

RISC, CISC, and Assemblers

Hakim Weatherspoon

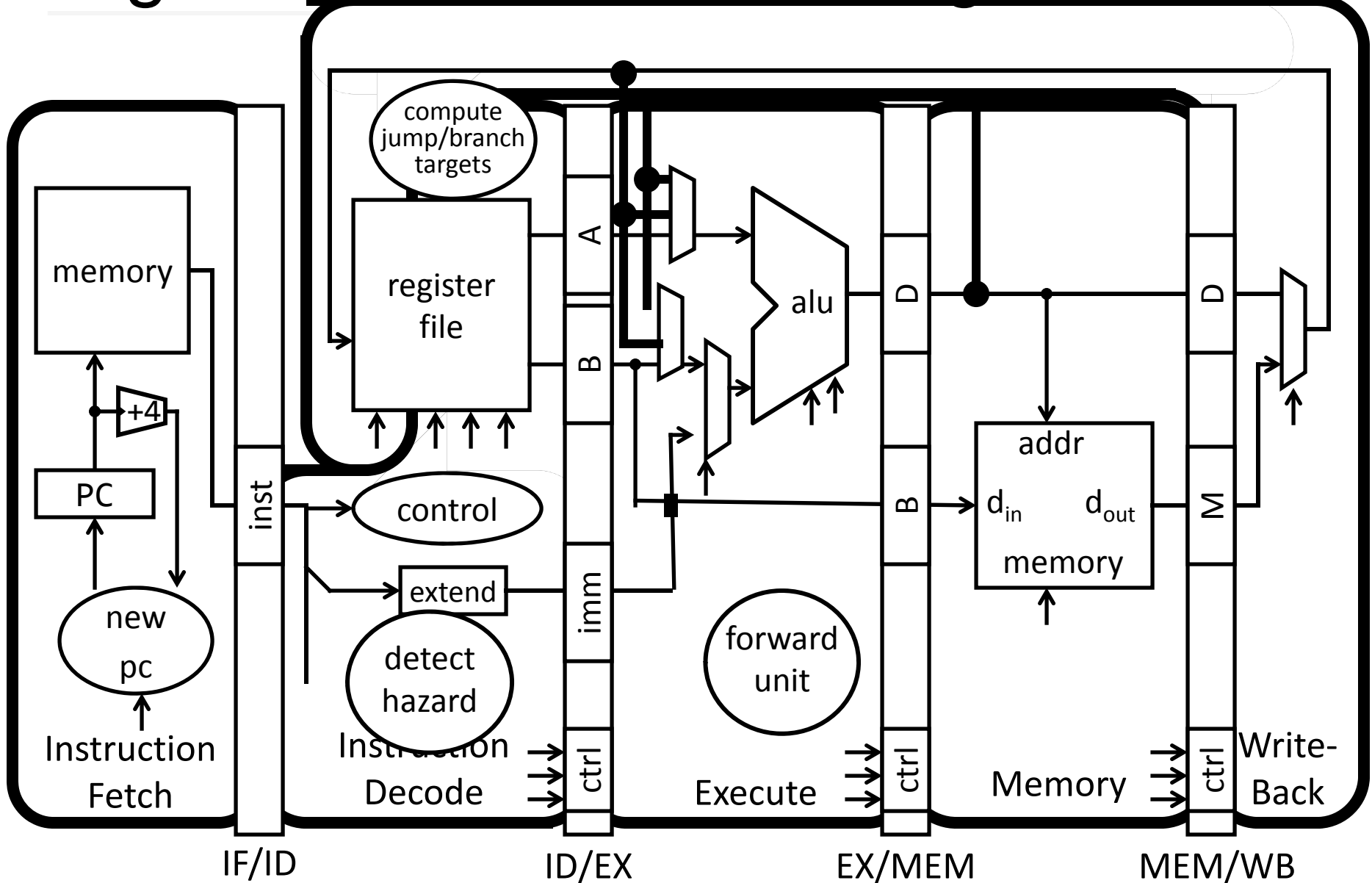
CS 3410, Spring 2012

Computer Science

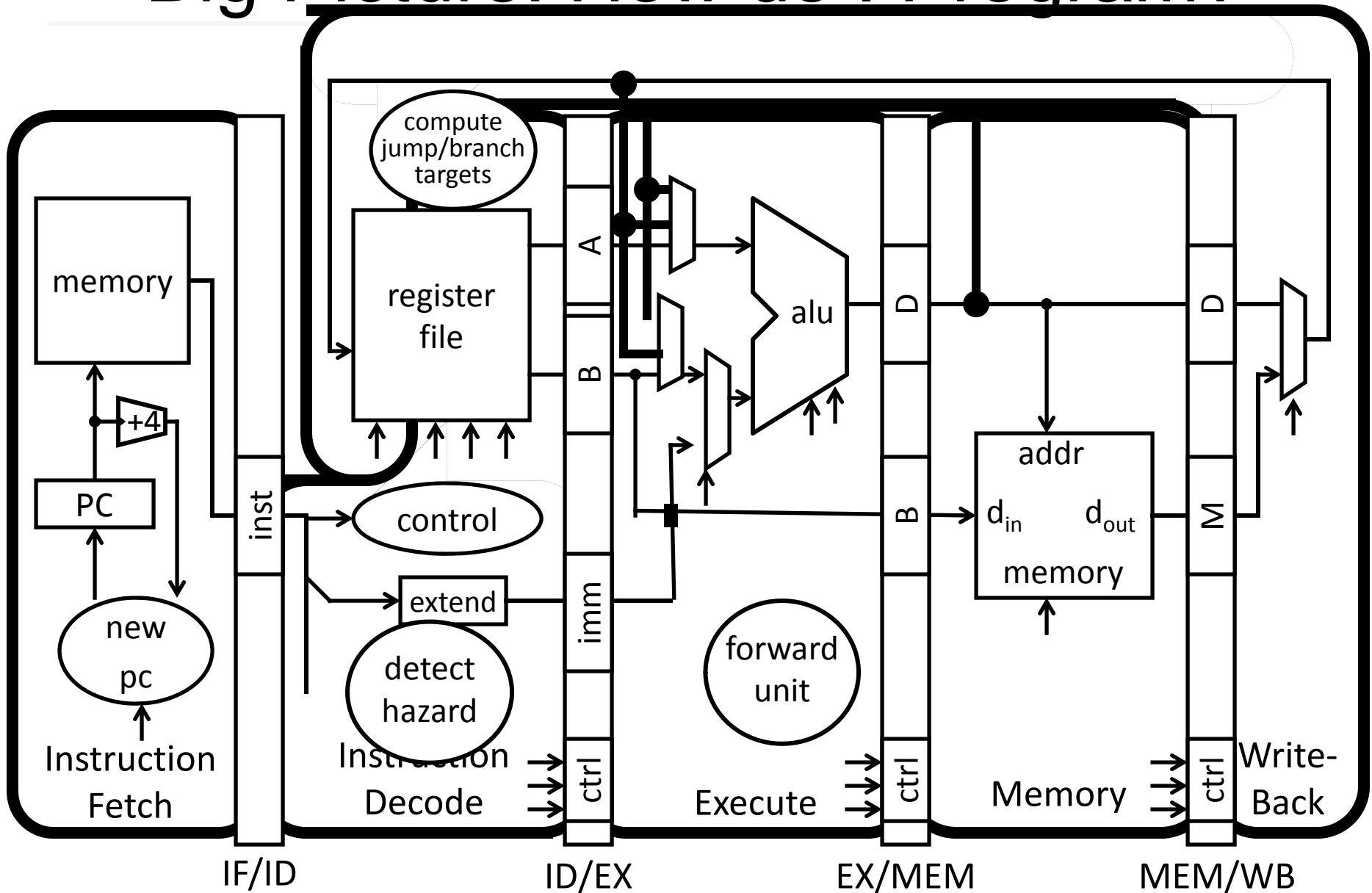
Cornell University

See P&H Appendix B.1-2, and Chapters 2.8 and 2.12; als 2.16 and 2.17

Big Picture: Understanding Tradeoffs



Big Picture: How do I Program?



