RISC, CISC, and Assemblers

Hakim Weatherspoon
CS 3410, Spring 2011
Computer Science
Cornell University

See P&H Appendix B.1-2, and Chapters 2.8 and 2.12

Announcements

PA1 due *this* Friday

Work in pairs

Use your resources

- FAQ, class notes, book, Sections, office hours, newsgroup, CSUGLab

Prelims1: next Thursday, March 10th in class

- Material covered
  - Appendix C (logic, gates, FSMs, memory, ALUs)
  - Chapter 4 (pipelined [and non-pipeline] MIPS processor with hazards)
  - Chapters 2 and Appendix B (RISC/CISC, MIPS, and calling conventions)
  - Chapter 1 (Performance)
  - HW1, HW2, PA1, PA2
- Practice prelims are online in CMS
- Closed Book: cannot use electronic device or outside material
- We will start at 1:25pm sharp, so come early

Goals for Today

Instruction Set Architetures

- Arguments: stack-based, accumulator, 2-arg, 3-arg
- Operand types: load-store, memory, mixed, stacks, …
- Complexity: CISC, RISC

Assemblers

- assembly instructions
- psuedo-instructions
- data and layout directives
- executable programs

Instruction Set Architecture

ISA defines the permissible instructions

- MIPS: load/store, arithmetic, control flow, …
- ARM: similar to MIPS, but more shift, memory, & conditional ops
- VAX: arithmetic on memory or registers, strings, polynomial evaluation, stacks/queues, …
- Cray: vector operations, …
- x86: a little of everything

One Instruction Set Architecture

Toy example: subleq a, b, target

Mem[b] = Mem[b] – Mem[a]
then if (Mem[b] <= 0) goto target
else continue with next instruction

clear a == subleq a, a, pc+4

jmp c == subleq Z, Z, c

add a, b == subleq a, Z, pc+4;
subleq Z, b, pc+4;
subleq Z, Z, pc+4

$a = a - 0$
$a = 0$ $\quad Z = 0$
$Z = 0 - a$
$= -a$
$b = b - Z$
$= b - (-a)$
$b = b + a$
$Z = Z - Z = 0$

PDP-8

Not-a-toy example: PDP-8

One register: AC - Accumulator

Eight basic instructions:

```
AND a    # AC = AC & MEM[a]
TAD a    # AC = AC + MEM[a]
ISZ a    # if (!++MEM[a]) skip next
DCA a    # MEM[a] = AC; AC = 0
JMS a    # jump to subroutine (e.g. jump and link)
JMP a    # jump to MEM[a]
IOT x    # input/output transfer
OPR x    # misc operations on AC
```

Stack Based

Stack machine

- data *stack* in memory, *stack pointer* register
- Operands popped/pushed as needed
  add

[Java Bytecode, PostScript, odd CPUs, some x86]

Tradeoffs:

Simple/elegant
Small ISA

depends on mem perf
Small args

Accumulator Based

Accumulator machine

- Results usually put in dedicated accumulator register

  add b

  store b

[Some x86]

EAX on x86 = acc

Tradeoffs:

Load-Store

Load/store (register-register) architecture

- computation only between registers

[MIPS, some x86]

Tradeoffs:

RISC = Reduced Inst Set Computer

3-args ⇒ slightly larger inst
more inst
(possibly) higher clock rate

Axes

Axes:

- Arguments: stack-based, accumulator, 2-arg, 3-arg
- Operand types: load-store, memory, mixed, stacks, …
- Complexity: CISC, RISC

Complex Instruction Set Computers

People programmed in assembly and machine code!

