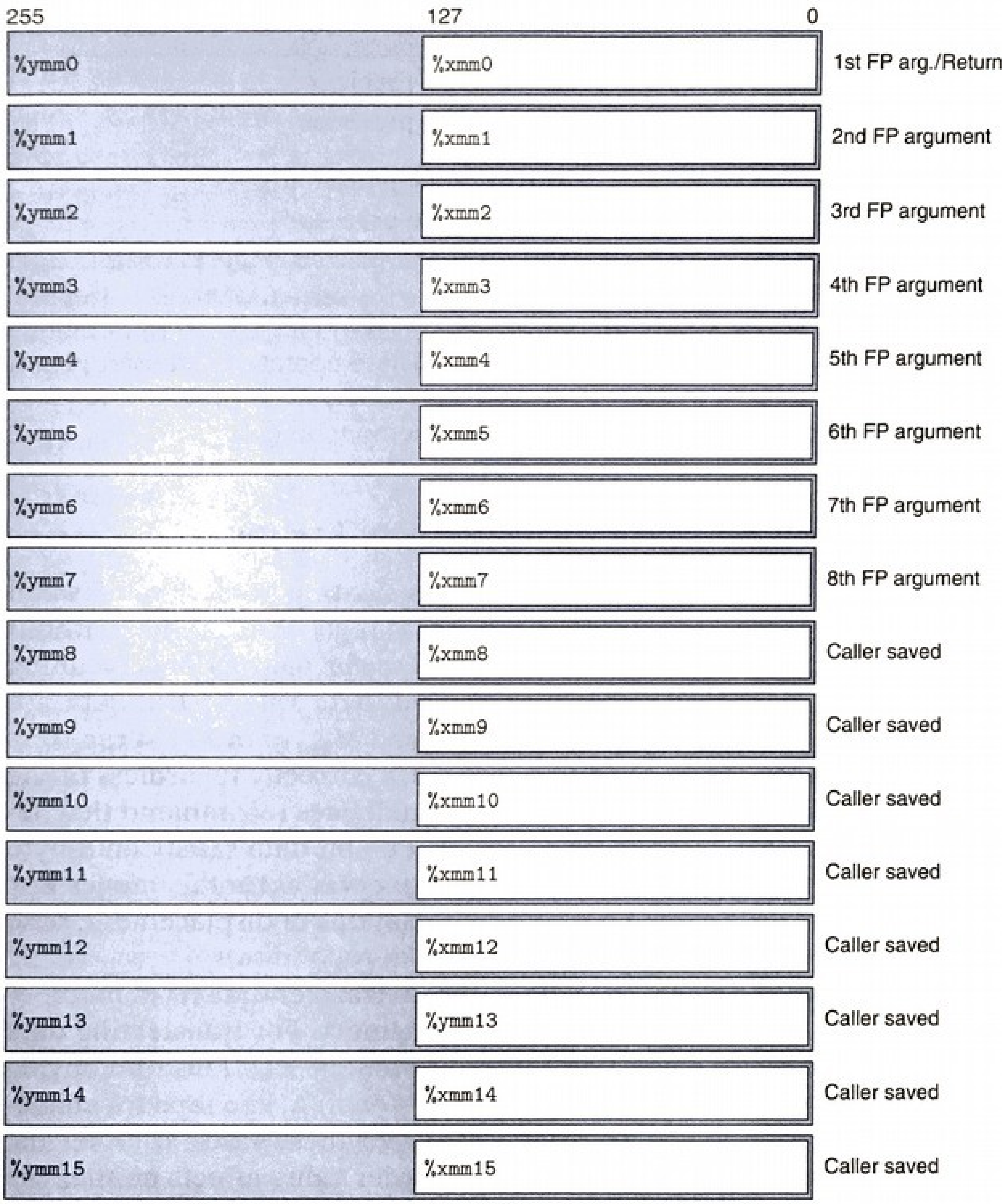
Floating-point code

- Single-Instruction, Multiple-Data (SIMD)
  - Performs the same operation on multiple elements
  - Also known as vector instructions
- Various floating-point SIMD instruction sets
  - MMX, SSE, SSE2, SSE3, SSE4, SSE5, AVX, AVX2
  - New extra-wide XMM (128-bit) or YMM (256-bit) registers for holding multiple elements
    - Floating-point arguments passed in `%xmm0-%xmm7`
    - Return value in `%xmm0`
    - All registers are caller-saved
  - New instructions for movement and arithmetic

