HARVARD
John A. Paulson
School of Engineering
and Applied Sciences

CS153: Compilers
Lecture 2: Assembly

Stephen Chong

https://www.seas.harvard.edu/courses/cs153

Announcements (1/2)

- Name tags
- Device free seating
  - Right side of classroom (as facing front): no devices
    - Allow you to commit to being device-free/avoid devices
- Student info: please complete END OF TODAY (Thursday Sept 6)
  - https://tiny.cc/cs153-registration
  - We need it to set you up for the projects

Announcements (2/2)

- Project 1 will be released today
  - We will email you link to instructions (on webpage) and project repo
  - Please don't share repo link!!!
- We **strongly** encourage you to do projects in pairs
  - You do not need to have the same partner for all projects
  - http://tiny.cc/cs153-partner
  - Fill in form by END OF FRIDAY (Sept 7)if you would like to be matched up with a partner

Today

- Quick overview of the MIPS instruction set.
  - We're going to be compiling to MIPS assembly language.
  - So you need to know how to program at the MIPS level.
    - Helps to have a bit of architecture background to understand *why* MIPS assembly is the way it is.
- Online resources describe MIPS in more detail (see end of lecture notes)

Turning C into Machine Code

C program
(myprog.c)

```
int dosum(int i, int j) {
    return i+j;
}
```

C compiler (gcc)

Assembly program
(myprog.s)

```
dosum:
        pushl   %ebp
        movl    %esp, %ebp
        movl    12(%ebp), %eax
        addl    8(%ebp), %eax
        popl    %ebp
         ret
```

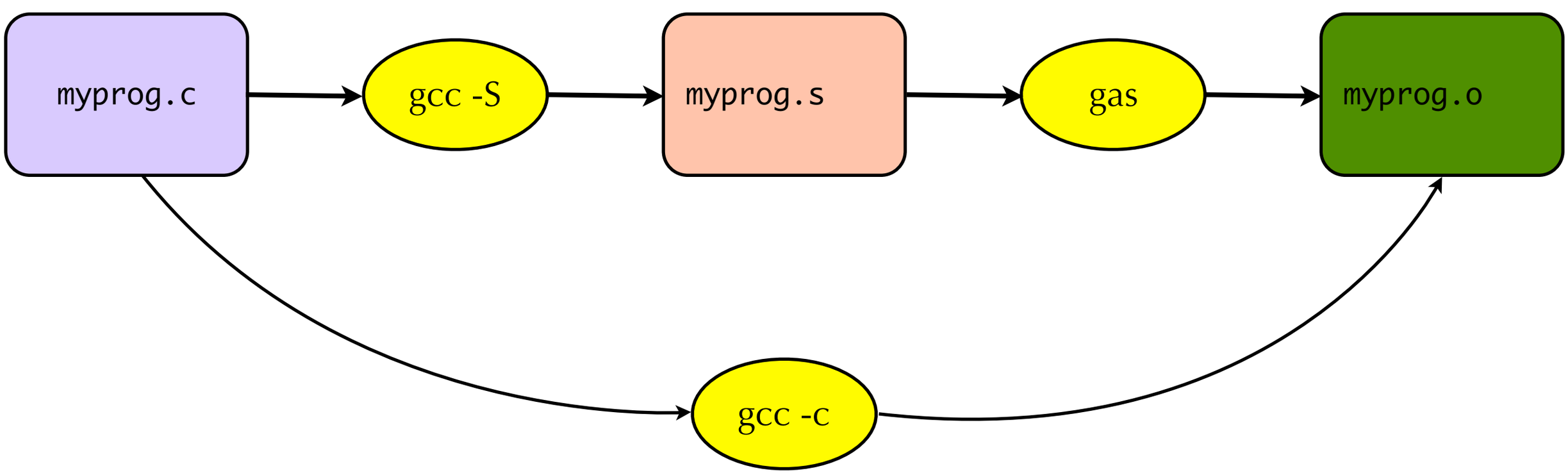
Assembler (gas)

