RISC, CISC, and Assemblers

Hakim Weatherspoon

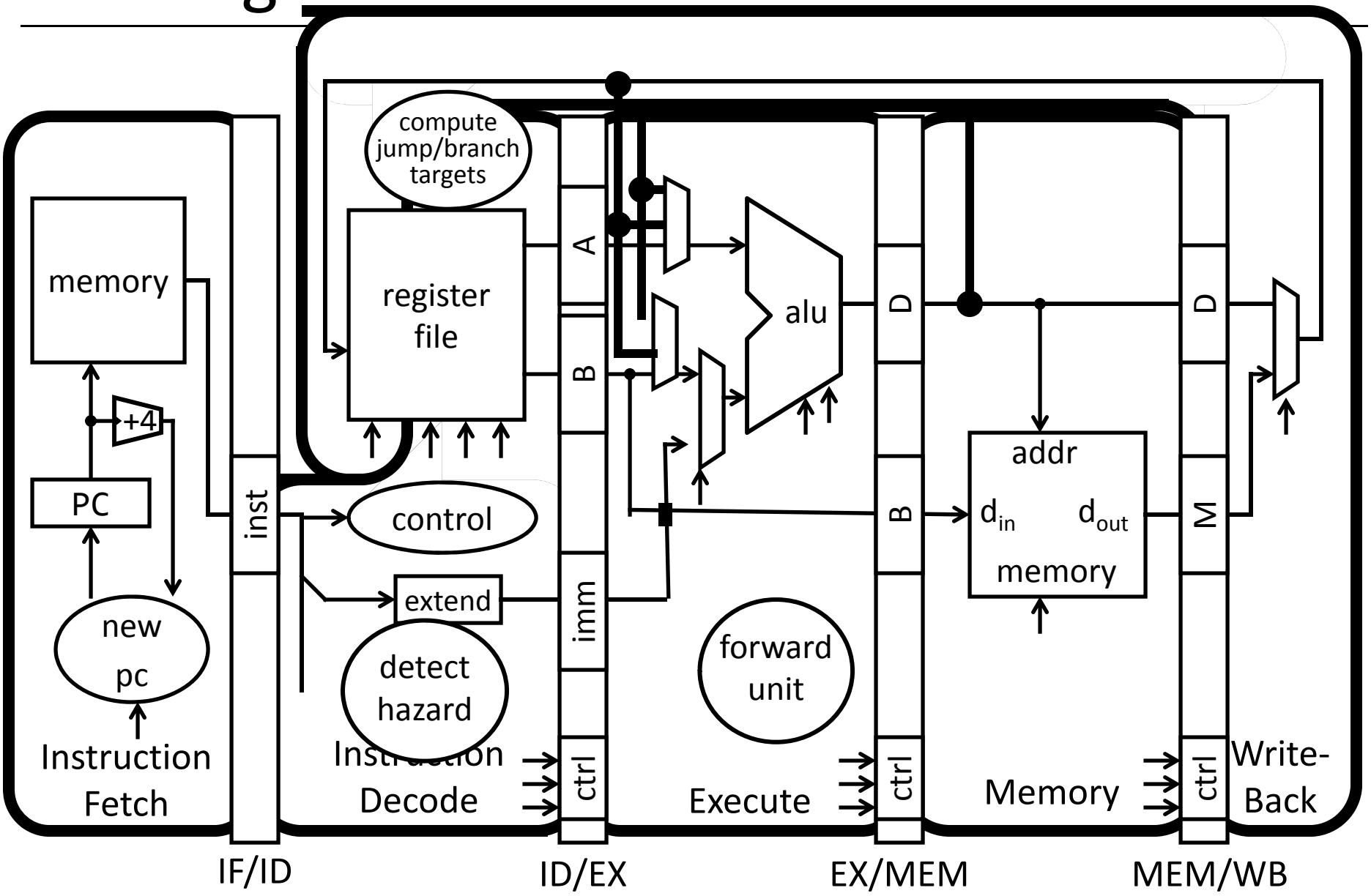
CS 3410, Spring 2013

Computer Science

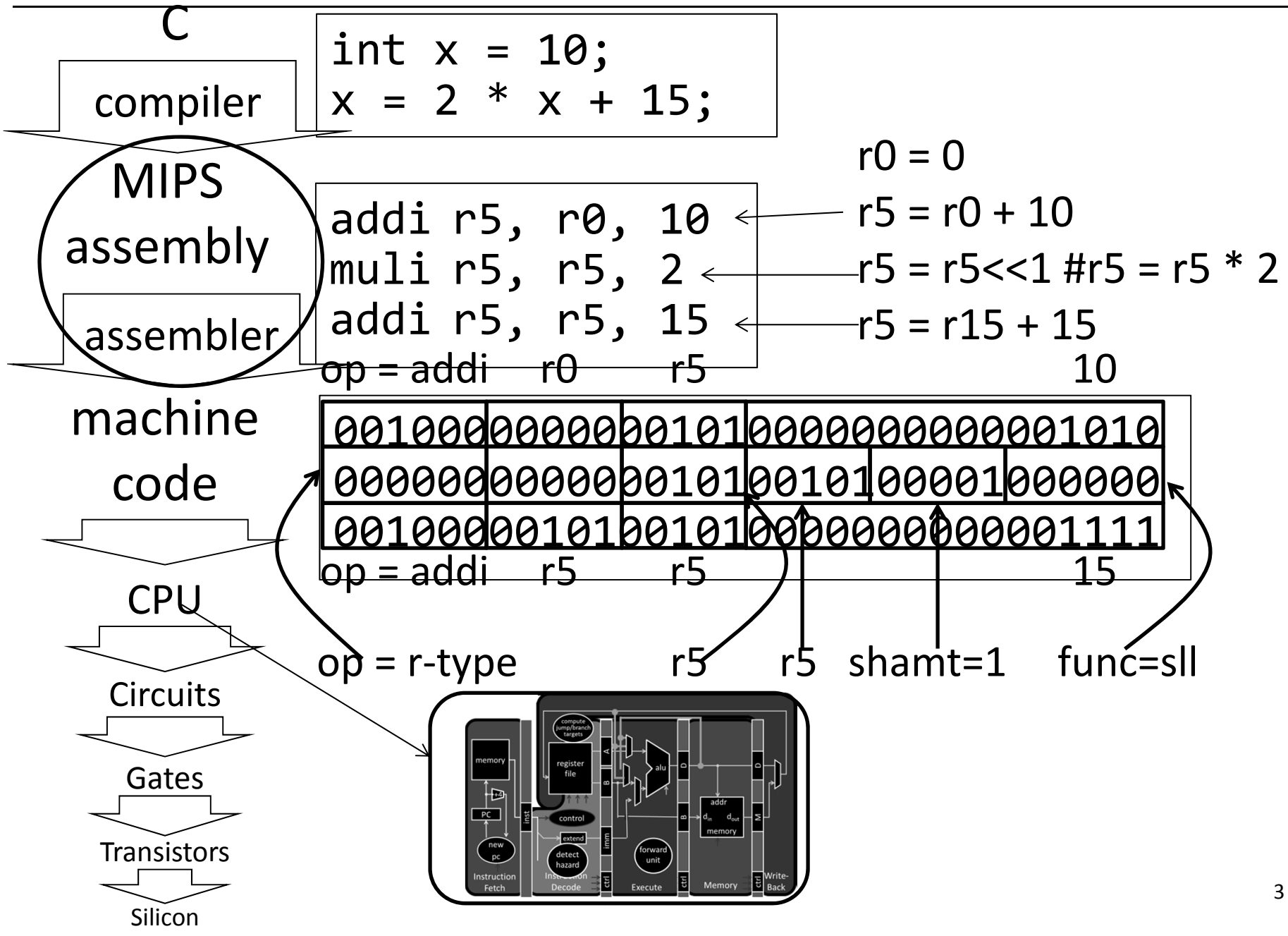
Cornell University

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

Big Picture: Where are we now?



Big Picture: Where are we going?



Goals for Today

Instruction Set Architectures

- ISA Variations (i.e. ARM), CISC, RISC

(Intuition for) Assemblers

- Translate symbolic instructions to binary machine code

Time for Prelim1 Questions

Next Time

- Program Structure and Calling Conventions

Next Goal

Is MIPS the only possible instruction set architecture (ISA)?

What are the alternatives?

MIPS Design Principles

Simplicity favors regularity

- 32 bit instructions

Smaller is faster

- Small register file

Make the common case fast

- Include support for constants

Good design demands good compromises

- Support for different type of interpretations/classes

What happens when the common case is slow?

- Can we add some complexity in the ISA for a speedup?

ISA Variations: Conditional Instructions

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

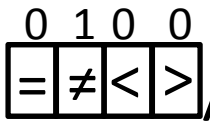
In MIPS, performance will be slow if code has a lot of branches

```
Loop: BEQ Ri, Rj, End      // if "NE" (not equal), then stay in loop
      SLT Rd, Rj, Ri       //  "GT" if (i > j),
      BNE Rd, R0, Else     //  …
      SUB Ri, Ri, Rj       // if "GT" (greater than), i = i-j;
      J Loop
Else: SUB Rj, Rj, Ri       // or "LT" if (i < j)
      J Loop               // if "LT" (less than), j = j-i;
End:
```

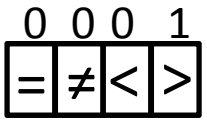
ISA Variations: Conditional Instructions

