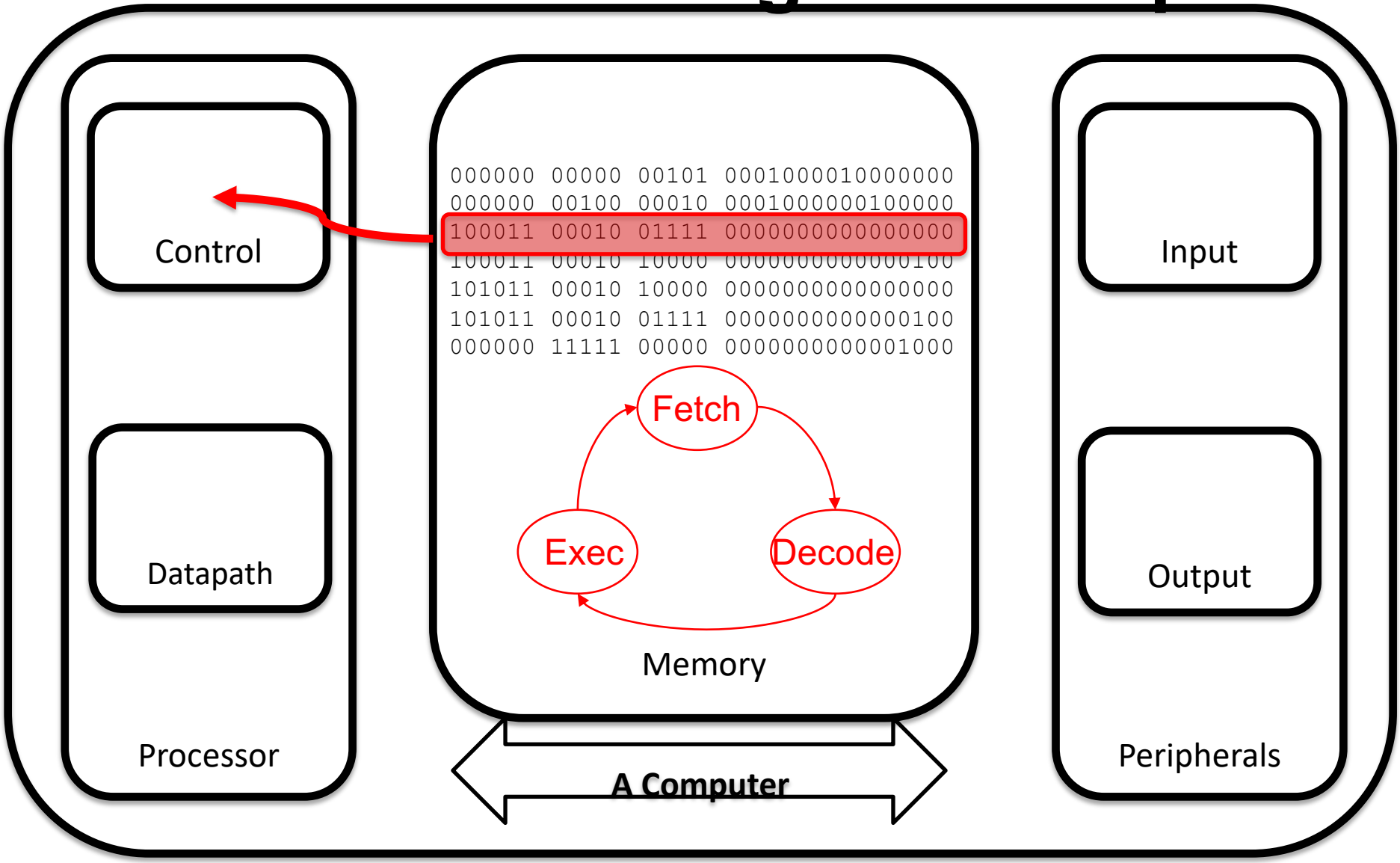
CprE 381: Computer Organization and Assembly Level Programming