Machine code
(myprog.o)

```
80483b0: 55 89 e5 8b 45 0c 03 45 08 5d c3
```

Skipping assembly language

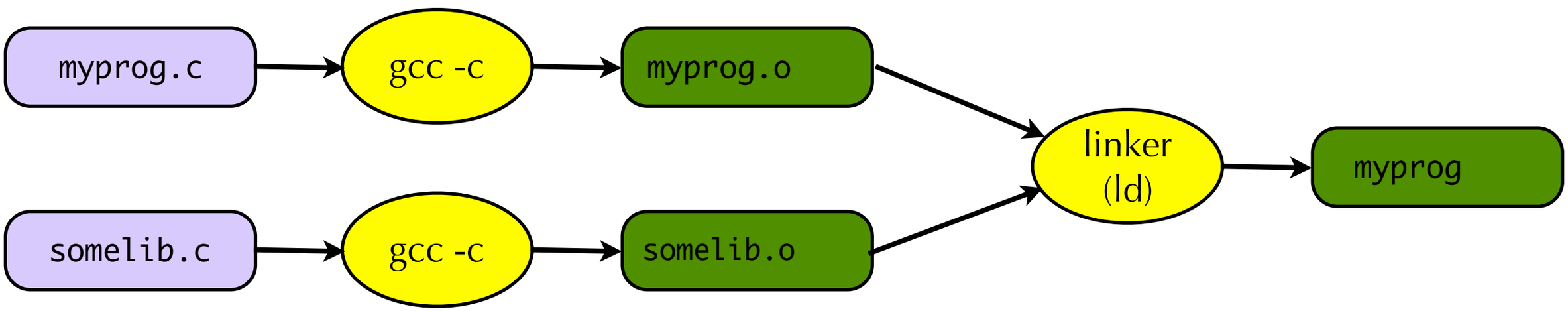
- Most C compilers generate machine code (object files) directly.
  - That is, without actually generating the human-readable assembly file.
  - Assembly language is mostly useful to people, not machines.



- Can generate assembly from C using "`gcc -S`"
  - And then compile to an object file by hand using "`gas`"

Object files and executables

- C source file (myprog.c) is compiled into an **object file** (myprog.o)
  - Object file contains the machine code for that C file.
  - It may contain references to external variables and routines
  - E.g., if myprog.c calls printf(), then myprog.o will contain a reference to printf().
- Multiple object files are **linked** to produce an executable file.
  - Typically, standard libraries (e.g., "libc") are included in the linking process.
  - Libraries are just collections of pre-compiled object files, nothing more!



Characteristics of assembly language

- Assembly language is very, very simple.
- Simple, minimal data types
  - Integer data of 1, 2, 4, or 8 bytes
  - Floating point data of 4, 8, or 10 bytes
  - No aggregate types such as arrays or structures!
- Primitive operations
  - Perform arithmetic operation on registers or memory (add, subtract, etc.)
  - Read data from memory into a register
  - Store data from register into memory
  - Transfer control of program (jump to new address)
  - Test a control flag, conditional jump (e.g., jump only if zero flag set)
- More complex operations must be built up as (possibly long) sequences of instructions.

Assembly vs Machine Code

- We write assembly language instructions
  - e.g., "`addi $r1, $r2, 42`"
- The machine interprets machine code bits
  - e.g., "`10101100110011...`"
- Your first assignment is to build an interpreter for a subset of the MIPS machine code.
- The assembler takes care of compiling assembly language to bits for us.
  - It also provides a few conveniences as we'll see.