Goals for Today

Instruction Set Architectures

- ISA Variations
- Complexity: CISC, RISC

Assemblers

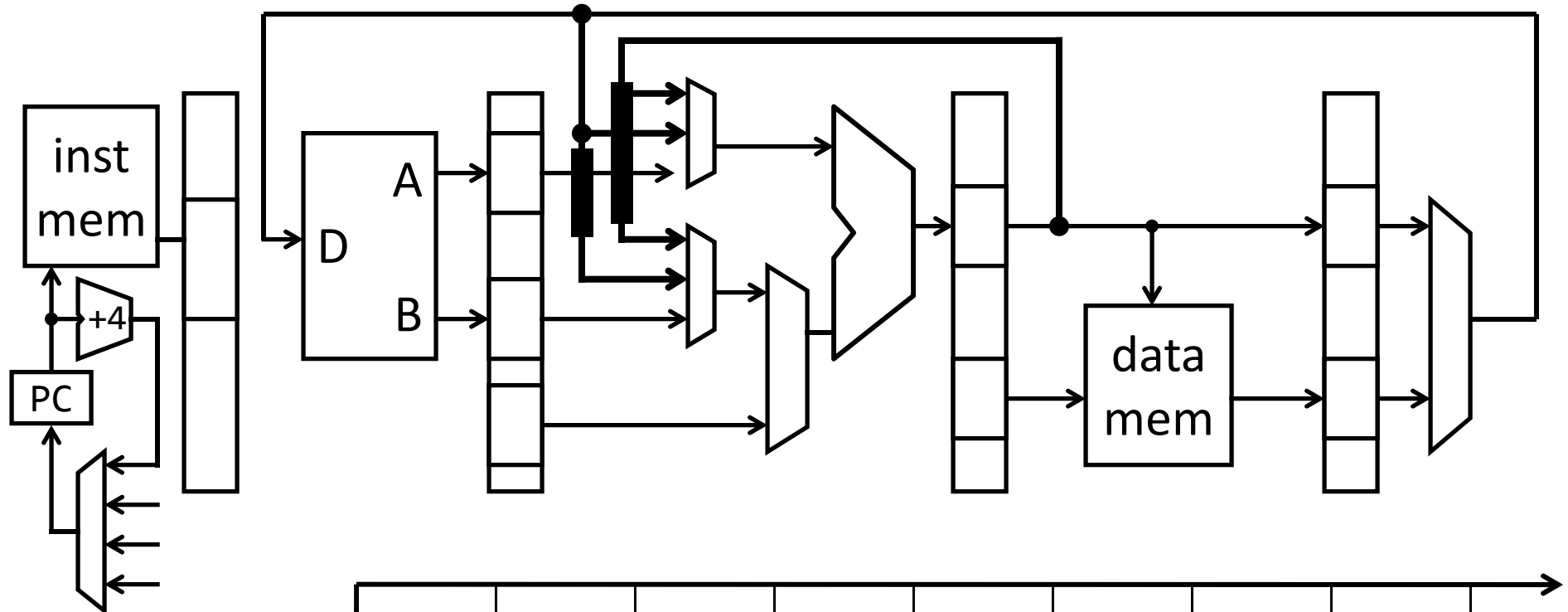
Translate symbolic instructions to binary machine code

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

Next Time

- Program Structure and Calling Conventions

Recall: Control Hazards



beq r1, r2, L	IF	ID	Ex	M	WB				
add r3, r0, r3									
sub r5, r4, r6									
L: or r3, r2, r4			IF	ID	Ex	M	WB		

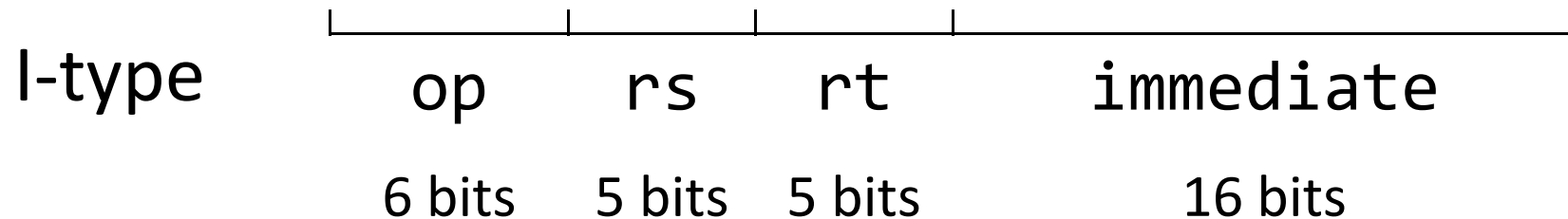
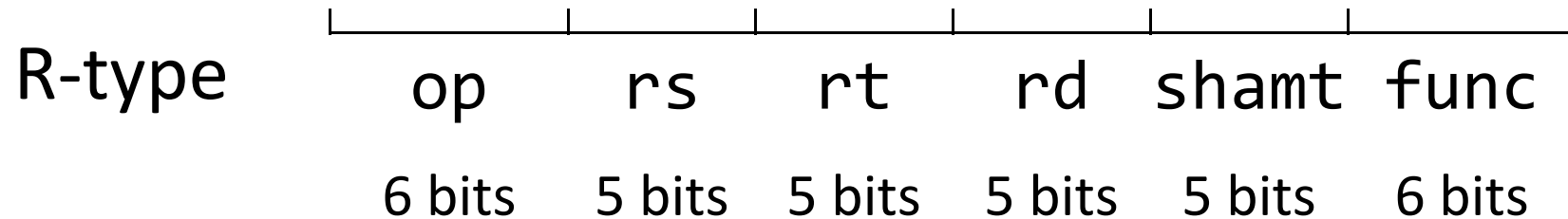
ISA Variations: Conditional Instructions