Henry Duwe
Electrical and Computer Engineering
Iowa State University

Administrative

- HW0 posted (due next Wed Jan 23)
- HW1 will be posted Mon (due Mon Jan 28)
- OH:
  - 321 Durham (W 3pm, F 1:30pm)

Review: Stored Program Computer

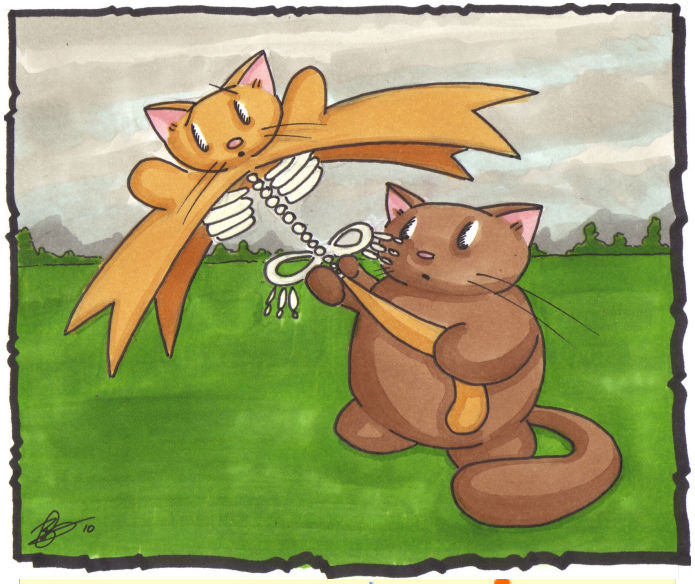


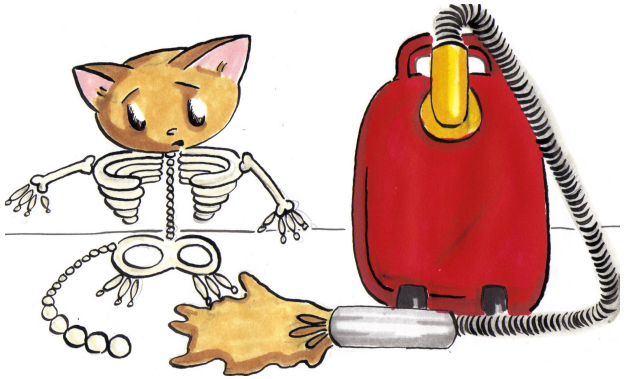
Review: ISA

- ISA – Instruction Set Architecture
  - Abstract interface (i.e., contract) between HW and SW
  - Indicates what *functionality* supported by HW
  - Indicates how SW must be encoded for HW to understand

- ABI – Application Binary Interface
  - ISA + OS interfaces (for portability and composability)
  - "Convention"