MIPS

- MIPS is a RISC computer architecture developed 1985 onwards
  - Multiple versions: MIPS I, II, III, IV, and V
- Designed as a general purpose processor
- Historically used in personal computers, workstations, servers, video game consoles (Nintendo 64, Sony PlayStation, PlayStation 2, and PlayStation Portable), supercomputers
- Currently used in embedded systems
  - E.g., residential gateways and routers
  - And many CS courses!
- Why are we using it?
  - Relatively simple instruction set
  - "The MIPS architecture may be the epitome of a simple, clean RISC machine." –James Larus

Some MIPS Assembly

```
int sum(int n) {
  int s = 0;
  for (; n != 0; n--)
    s += n;
  return s;
}
```

```
sum:    ori  $2,$0,$0
        b    test
loop:   add  $2,$2,$4
        subi $4,$4,1
test:   bne  $4,$0,loop
        jr   $31
```

```
int main() {
  return sum(42);
}
```

```
main:   ori  $4,$0,42
        move $17,$31
        jal  sum
        jr   $17
```

An X86 Example (-O0):

```
_sum:
   pushq %rbp
   movq  %rsp, %rbp
   movl  %edi, -4(%rbp)
   movl  $0, -8(%rbp)
LBB0_1:
   cmpl  $0, -4(%rbp)
   je LBB0_4
   movl  -4(%rbp), %eax
   addl  -8(%rbp), %eax
   movl  %eax, -8(%rbp)
   movl  -4(%rbp), %eax
   addl  $-1, %eax
   movl  %eax, -4(%rbp)
   jmp   LBB0_1
LBB0_4:
   movl  -8(%rbp), %eax
   popq  %rbp
   retq
```

```
_main:
   pushq %rbp
   movq  %rsp, %rbp
   subq  $16, %rsp
   movl  $42, %edi
   movl  $0, -4(%rbp)
   callq _sum
   addq  $16, %rsp
   popq  %rbp
   retq
```

An X86 Example (-O3):

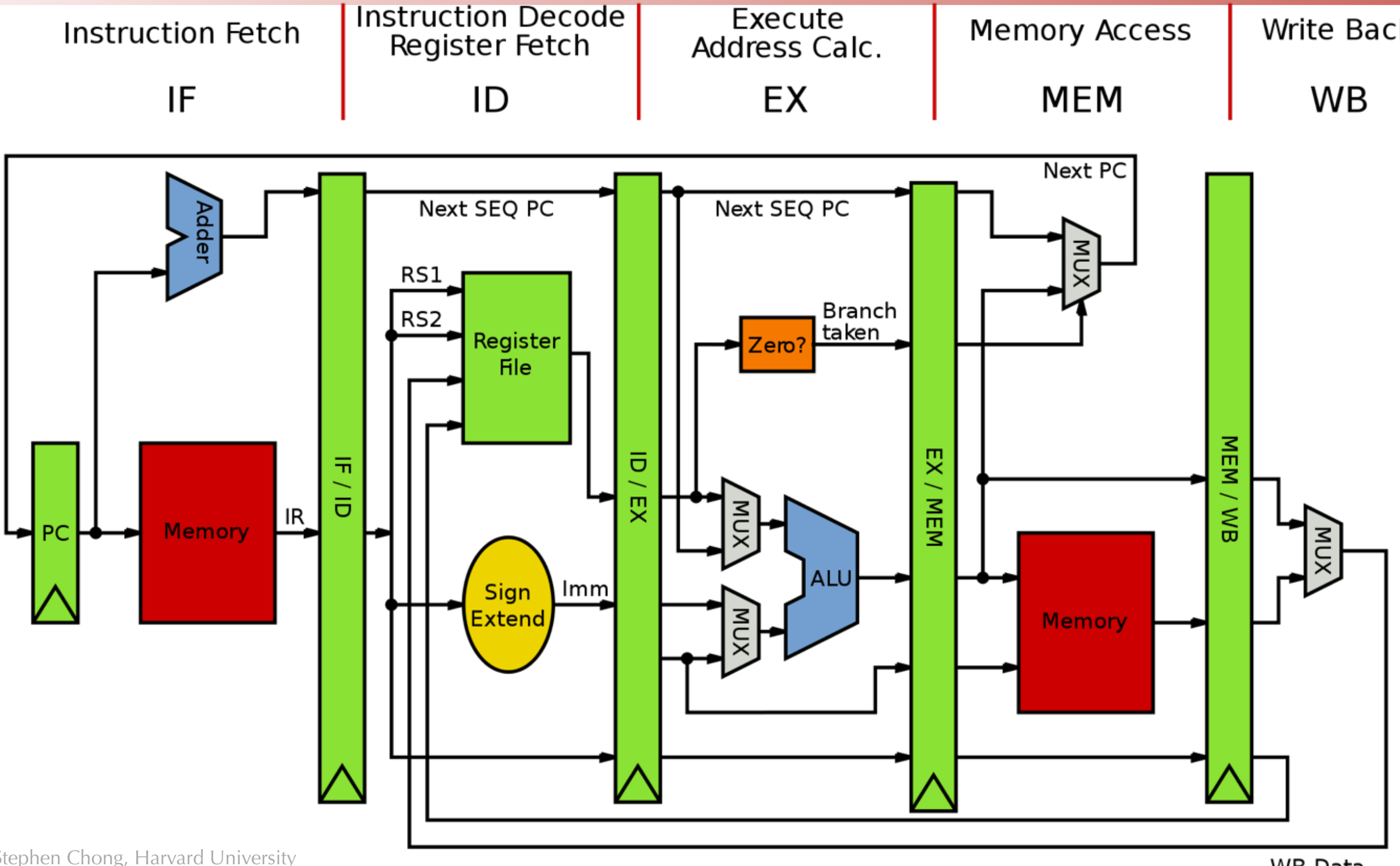
```
_sum:
   pushq %rbp
   movq  %rsp, %rbp
   testl %edi, %edi
   je LBB0_1
   leal  -1(%rdi), %eax
   leal  -2(%rdi), %ecx
   imulq %rax, %rcx
   imull %eax, %eax
   shrq  %rcx
   addl  %edi, %eax
   subl  %ecx, %eax
   popq  %rbp
   retq
LBB0_1:
   xorl  %eax, %eax
   popq  %rbp
   retq
```

```
_main:
   pushq %rbp
   movq  %rsp, %rbp
   movl  $903, %eax
   popq  %rbp
   retq
```