- while(i != j) {
- if (i > j)
- i -= j;
- else
- j -= i;
- }

```
LOOP: CMP Ri, Rj           // set condition "NE" if (i != j)
                               // "GT" if (i > j),
                               // or "LT" if (i < j)
        SUBGT Ri, Ri, Rj    // if "GT" (greater than), i = i-j;
        SUBLT Rj, Rj, Ri    // if "LT" (less than), j = j-i;
        BNE loop           // if "NE" (not equal), then loop
```

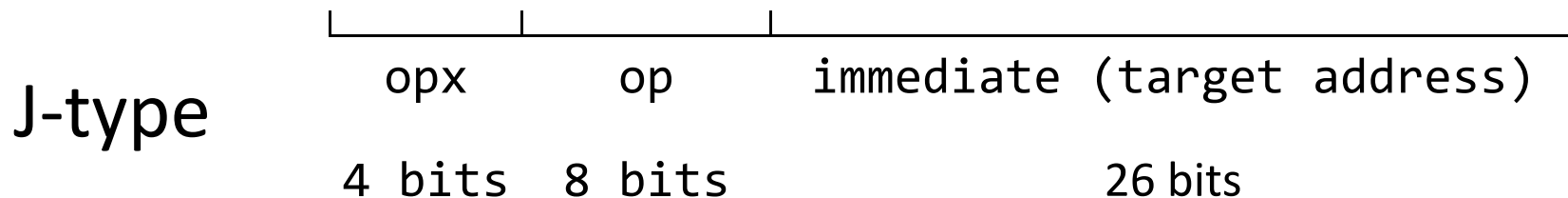
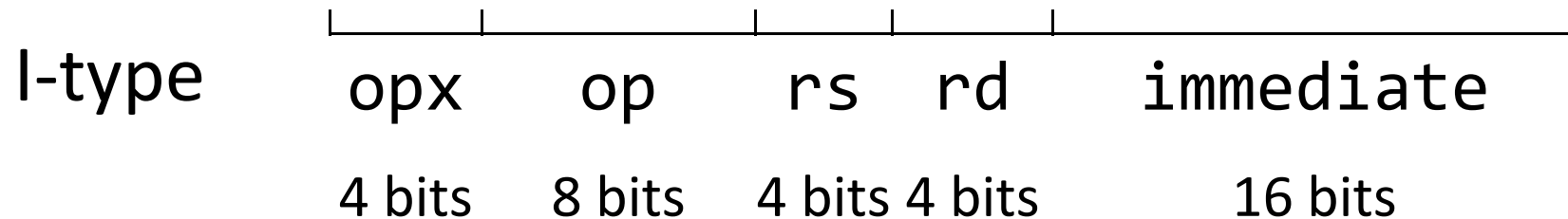
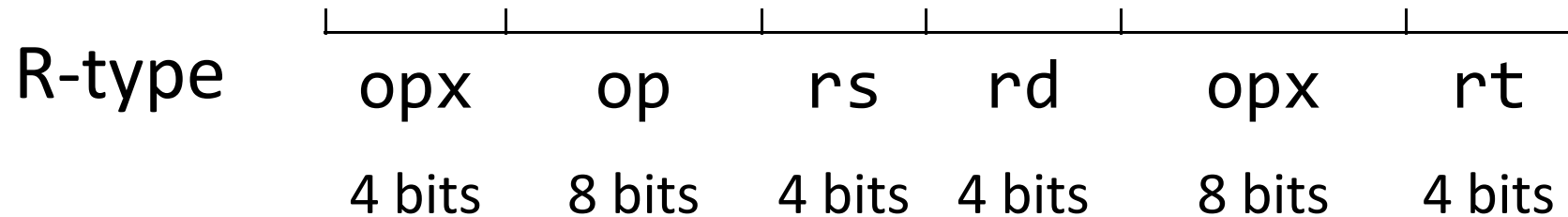
MIPS instruction formats

All MIPS instructions are 32 bits long, has 3 formats



ARM instruction formats

All MIPS instructions are 32 bits long, has 3 formats



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

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 transistors & 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)

- \approx 200 instructions, 32 bits each, 3 formats
- all operands in registers
 - almost all are 32 bits each
- \approx 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

CISC Rebuttal

Compilers can be smart

Transistors are plentiful

Legacy is important

Code size counts

Micro-code!