- Needed as many addressing modes as possible
- Memory was (and still is) slow

CPUs had relatively few registers

- Register's were more "expensive" than external mem
- Large number of registers requires many bits to index

Memories were small

- Encoraged highly encoded microcodes as instructions
- Variable length instructions, load/store, conditions, etc

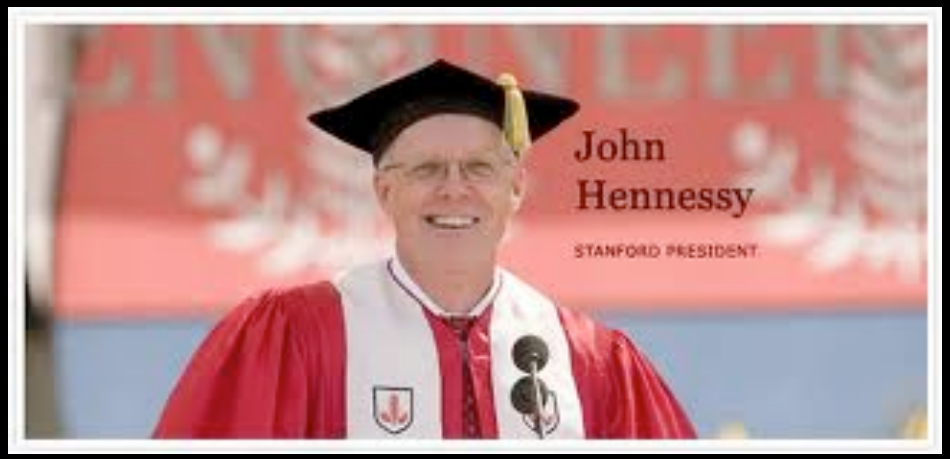
Reduced Instruction Set Computer

Dave Patterson

- RISC Project, 1982
- UC Berkeley
- RISC-I: ½ transtisters & 3x faster
- Influences: Sun SPARC, namesake of industry

John L. Hennessy

- MIPS, 1981
- Stanford
- Simple pipelining, keep full
- Influences: MIPS computer system, PlayStation, Nintendo



Complexity

MIPS = Reduced Instruction Set Computer (RISC)

- ≈ 200 instructions, 32 bits each, 3 formats
- all operands in registers
  - almost all are 32 bits each
- ≈ 1 addressing mode: Mem[reg + imm]

x86 = Complex Instruction Set Computer (CISC)

- > 1000 instructions, 1 to 15 bytes each
- operands in dedicated registers, general purpose registers, memory, on stack, …
  - can be 1, 2, 4, 8 bytes, signed or unsigned
- 10s of addressing modes
  - e.g. Mem[segment + reg + reg*scale + offset]

RISC vs CISC

RISC Philosophy

Regularity & simplicity

Leaner means faster

Optimize the common case

energy

embedded Systems

CISC Rebuttal

Compilers can be smart

Transistors are plentiful

Legacy is important

Code size counts

Micro-code!

desktop Servers

Goals for Today

Instruction Set Architetures

- Arguments: stack-based, accumulator, 2-arg, 3-arg
- Operand types: load-store, memory, mixed, stacks, …
- Complexity: CISC, RISC

Assemblers

- assembly instructions
- psuedo-instructions
- data and layout directives
- executable programs

Examples

```
    ...
T:  ADDI r4, r0, -1
    BEQ r3, r0, B
    ADDI r4, r4, 1
    LW r3, 0(r3)
    J T
    NOP
B:  ...
```

```
    ...
    JAL L
    nop
    nop
L:  LW r5, 0(r31)
    ADDI r5, r5, 1
    SW r5,  0(r31)
    ...
```

cs3410 Recap/Quiz

C

compiler

```
int x = 10;
x = 2 * x + 15;
```

MIPS assembly

assembler

```
addi r5, r0, 10
muli r5, r5, 2
addi r5, r5, 15
```

machine code

```
00100000000001010000000000001010
00000000000001010010100001000000
00100000101001010000000000001111
```