MIPS

- Reduced Instruction Set Computer (RISC)
  - Load/store architecture
    - i.e., only memory operations are load and store
  - All operands are either registers or constants
  - All instructions same size (4 bytes) and aligned on 4-byte boundary.
  - Simple, orthogonal instructions
    - e.g., no `subi`, (`addi` and negate value)
  - All registers (except $0) can be used in all instructions.
    - Reading $0 always returns the value 0
- Easy to make fast: pipeline, superscalar

MIPS Datapath



x86

- Complex Instruction Set Computer (CISC)
  - Instructions can operate on memory values
    - e.g., `add [eax],ebx`
  - Complex, multi-cycle instructions
    - e.g., string-copy, call
  - Many ways to do the same thing
    - e.g., `add eax,1` `inc eax` `sub eax,-1`
  - Instructions are variable-length (1-10 bytes)
  - Registers are not orthogonal
- Hard to make fast…(but they do anyway)

Tradeoffs

- x86 (as opposed to MIPS):
  - Lots of existing software.
  - Harder to decode (i.e., parse).
  - Harder to assemble/compile to.
  - Code can be more compact (3 bytes on avg.)
  - I-cache is more effective…
  - Easier to add new instructions.
- Todays implementations have the best of both:
  - Intel & AMD chips suck in x86 instructions and compile them to "micro-ops", caching the results.
  - Core execution engine more like MIPS.

MIPS Registers and Usage Conventions

Register name	Number	Usage
$zero	0	constant 0
$at	1	reserved for assembler
$v0	2	expression evaluation and results of a function
$v1	3	expression evaluation and results of a function
$a0	4	argument 1
$a1	5	argument 2
$a2	6	argument 3
$a3	7	argument 4
$t0	8	temporary (not preserved across call)
$t1	9	temporary (not preserved across call)
$t2	10	temporary (not preserved across call)
$t3	11	temporary (not preserved across call)
$t4	12	temporary (not preserved across call)
$t5	13	temporary (not preserved across call)
$t6	14	temporary (not preserved across call)
$t7	15	temporary (not preserved across call)