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

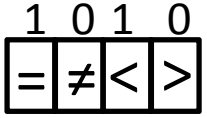
In ARM, can avoid delay due to Branches with conditional instructions

LOOP: CMP Ri, Rj 

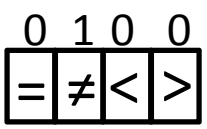
// set condition "NE" if (i != j)
// "GT" if (i > j),
// or "LT" if (i < j)



SUBGT Ri, Ri, Rj // if "GT" (greater than), i = i-j;



SUBLE Rj, Rj, Ri // if "LE" (less than or equal), j = j-i;



BNE loop // if "NE" (not equal), then loop

ARM: Other Cool operations

Shift one register (e.g. Rc) any amount

Add to another register (e.g. Rb)

Store result in a different register (e.g. Ra)

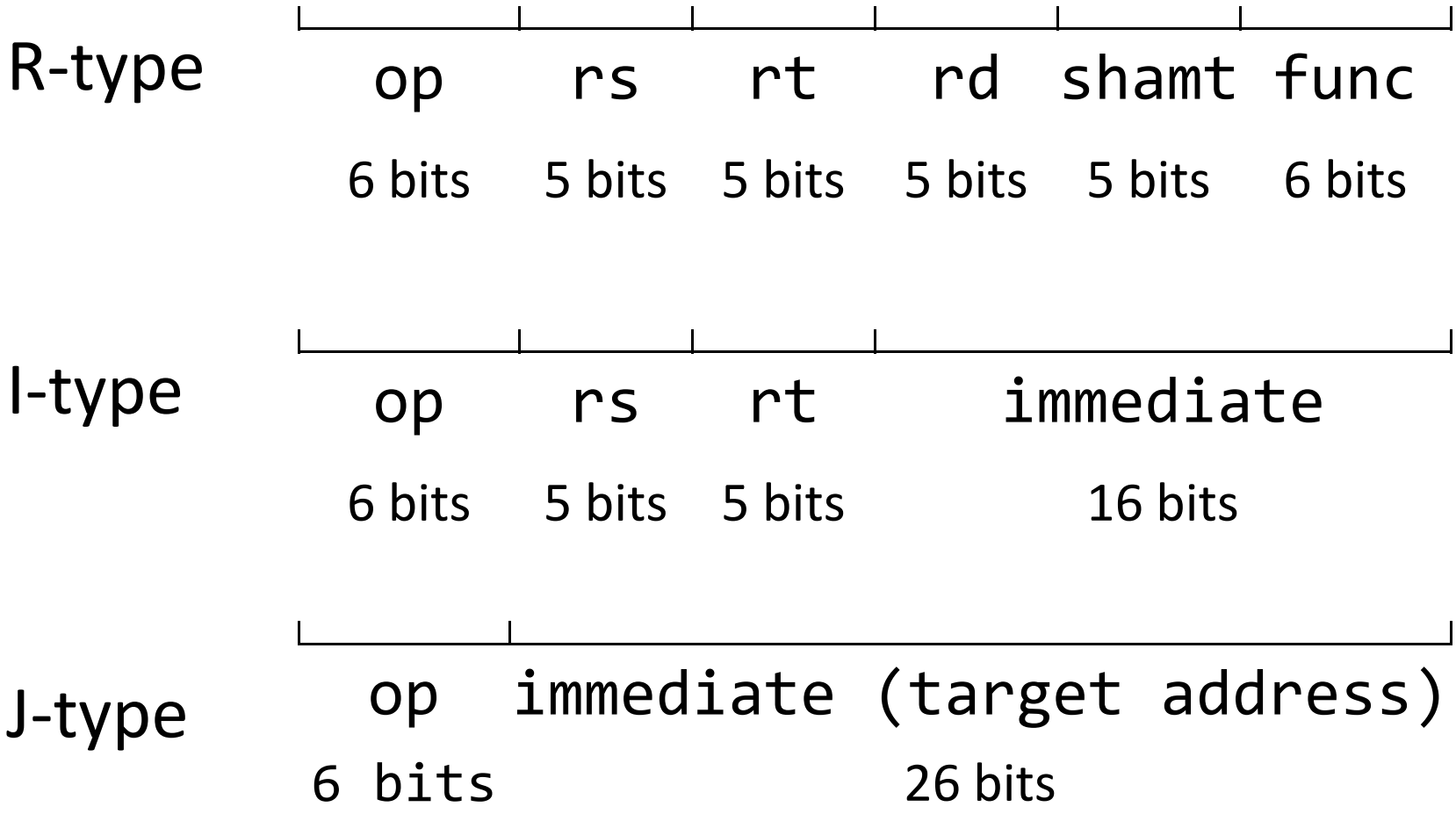
ADD Ra, Rb, Rc LSL #4

Ra = Rb + Rc<<4

Ra = Rb + Rc x 16

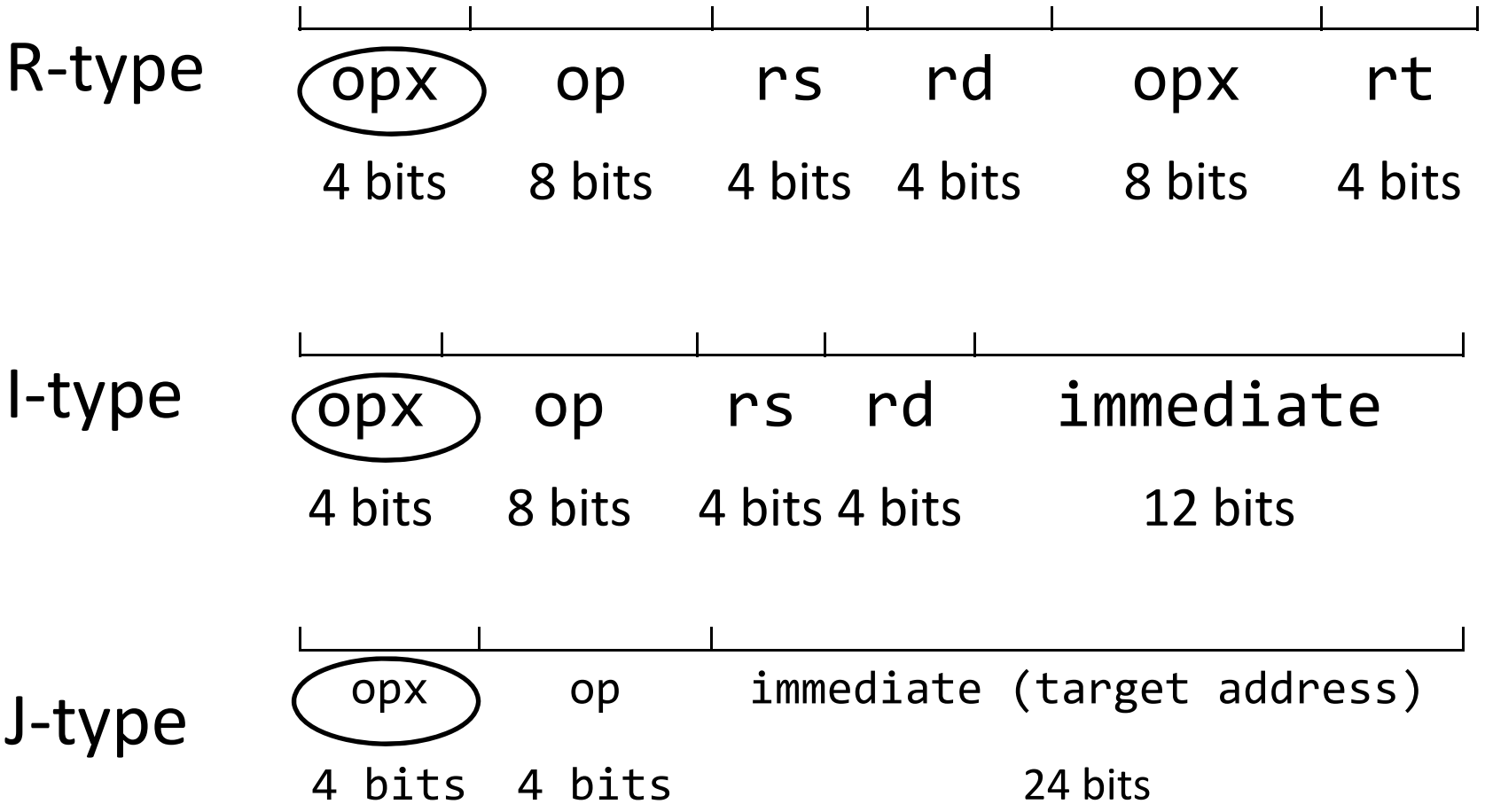
MIPS instruction formats

All MIPS instructions are 32 bits long, has 3 formats



ARM instruction formats

All ARM 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

ARM Instruction Set Architecture

All ARM instructions are 32 bits long, has 3 formats

Reduced Instruction Set Computer (RISC) properties

- Only Load/Store instructions access memory
- Instructions operate on operands in processor registers
- 16 registers

Complex Instruction Set Computer (CISC) properties

- Autoincrement, autodecrement, PC-relative addressing
- Conditional execution
- Multiple words can be accessed from memory with a single instruction (SIMD: single instr multiple data)

Takeaway

We can reduce the number of instructions to execute a program and possibly increase performance by adding complexity to the ISA.

Next Goal

How much complexity to add to an ISA?

How does the CISC philosophy compare to RISC?