CPU

Circuits

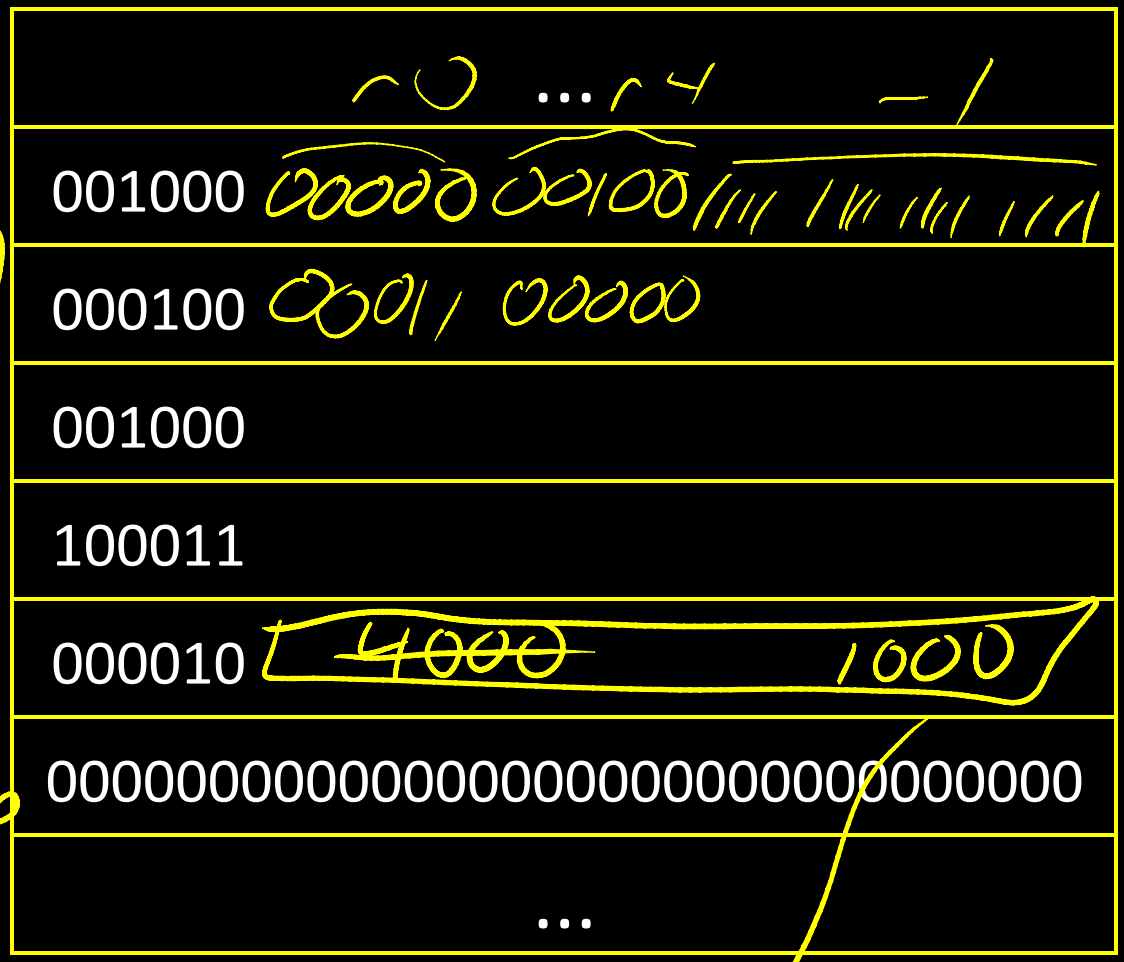
Gates

Transistors

Silicon

Example 1

```
4000..
T: ADDI r4,r0,-1
   BEQ r3, r0, B
   ADDI r4,r4, 1
   LW r3, 0(r3)
   J T
   NOP
B: ...
```