MIPS Registers and Usage Conventions

Register name	Number	Usage
$s0	16	saved temporary (preserved across call)
$s1	17	saved temporary (preserved across call)
$s2	18	saved temporary (preserved across call)
$s3	19	saved temporary (preserved across call)
$s4	20	saved temporary (preserved across call)
$s5	21	saved temporary (preserved across call)
$s6	22	saved temporary (preserved across call)
$s7	23	saved temporary (preserved across call)
$t8	24	temporary (not preserved across call)
$t9	25	temporary (not preserved across call)
$k0	26	reserved for OS kernel
$k1	27	reserved for OS kernel
$gp	28	pointer to global area
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address (used by function call)

MIPS Instructions

- Arithmetic & logical instructions:
  - `add`, `sub`, `and`, `or`, `sll`, `srl`, `sra`, …
  - Register and immediate forms:
    - `add  $rd, $rs, $rt`
    - `addi  $rd, $rs, <16-bit-immed>`
  - Any registers (except $0 returns 0)
  - Also a distinction between overflow and no-overflow (we'll ignore for now.

Detour: 2's complement

- Representing non-negative integers in bits is straightforward
- How do we represent negative integers in bits?
- Three common encodings:
  - Sign and magnitude
  - Ones' complement
  - Two's complement

Two's complement

- If integer k is represented by bits $b_1 \ldots b_n$, then $-k$ is represented by `100...00` - $b_1 \ldots b_n$ (where |`100...00`|=n+1)
  - Equivalent to taking ones' complement and adding 1
  - E.g., using 4 bits:
    - 6 = `0110`
    - -6 = `10000-0110 = 1010 = (1111-0110)+1`
- Using n bits, can represent numbers 2^n values
  - E.g., using 4 bits, can represent integers

    -8, -7, …, -1, 0, 1, …, 6, 7
  - Like sign and magnitude and ones' complement, first bit indicates whether number is negative