Complex Instruction Set Computers (CISC)

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

Reduced Instruction Set Computer

John Cock

- IBM 801, 1980 (started in 1975)
- Name 801 came from the bldg that housed the project
- Idea: Possible to make a very small and very fast core
- Influences: Known as “the father of RISC Architecture”. Turing Award Recipient and National Medal of Science.

Complexity

MIPS = Reduced Instruction Set Computer (RlSC)

- ≈ 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 (ClSC)

- > 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	CISC Rebuttal
Regularity & simplicity	Compilers can be smart
Leaner means faster	Transistors are plentiful
Optimize the common case	Legacy is important
	Code size counts
	Micro-code!
Energy efficiency	Desktops/Servers
Embedded Systems	
Phones/Tablets	

ARMDroid vs WinTel

- Android OS on ARM processor

- Windows OS on Intel (x86) processor

Takeaway

We can reduce the number of instructions to execute a program and possibly increase performance by adding complexity to the ISA.

Back in the day… CISC was necessary because everybody programmed in assembly and machine code! Today, CISC ISA's are still dominate today due to the prevalence of x86 ISA processors. However, RISC ISA's today such as ARM have an ever increase marketshare (of our everyday life!).

ARM borrows a bit from both RISC and CISC.

Goals for Today

Instruction Set Architectures

- ISA Variations (i.e. ARM), CISC, RISC

(Intuition for) Assemblers

- Translate symbolic instructions to binary machine code

Time for Prelim1 Questions

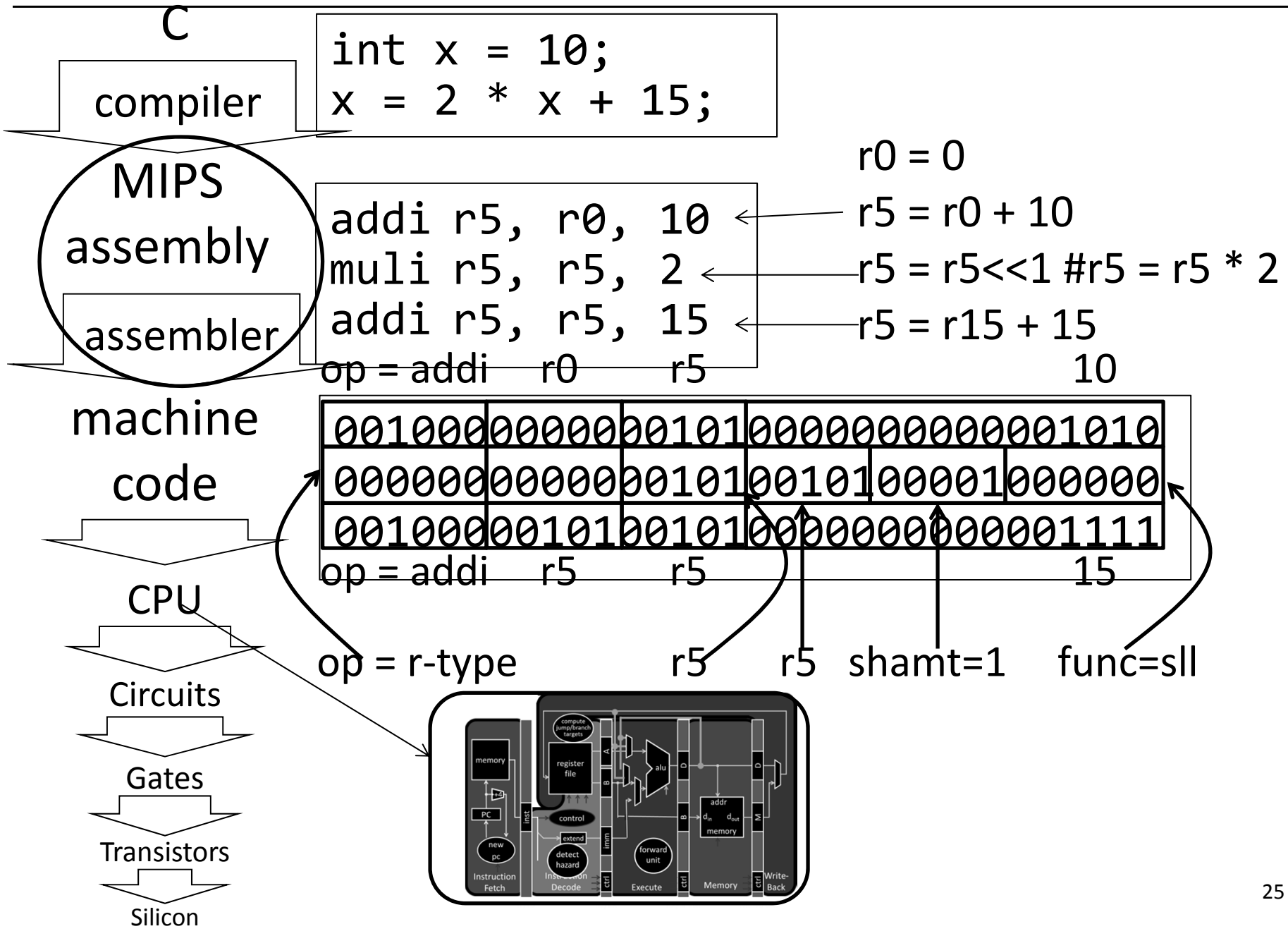
Next Time

- Program Structure and Calling Conventions

Next Goal

How do we (as humans or compiler) program on top of a given ISA?