SSE/AVX

- **Movement**
  - `movss / movsd`
  - `movaps / movapd`
- **Conversion**
  - `cvtsi2ss / cvtsi2sd`
  - `cvtss2si / cvtsd2si`
  - `cvtss2sd / cvtsd2ss`
- **Arithmetic**
  - `addss / addsd`
  - `addps / addpd`
  - `… (sub, mul, div,`
  - `      max, min, sqrt)`
  - `andps / andpd`
  - `xorps / xorpd`
- **Comparison**
  - `ucomiss / ucomisd`

(AVX has "v____" opcodes)

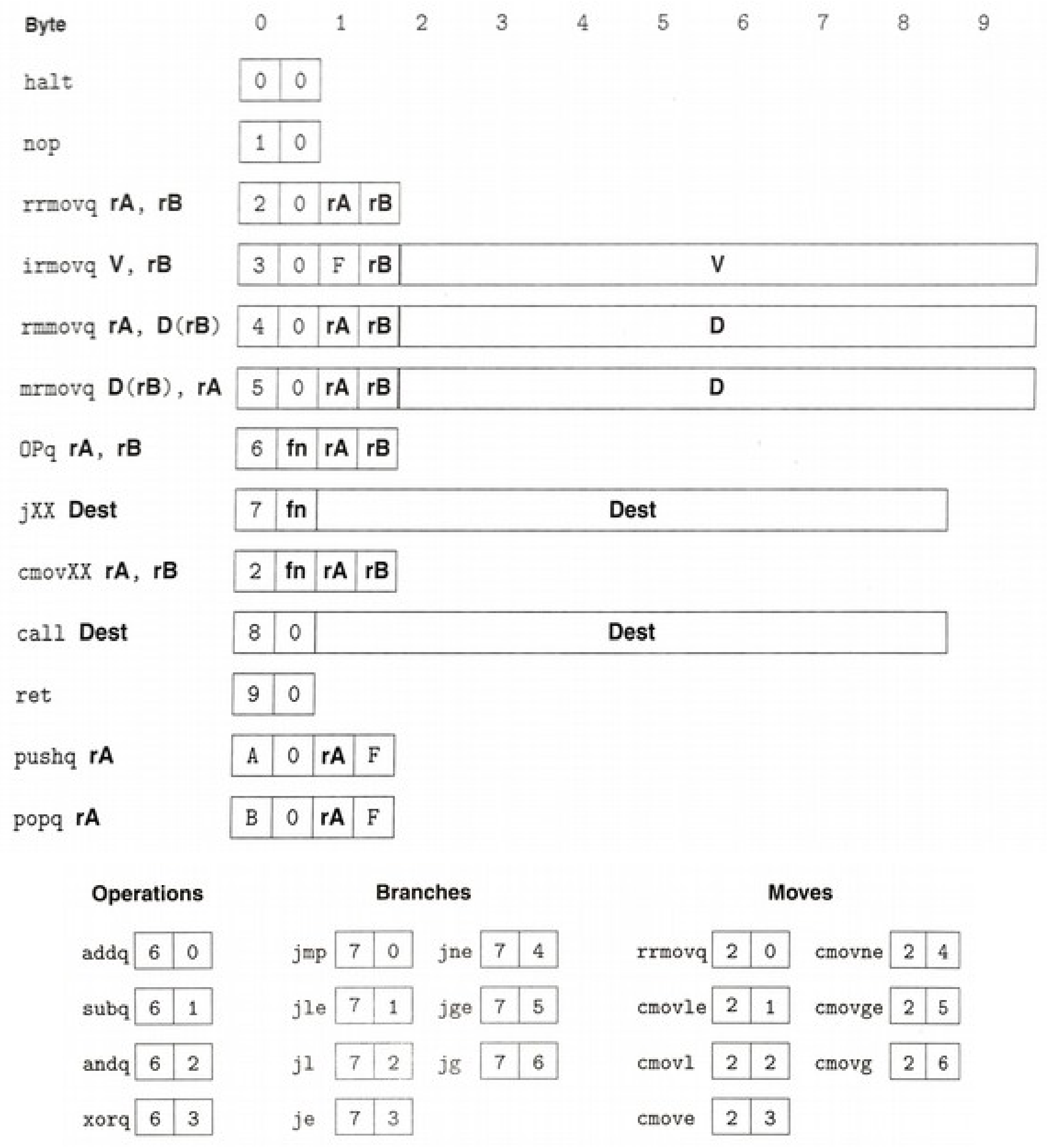


255	127 0	
%ymm0	%xmm0	1st FP arg./Return
%ymm1	%xmm1	2nd FP argument
%ymm2	%xmm2	3rd FP argument
%ymm3	%xmm3	4th FP argument
%ymm4	%xmm4	5th FP argument
%ymm5	%xmm5	6th FP argument
%ymm6	%xmm6	7th FP argument
%ymm7	%xmm7	8th FP argument
%ymm8	%xmm8	Caller saved
%ymm9	%xmm9	Caller saved
%ymm10	%xmm10	Caller saved
%ymm11	%xmm11	Caller saved
%ymm12	%xmm12	Caller saved
%ymm13	%ymm13	Caller saved
%ymm14	%xmm14	Caller saved
%ymm15	%xmm15	Caller saved

Bitwise operations in SSE/AVX

- Assembly instructions provide low-level access to floating-point numbers