ARMDroid vs WinTel

- Android OS on ARM processor



- Windows OS on Intel (x86) processor



Administrivia

Project1 (PA1) due next Monday, March 5th

- Continue working diligently. Use design doc momentum

Save your work!

- **Save often.** Verify file is non-zero. Periodically save to Dropbox, email.
- Beware of MacOSX 10.5 (leopard) and 10.6 (snow-leopard)

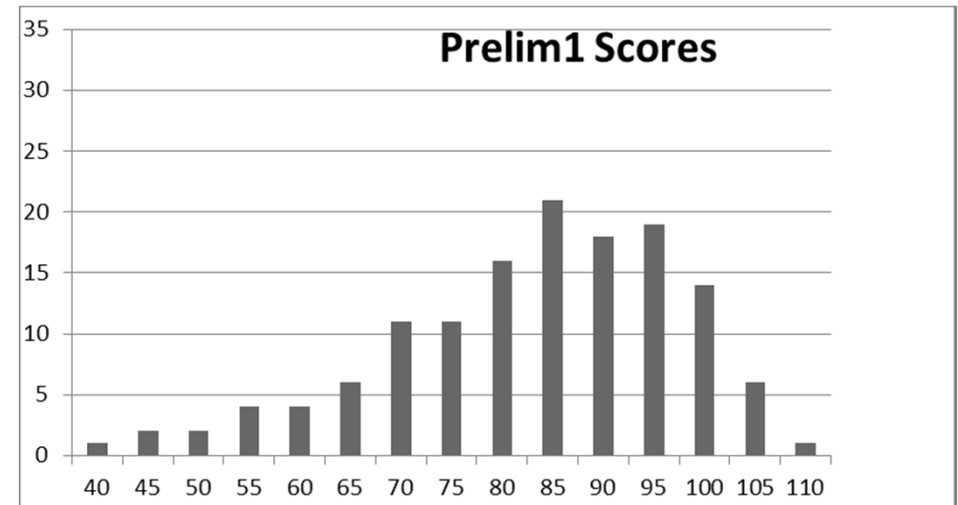
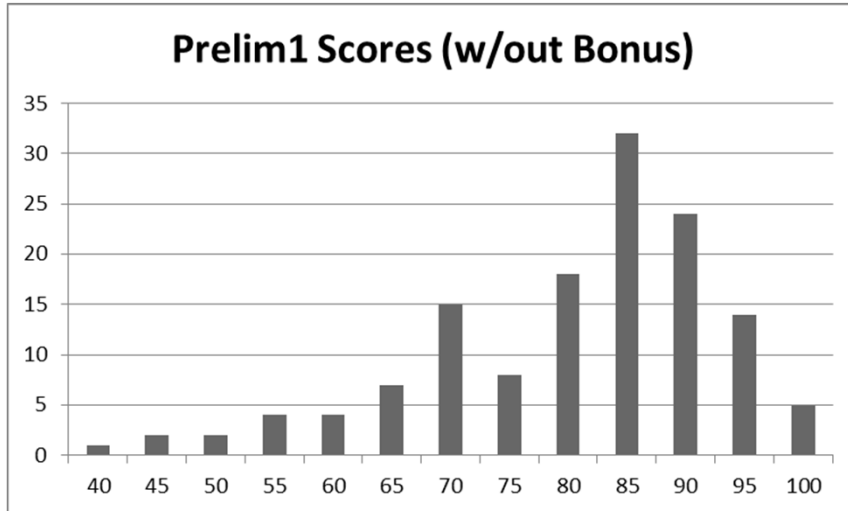
Use your resources

- Lab Section, Piazza.com, Office Hours, Homework Help Session,
- Class notes, book, Sections, CSUGLab

Administrivia

Prelim1 results

- Mean 80 (without bonus 78), standard deviation 15



- Prelims available in Upson 360 after today
- Regrade requires written request
 - ***Whole test is regraded***