Big Picture: Where are we going?



Assembler

Translates text *assembly language* to binary machine code

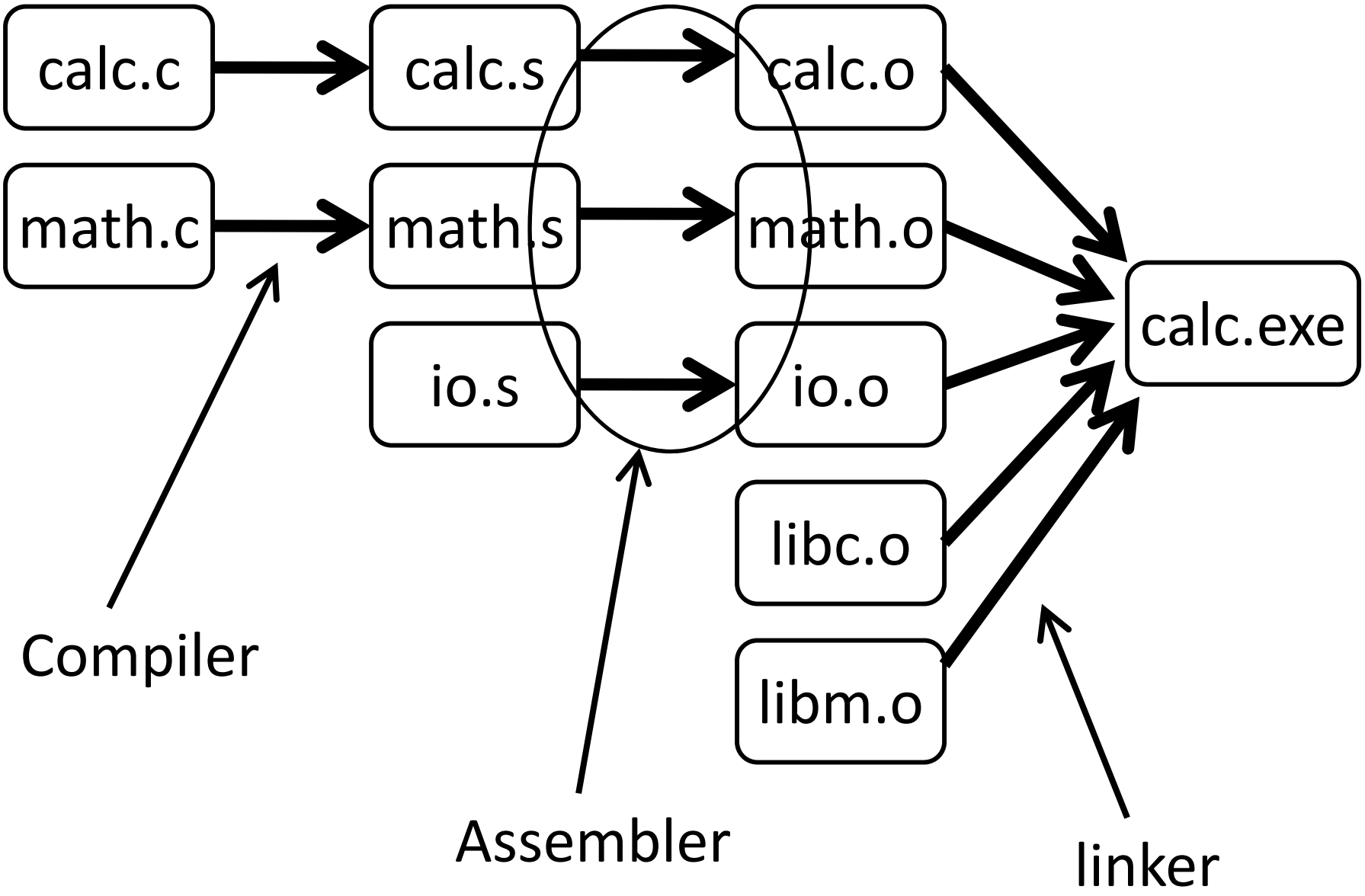
Input: a text file containing MIPS instructions in human readable form

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

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

```
00100000000001010000000000001010
00000000000001010010100001000000
00100000101001010000000000001111
```

Assembler


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

Assembler

Translates text *assembly language* to binary machine code

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

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

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

```
00100000000001010000000000001010
00000000000001010010100001000000
00100000101001010000000000001111
```

Assembly Language

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

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, ...