  - Some numeric operations can be done more efficiently with simple bitwise operations
- AKA: Stupid Floating-Point Hacks™
  - Set to zero (value XOR value)
  - Absolute value (value AND `0x7fffffff`)
  - Additive inverse (value XOR `0x80000000`)
- Lesson: Information = Bits + Context
    - *(even if it wasn't the intended context!)*

Projects 3 & 4: Y86-64 ISA



Byte	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
rrmovq rA, rB	2	0	rA	rB						
irmovq V, rB	3	0	F	rB	V					
rmmovq rA, D(rB)	4	0	rA	rB	D					
mrmovq D(rB), rA	5	0	rA	rB	D					
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn	Dest							
cmovXX rA, rB	2	fn	rA	rB						
call Dest	8	0	Dest							
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

Operations

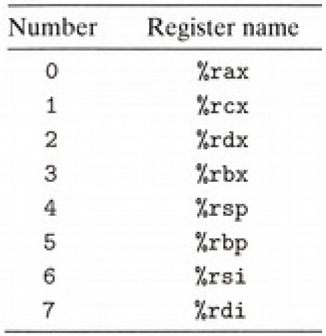
addq	6	0
subq	6	1
andq	6	2
xorq	6	3

Branches

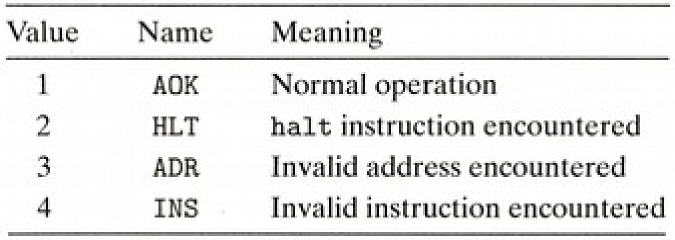
jmp	7	0	jne	7	4
jle	7	1	jge	7	5
jl	7	2	jg	7	6
je	7	3			