ISA Design



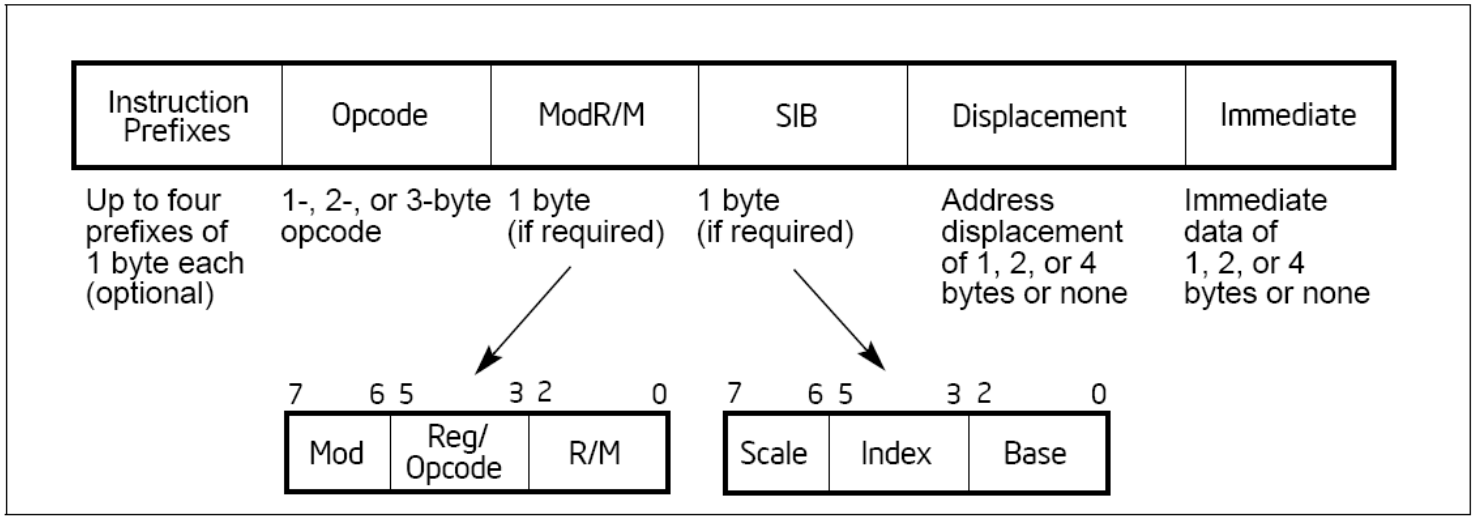


ISA Design

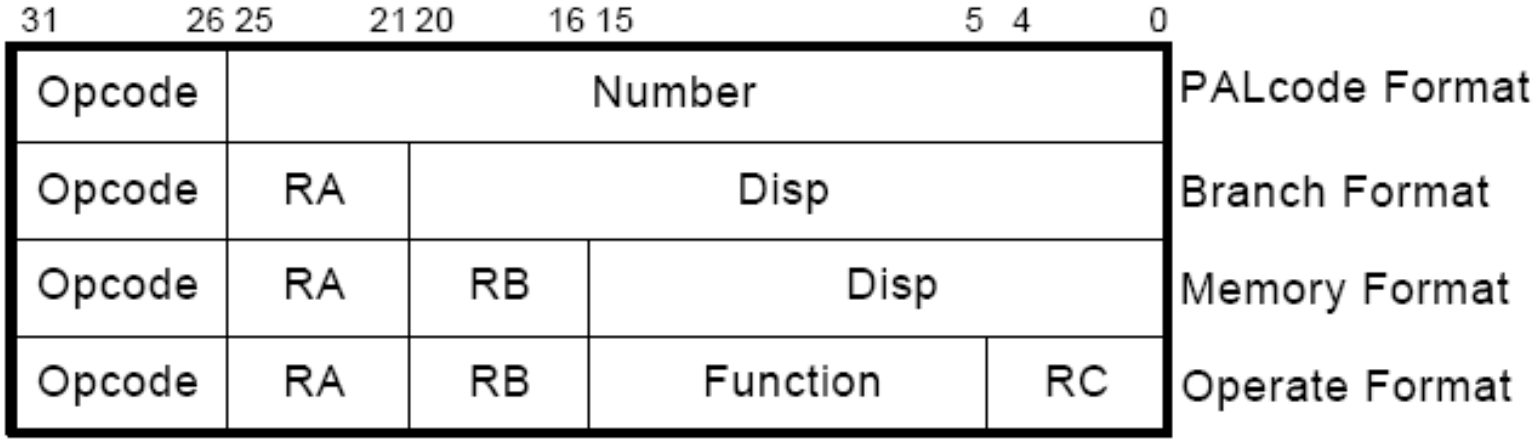
- Complex Instruction Set Computers (CISC)
  - More and more complex instructions
    - e.g., inc top of stack, insert into queue, mul polynomials
  - Smaller code (fewer instructions per program)
  - Variable length instructions
- Reduced Instruction Set Computers (RISC)
  - Fewer, simpler instructions
  - Regular construction
  - Easier to design fast, efficient HW
  - Easier for compilers to generate code
- Useful to understand ISA design choices/philosophies
- What about One Instruction Set Computers (OISC)?