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)

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)

MIPS Assembly Language Instructions

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

Pseudo-Instructions

Pseudo-Instructions

NOP # do nothing

- SLL r0, r0, 0

MOVE reg, reg # copy between regs

- ADD R2, R0, R1 # copies contents of R1 to R2

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

LA reg, label # load address (32 bits)

B label # unconditional branch

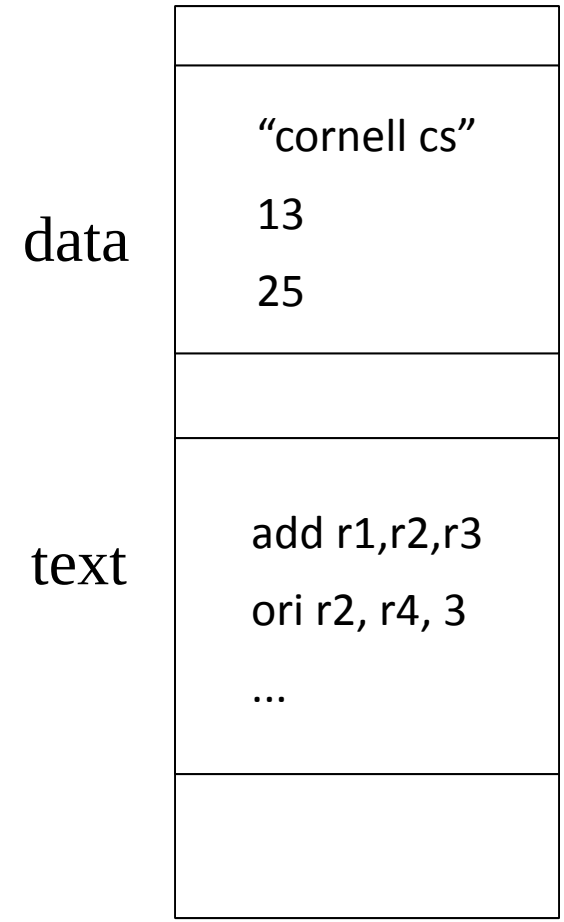
BLT reg, reg, label # branch less than

- SLT r1, rA, rB # r1 = 1 if R[rA] < R[rB]; o.w. r1 = 0
- BNE r1, r0, label # go to address label if r1!=r0; i.t. rA < rB

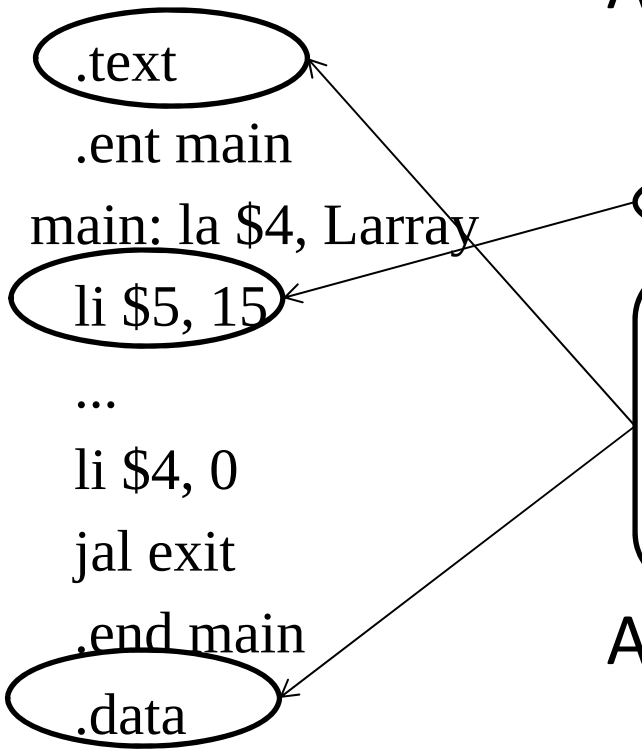
Program Layout

Programs consist of segments used for different purposes

- Text: holds instructions
- Data: holds statically allocated program data such as variables, strings, etc.