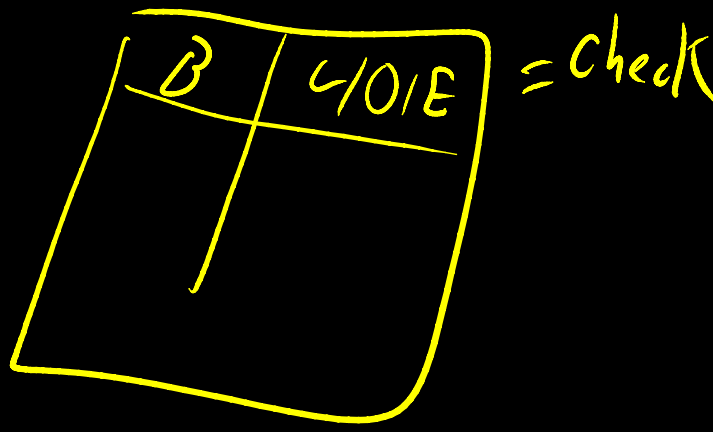


References

Q: How to resolve labels into offsets and addresses?

A: Two-pass assembly

- 1st pass: lay out instructions and data, and build a *symbol table* (mapping labels to addresses) as you go
- 2nd pass: encode instructions and data in binary, using symbol table to resolve references



30 =

Example 2

```
   ...
   JAL L
   nop
   nop
L: LW r5, 0(r31)
   ADDI r5,r5,1
   SW r5, 0(r31)
   ...
```

...
00100000001000000000000000000100
00000000000000000000000000000000
00000000000000000000000000000000
10001111111001010000000000000000
00100000101001010000000000000001
00000000000000000000000000000000
...

counter=0

Von Neuman Machine = mixes code & dat

① seperate code & data

② L: word "0"

.text .data

Example 2 (better)

```
.text 0x00400000 # code segment
    ...
    ORI r4, r0, counter
    LW r5, 0(r4)
    ADDI r5, r5, 1
    SW r5,  0(r4)
    ...
.data 0x10000000 # data segment
counter:
    .word 0
```

LA = load address

LA = Pseudo inst

LI - load Imm

Lessons

Lessons:

- Mixed data and instructions (von Neumann)
- … but best kept in separate *segments*
- Specify layout and data using *assembler directives*
- Use *pseudo-instructions*

Pseudo-Instructions

Pseudo-Instructions

NOP # do nothing

MOVE reg, reg # copy between regs

LI reg, imm # load immediate (up to 32 bits)

LA reg, label # load address (32 bits)

B label # unconditional branch

BLT reg, reg, label # branch less than

Assembler

Assembler:

assembly instructions

+ psuedo-instructions

+ data and layout directives

= executable program

Slightly higher level than plain assembly

e.g: takes care of delay slots

(will reorder instructions or insert nops)