Example: x86 versus Alpha

- x86:



- Alpha:



MIPS ISA

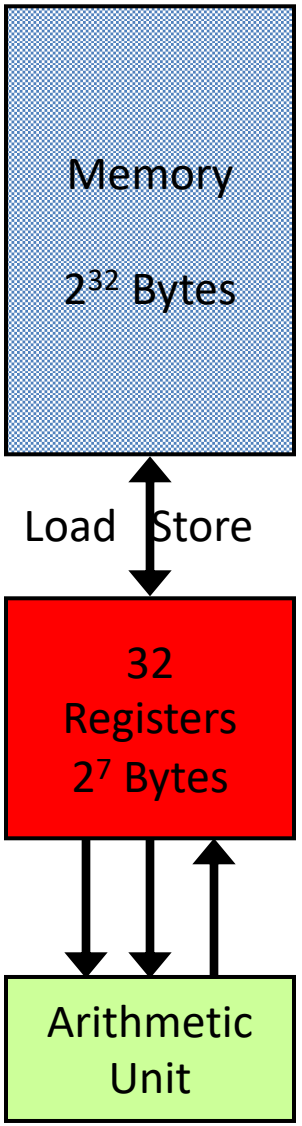
- MIPS – semiconductor company that built one of the first commercial RISC architectures
- MIPS primary ISA for detailed study in this class
- Why MIPS instead of Intel x86?
  - MIPS is simple, elegant, and easy to understand
  - x86 is ugly and complicated to explain
  - MIPS (and derivatives) widely used in embedded apps

C vs. MIPS (Programmer's Interface)

	C	MIPS ISA
Registers		32 32b integer, R0=0 32 32b single FP 16 64b double FP PC and special registers
Memory	local variables global variables	2^{32} linear array of bytes
Data types	int, short, char, unsigned, float, double, aggregate data types, pointers	word(32b), byte(8b) half-word(16b), single FP(32b), double FP(64b)
Arithmetic operators	`+`,`-`,`*`,`%`,`++`,`<`,etc.	`add`, `sub`, `mult`, `slt`, `etc`.
Memory Access	`a`, `*a`, `a[i]`, `a[i][j]`	`lw`, `sw`, `lh`, `sh`, `lb`, `sb`
Control	if-else, while, do-while, for, switch, procedure call	branches, jumps, jump and link

Why Have Registers?

- Memory-memory ISA
  - All HLL variables declared in memory
  - Why not directly operate on memory operands?
- Benefits of registers
  - Smaller is faster
  - Multiple concurrent accesses
  - Shorter names
- Load-store ISA
  - Arithmetic operations only use register operands
  - Data is loaded into registers, operated on, and stored back to memory
  - All RISC instruction sets



Using Registers

- Registers are a finite resource that need to be managed
  - (Assembly) Programmer
  - Compiler: register allocation
- Goals
  - Keep data in registers as much as possible
  - Always use data in registers if possible
- Issues
  - Finite number of registers available
    - Spill registers to memory when all registers in use
  - Arrays
    - Data is too large to store in registers
  - What's the impact of fewer or more registers?

C vs. MIPS (Programmer's Interface)

	C	MIPS ISA
Registers		32 32b integer, R0=0 32 32b single FP 16 64b double FP PC and special registers
Memory	local variables global variables	2^{32} linear array of bytes
Data types	int, short, char, unsigned, float, double, aggregate data types, pointers	word(32b), byte(8b) half-word(16b), single FP(32b), double FP(64b)
Arithmetic operators	+,−,*,%,++,<,etc.	`add`, `sub`, `mult`, `slt`, `etc`.
Memory Access	`a`, `*a`, `a[i]`, `a[i][j]`	`lw`, `sw`, `lh`, `sh`, `lb`, `sb`
Control	if-else, while, do-while, for, switch, procedure call	branches, jumps, jump and link

MIPS Assembly Instructions

- The basic type of instruction has four components:
  - Operation name
  - Destination operand
  - 1st source operand
  - 2nd source operand

```
add dst, src1, src2    #dst=src1+src2
```

- `dst`, `src1`, and `src2` are register names ($)
- What do these instructions do?
  - `add $1, $1, $1`
  - `sub $1, $1, $1`
  - `add $2, $1, $0`