Goals for Today

Instruction Set Architectures

- ISA Variations
- Complexity: CISC, RISC

Assemblers

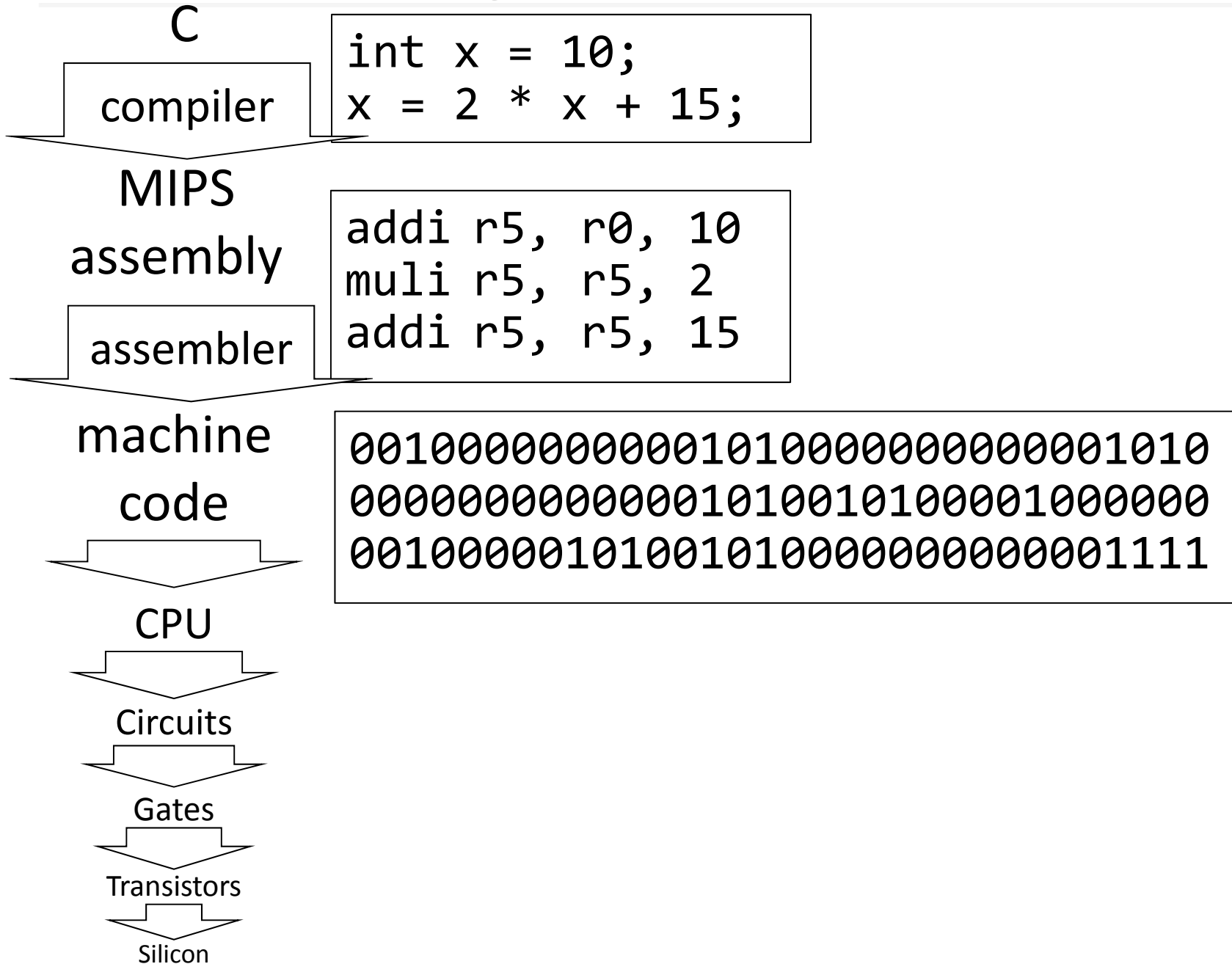
Translate symbolic instructions to binary machine code

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

Next Time

- Program Structure and Calling Conventions

How do I program a MIPS processor?



Assembler

Translates text *assembly language* to binary machine code

Input: a text file containing MIPS instructions in human readable form

Output: an object file (.o file in Unix, .obj in Windows) containing MIPS instructions in executable form

Assembly Language

Assembly language is used to specify programs at a low-level

What does a program consist of?

- MIPS instructions
- Program data (strings, variables, etc)

MIPS Instruction Types

Arithmetic/Logical

- ADD, ADDU, SUB, SUBU, AND, OR, XOR, NOR, SLT, SLTU
- ADDI, ADDIU, ANDI, ORI, XORI, LUI, SLL, SRL, SLLV, SRLV, SRAV, SLTI, SLTIU
- MULT, DIV, MFLO, MTLO, MFHI, MTHI

Memory Access

- LW, LH, LB, LHU, LBU, LWL, LWR
- SW, SH, SB, SWL, SWR

Control flow

- BEQ, BNE, BLEZ, BLTZ, BGEZ, BGTZ
- J, JR, JAL, JALR, BEQL, BNEL, BLEZL, BGTZL

Special

- LL, SC, SYSCALL, BREAK, SYNC, COPROC

Assembling Programs

```
.text
.ent main
main: la $4, Larray
      li $5, 15
      ...
      li $4, 0
      jal exit
.end main
.data
Larray:
      .long 51, 491, 3991
```

Assembly files consist of a mix of

+ instructions

+ pseudo-instructions

+ assembler (data/layout) directives

(Assembler lays out binary values
in memory based on directives)

Assembled to an Object File

- Header
- Text Segment
- Data Segment
- Relocation Information
- Symbol Table
- Debugging Information

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

Example 2

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

...
00100000000100000000000000000100
00000000000000000000000000000000
00000000000000000000000000000000
10001111110010100000000000000000
00100000101001010000000000000001
00000000000000000000000000000000
...