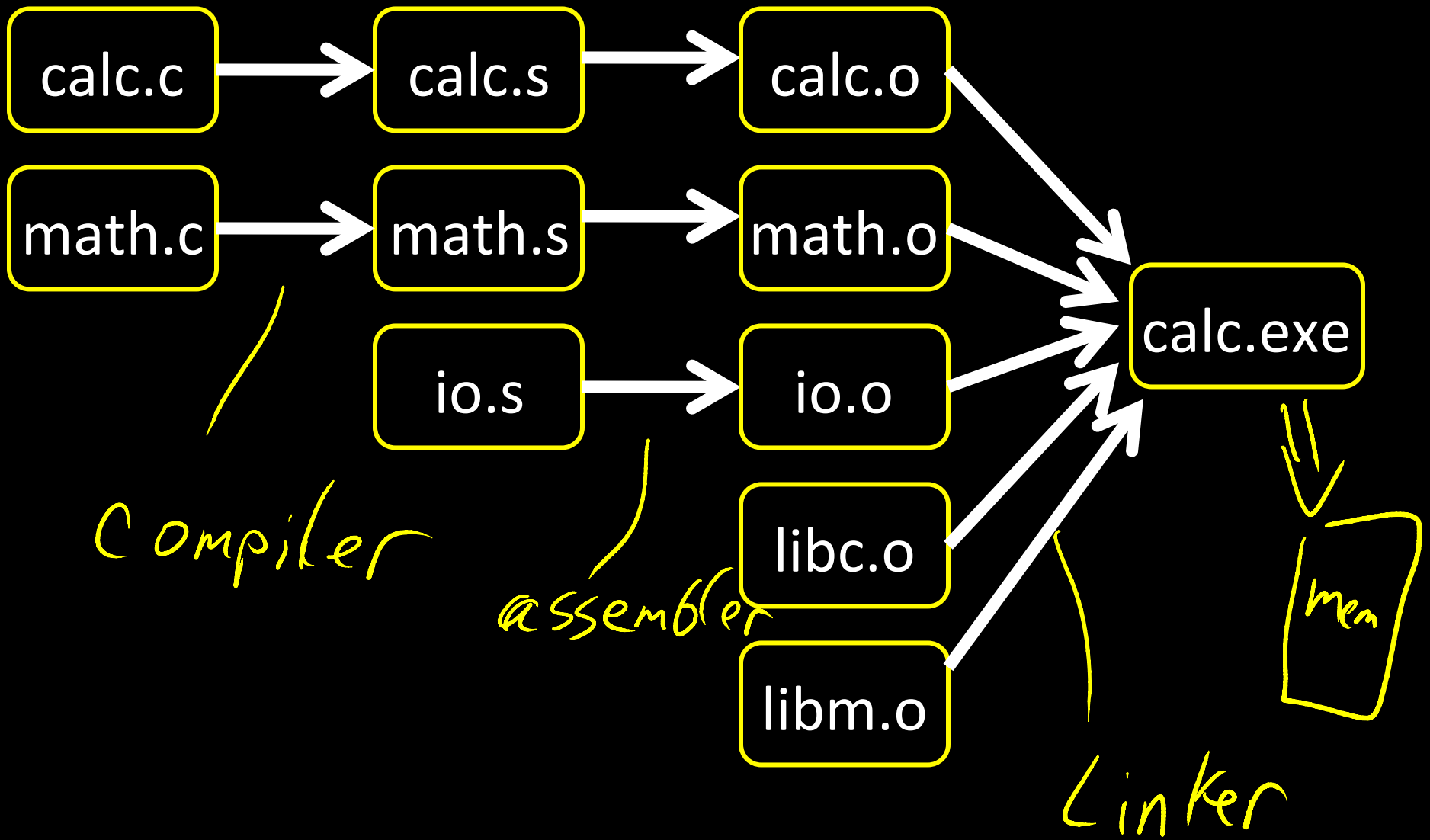
Motivation

Q: Will I program in assembly?