C Code Example

Simple C procedure: sum_pow2=2^{b+c}

```
 1: int sum_pow2 (int b, int c)
 2: {
 3:   int pow2[8] = {1, 2, 4, 8, 16, 32, 64, 128};
 4:   int a, ret;
 5:   a = b + c;
 6:   if (a < 8)
 7:      ret = pow2[a];
 8:   else
 9:      ret = 0;
10:   return(ret);
11: }
```

Arithmetic Operations

- Consider line 5, C operation for addition

```
a = b + c;
```

- Assume the variables are in registers `$1-$3` respectively
- The `add` operator using registers

```
add $1, $2, $3    # a = b + c
```

- Use the `sub` operator for `a=b-c` in MIPS

```
sub $1, $2, $3    # a = b - c
```

- But we know that `a`, `b`, and `c` really refer to memory locations

Complex Statements

- What about more complex statements?

```
a = b + c + d - e;
```

- Break into multiple instructions

```
add $t0, $s1, $s2   # $t0 = b + c
add $t1, $t0, $s3   # $t1 = $t0 + d
sub $s0, $t1, $s4   # a = $t1 - e
```

Constants

- Often want to be able to specify operand in the instruction: immediate or literal
- Use the `addi` instruction

```
addi dst, src, immediate
```

- The immediate is a 16 bit signed value between -2^{15} and $2^{15}-1$
- Sign extended to 32 bits
- Consider the following C code

```
a++;
```

- Implemented using the `addi` operator

```
addi $s0, $s0, 1    # a = a + 1
```

- What do these instructions do?
  - `add $2, $1, 0`

MIPS Simple Arithmetic

Instruction	Example	Meaning	Comments
add	`add $1,$2,$3`	`$1 = $2 + $3`	3 operands; Overflow
subtract	`sub $1,$2,$3`	`$1 = $2 - $3`	3 operands; Overflow
add immediate	`addi $1,$2,100`	`$1 = $2 + 100`	+ constant; Overflow
add unsigned	`addu $1,$2,$3`	`$1 = $2 + $3`	3 operands; No overflow
sub unsigned	`subu $1,$2,$3`	`$1 = $2 - $3`	3 operands; No overflow
add imm unsign	`addiu $1,$2,100`	`$1 = $2 + 100`	+ constant; No overflow

- How does C treat overflow?

MIPS Simple Arithmetic

Instruction	Example	Meaning	Comments
add	`add $1,$2,$3`	`$1 = $2 + $3`	3 operands; Overflow
subtract	`sub $1,$2,$3`	`$1 = $2 - $3`	3 operands; Overflow
add immediate	`addi $1,$2,100`	`$1 = $2 + 100`	+ constant; Overflow
add			w
sub			w
add			ow

In-class Assessment!

- How does C treat overflow?

C vs. MIPS (Programmer's Interface)

	C	MIPS ISA
Registers		32 32b integer, R0=0 32 32b single FP 16 64b double FP PC and special registers
Memory	local variables global variables	2^{32} linear array of bytes
Data types	int, short, char, unsigned, float, double, aggregate data types, pointers	word(32b), byte(8b) half-word(16b), single FP(32b), double FP(64b)
Arithmetic operators	+,−,*,%,++,<,etc.	`add`, `sub`, `mult`, `slt`, `etc`.
Memory Access	`a`, `*a`, `a[i]`, `a[i][j]`	`lw`, `sw`, `lh`, `sh`, `lb`, `sb`
Control	if-else, while, do-while, for, switch, procedure call	branches, jumps, jump and link

Disclaimer

- There were no cats harmed in the making of this lecture.

Acknowledgments

- These slides contain material developed and copyright by:
  - Joe Zambreno (Iowa State)
  - David Patterson (UC Berkeley)
  - Mary Jane Irwin (Penn State)
  - Christos Kozyrakis (Stanford)
  - Onur Mutlu (Carnegie Mellon)
  - Krste Asanović (UC Berkeley)