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
```

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

Lessons:

- Von Neumann architecture mixes data and instructions
- ... but best kept in separate *segments*
- Specify layout and data using *assembler directives*
- Use *pseudo-instructions*

Assembler

Assembler:

- assembly instructions

- + pseudo-instructions

- + data and layout directives

- = executable program

Slightly higher level than plain assembly

- e.g: takes care of delay slots

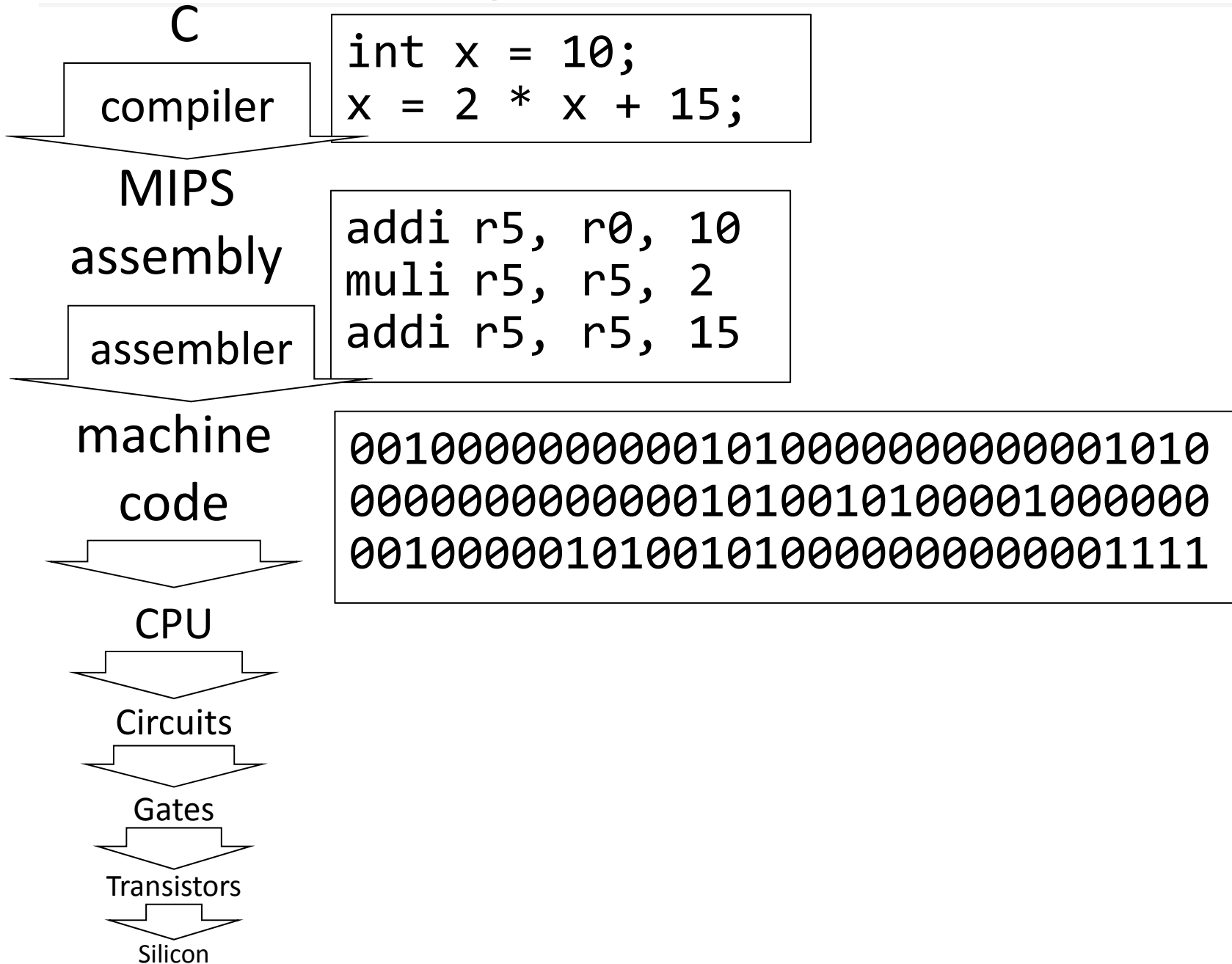
- (will reorder instructions or insert nops)

Will I program in assembly?

A: I do...

- For CS 3410 (and some CS 4410/4411)
- 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, ...

How do I program a MIPS processor?



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);
```