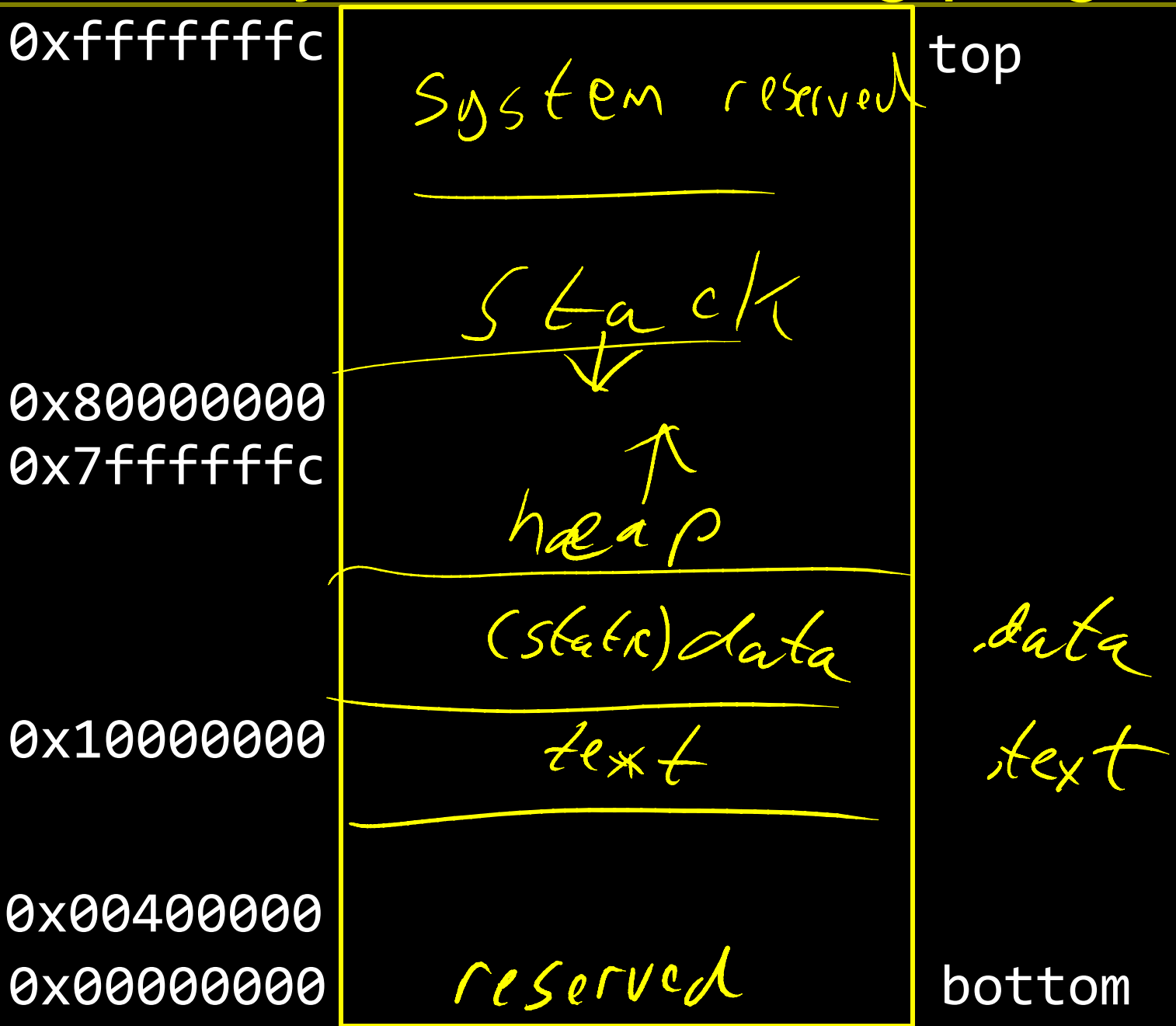
A: I do...

- For kernel hacking, device drivers, GPU, etc.
- For performance (but compilers are getting better)
- For highly time critical sections
- For hardware without high level languages
- For new & advanced instructions: rdtsc, debug registers, performance counters, synchronization, ...

Stages


calc.c
calc.s
calc.o
math.c
math.s
math.o
io.s
io.o
calc.exe
libc.o
libm.o
Compiler
assembler
Linker
mem