Moves

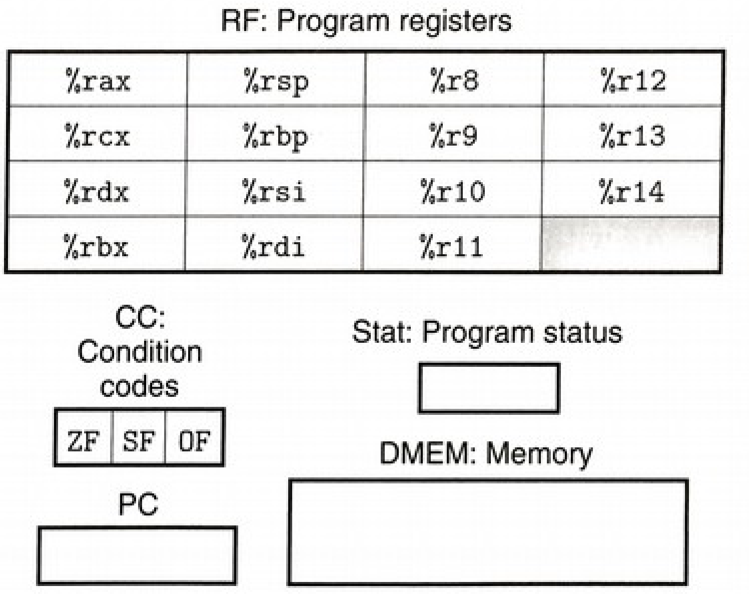
rrmovq	2	0	cmovne	2	4
cmovle	2	1	cmovge	2	5
cmovl	2	2	cmovg	2	6
cmove	2	3			



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



Value	Name	Meaning
1	AOK	Normal operation
2	HLT	halt instruction encountered
3	ADR	Invalid address encountered
4	INS	Invalid instruction encountered



RF: Program registers

%rax	%rsp	%r8	%r12
%rcx	%rbp	%r9	%r13
%rdx	%rsi	%r10	%r14
%rbx	%rdi	%r11	

Projects 3 & 4: Support Utilities

- New folder on stu: `/cs/students/cs261/f16/src/`**`y86`**
  - **`isa.pdf`**: Y86-64 reference sheet
  - **`y86`**: compiled reference solution to P3/P4
  - **`yas`**: Y86-64 assembler (.ys → .yo and .o)
  - **`yis`**: Y86-64 simulator (takes .yo)
  - **`ssim`**: CPU simulator (takes .yo)
  - **`simple.ys`**: sample Y86-64 assembly program

report any discrepancies!

- These will help with P3/P4: learn to use them!
  - "`yas <yourfile.ys>`" to assemble code into object files
- Hint: make shortcuts in your working folder for easier access
  - "`ln -s /cs/students/cs261/f16/src/y86/yas yas`"
  - "`ln -s /cs/students/cs261/f16/src/y86/y86 ref-y86`"

Projects 3 & 4: Hints & Thoughts

- Work incrementally
  - Gaps from C → B → A are much wider now
  - Remember that the grade is not the goal of the project
  - Start early enough to experiment and play
- Make your own examples to test with
  - Ignore our test suite while developing
  - Work until you think you've got the next grade, then test
- Remember the academic honesty policy
  - Working in the same space and sharing ideas is encouraged
  - Directly copying code is an honor code violation
  - This includes file transfers and cell phone photos

Course status

- We're nearly halfway through the semester
  - One exam, two projects, seven labs, ten quizzes
  - Crucial point in the semester
- We've learned a lot but still have a lot ahead
  - At this point you should have a good feel for how the course is going to go for the remainder of the semester
  - Keep in mind the withdrawal date is Oct 27
- I hope the course has been challenging but rewarding
  - Let me know how we're doing!

Good luck!

"[Coding] is a journey, not a destination."