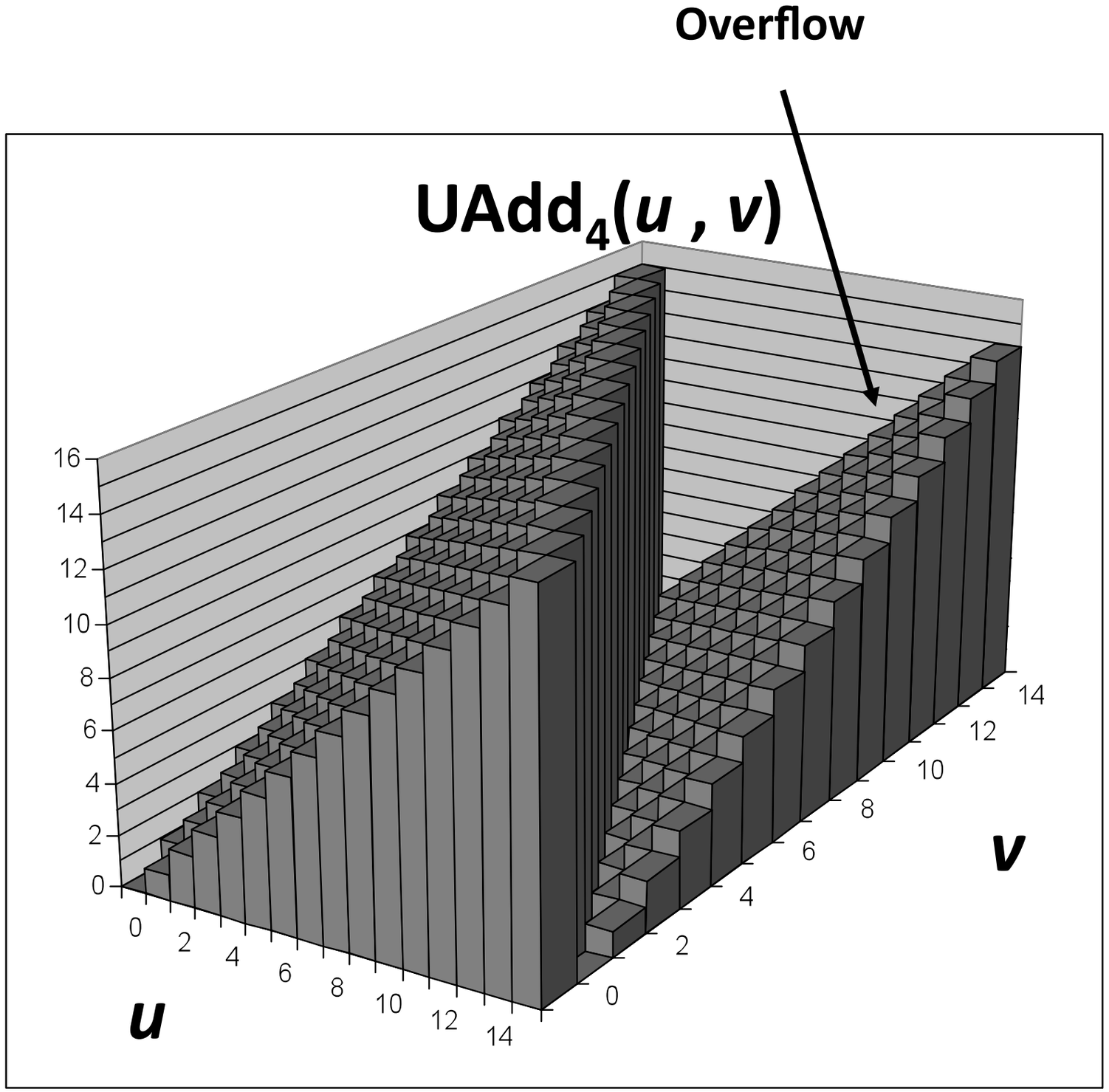
Properties of two's complement

- Same implementation of arithmetic operations as for unsigned
  - E.g., addition, using 4 bits
    - unsigned: `0001` + `1001` = 1 + 9 = 10 = `1010`
    - two's complement: `0001` + `1001` = 1 + -7 = -6 = `1010`
- Only one representation of zero!
  - Simpler to implement operations
- Not symmetric around zero
  - Can represent more negative numbers than positive numbers
- Most common representation of negative integers