Anatomy of an executing program



Example program

calc.c

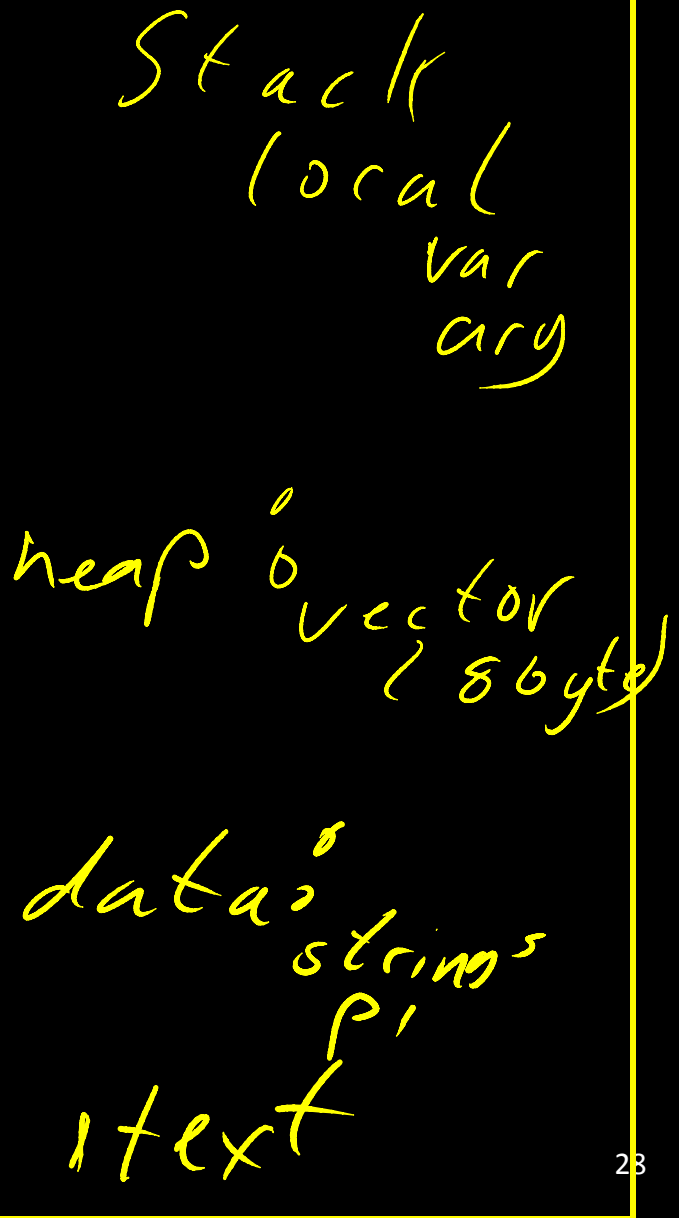
```
vector v = malloc(8);
v->x = prompt(“enter x”);
v->y = prompt(“enter y”);
int c = pi + tnorm(v);
print(“result”, c);
```

math.c

```
int tnorm(vector v) {
 return abs(v->x)+abs(v->y);
}
```

lib3410.o

```
  global variable: pi
  entry point: prompt
  entry point: print
  entry point: malloc
```



math.s

math.c

```
int abs(x) {
  return x < 0 ? -x : x;
}
int tnorm(vector v) {
 return abs(v->x)+abs(v->y);
}
```

abs:

```
  # arg in r3, return address in r31
  # leaves result in r3
```

.global tnorm

tnorm:

```
  # arg in r4, return address in r31
  # leaves result in r4
```

calc.s

calc.c

```
vector v = malloc(8);
v->x = prompt("enter x");
v->y = prompt("enter y");
int c = pi + tnorm(v);
print("result", c);
```

```
.data
str1: .asciiz "enter x"
str2: .asciiz "enter y"
str3: .asciiz "result"
.text
    .extern prompt
    .extern print
    .extern malloc
    .extern tnorm
    .global dostuff
```

```
dostuff:
  # no args, no return value, return addr in r31
  MOVE r30, r31
  LI r3, 8          # call malloc: arg in r3, ret in r3
  JAL malloc
  MOVE r6, r3 # r6 holds v
  LA r3, str1     # call prompt: arg in r3, ret in r3
  JAL prompt
  SW r3, 0(r6)
  LA r3, str2     # call prompt: arg in r3, ret in r3
  JAL prompt
  SW r3, 4(r6)
  MOVE r4, r6 # call tnorm: arg in r4, ret in r4
  JAL tnorm
  LA r5, pi
  LW r5, 0(r5)
  ADD r5, r4, r5
  LA r3, str3     # call print: args in r3 and r4
  MOVE r4, r5
  JAL print
  JR r30
```

Next time

Calling Conventions!

PA1 due Friday

Prelim1 Next Thursday, in class