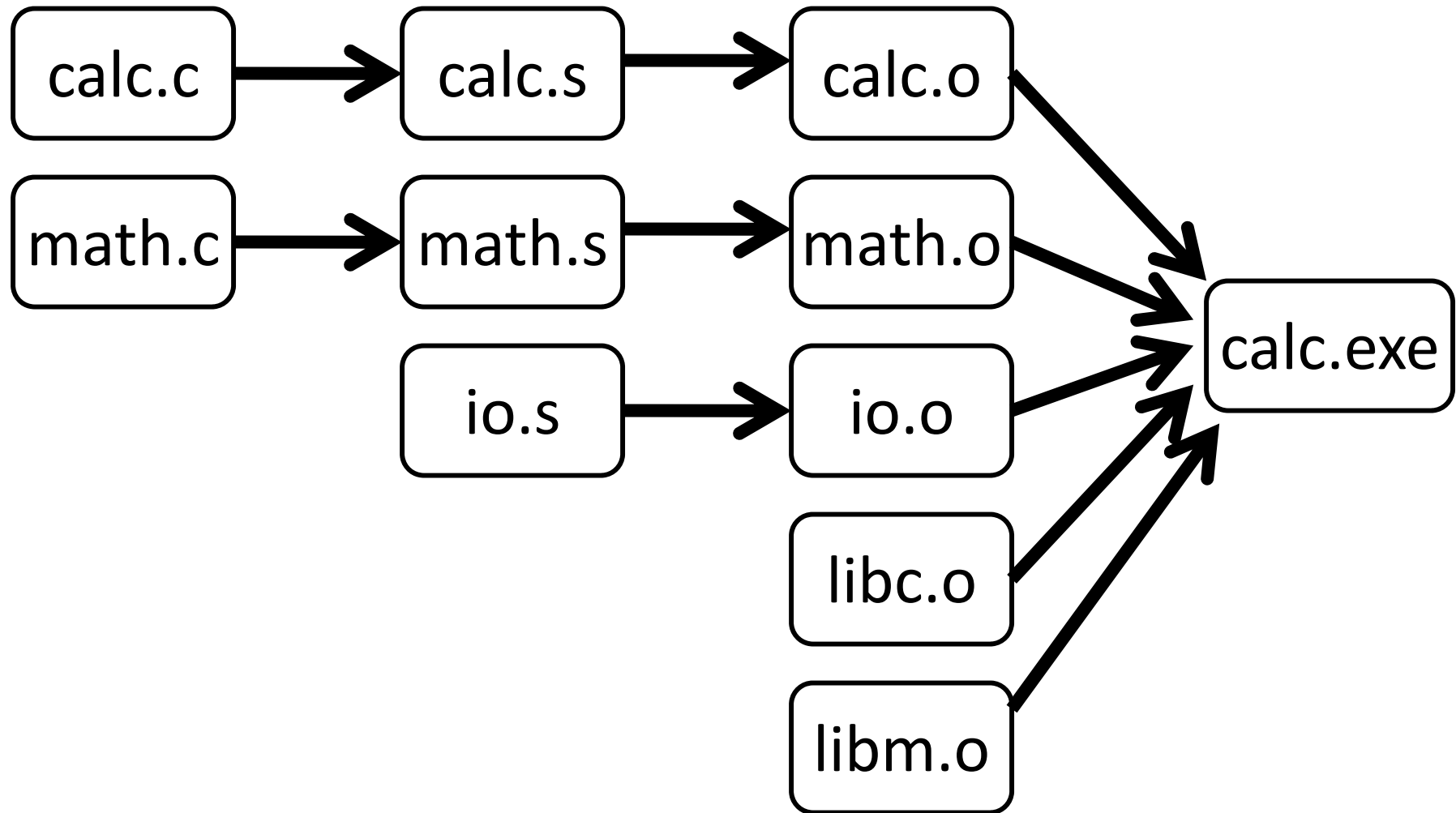
math.c

```
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
```


Stages



Anatomy of an executing program

0xffffffffc

top

0x80000000

0x7fffffff

0x10000000

0x00400000

0x00000000

bottom

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

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

How do we coordinate use of registers?

Calling Conventions!

PA1 due Monday