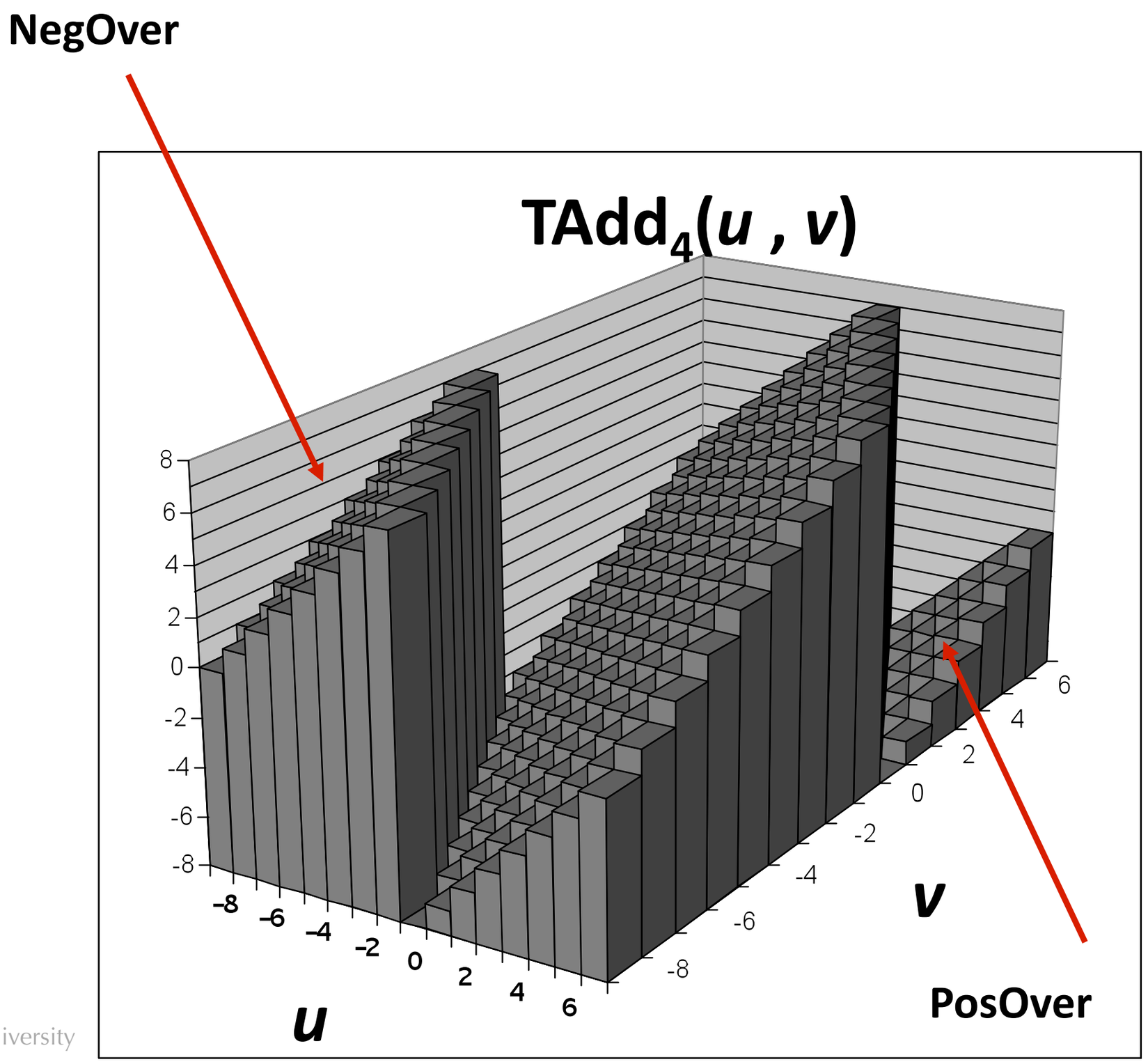
Integer overflow

- Overflow can also occur with negative integers
- With 32 bits, maximum integer expressible in 2's complement is 2^{31}-1 = `0x7fffffff`
- `0x7fffffff` + `0x1` = `0x80000000` = -2^{31}
  - Minimum integer expressible in 32-bit 2's complement
- `0x80000000` + `0x80000000` = `0x0`