Assembling Programs



Assembly files consist of a mix of

+ instructions

+ pseudo-instructions

+ assembler (data/layout) directives

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

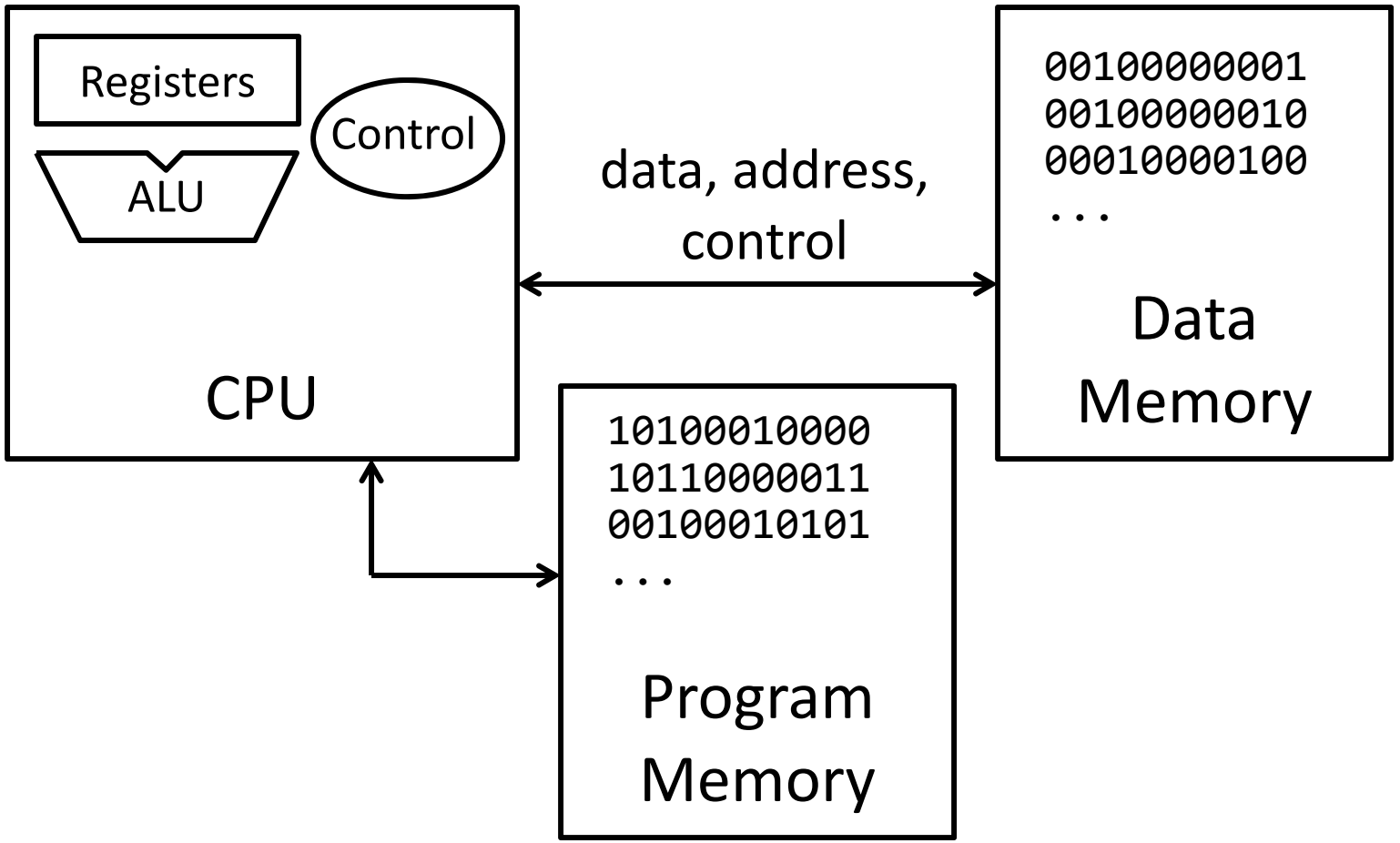
Assembled to an Object File

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

Assembling Programs

Assembly with a but using (modified) Harvard architecture

- Need segments since data and program stored together in memory



Takeaway

Assembly is a low-level task

- Need to assemble assembly language into machine code binary. Requires
- Assembly language instructions
- *pseudo-instructions*
- And Specify layout and data using *assembler directives*

- Since we use a modified Harvard Architecture (Von Neumann architecture) that mixes data and instructions in memory

  … but best kept in separate *segments*

Next time

How do we coordinate use of registers?

Calling Conventions!

PA1 due Monday

Administrivia

Prelim1: ***Today***, Tuesday, February 26th in evening

- **Location: GSHG76: Goldwin Smith Hall room G76**
- Time: We will start at ***7:30pm sharp***, so come early

- Closed Book: ***NO NOTES, BOOK, CALCULATOR, CELL PHONE***
  - Cannot use electronic device or outside material
- Practice prelims are online in CMS
- Material covered everything up to end of ***last*** week
  - Appendix C (logic, gates, FSMs, memory, ALUs)
  - Chapter 4 (pipelined [and non-pipeline] MIPS processor with hazards)
  - Chapters 2 (Numbers / Arithmetic, simple MIPS instructions)
  - Chapter 1 (Performance)
  - HW1, HW2, Lab0, Lab1, Lab2

Administrivia

Project1 (PA1) due next Monday, March 4th

- 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