Integer overflow

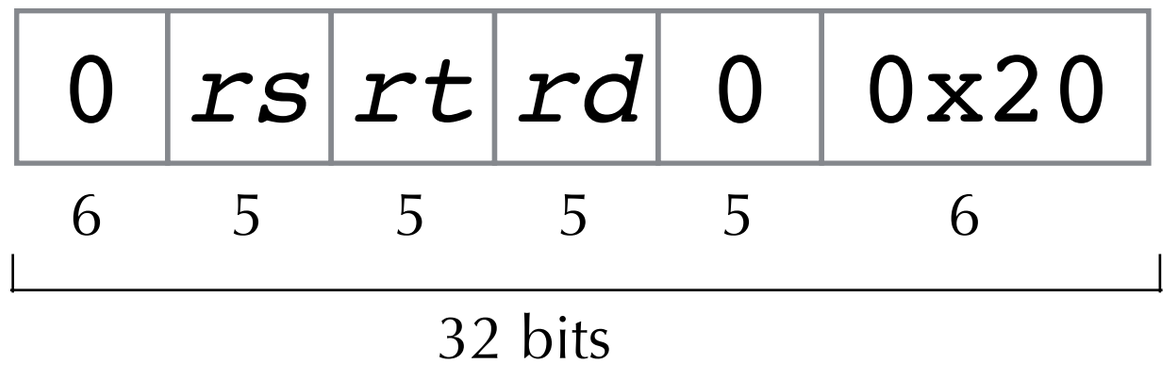


Integer overflow



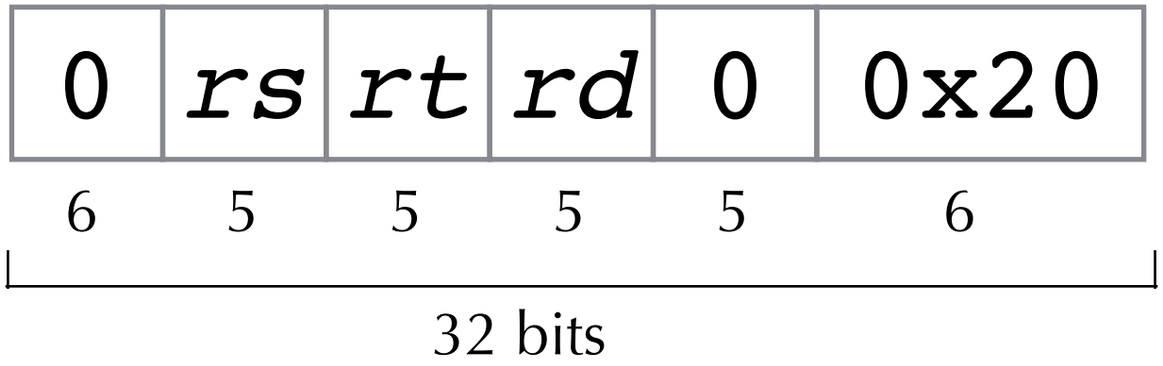
Instruction encodings

- How instructions are represented in 4 bytes
- `add $rd, $rs, $rt`

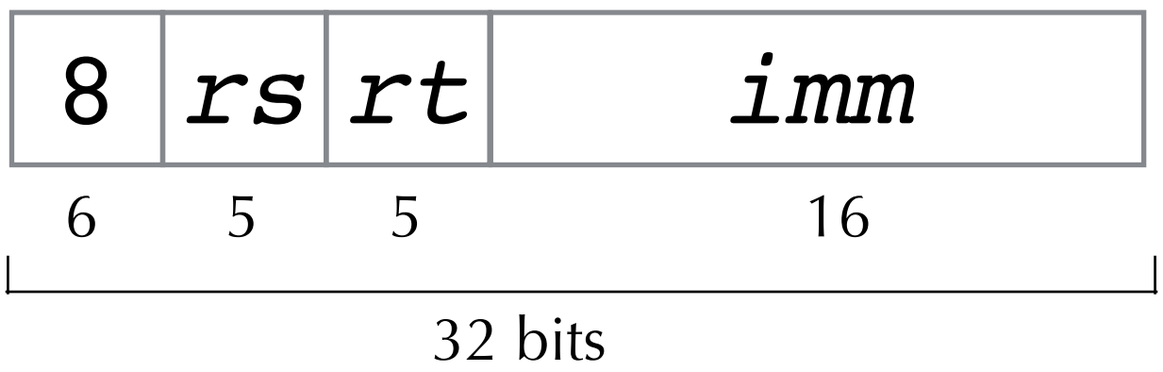


Instruction encodings

- How instructions are represented in 4 bytes
- `add $rd, $rs, $rt`



- `addi $rt, $rs, <imm>`



- More details in the SPIM Simulator manual

Movement

- MIPS has no instruction to move contents of one register to another
  - But assembler provides **pseudo-instructions**
  - `move $rd, $rs`

    becomes `or $rd, $rs, $0`
- Has instruction to load 16-bit immediate values into registers, but not for 32-bit immediate. (Why?)
  - `li $rd, <32-bit-imm>`

    becomes `lui $rd, <hi-16-bits>`

    `ori $rd, $rd, <lo-16-bits>`

Multiply and Divide Instructions

- Instructions to multiply
  - `mul $rd, $rs, $rt`
    multiplies rs and rt (as signed integers), puts result in rd
- Any issues?
- Could overflow...

Multiply and Divide Instructions

- Use two special register `lo` and `hi` (cannot be used as arguments for instructions)
- `mult $`*`rs`*`, $`*`rt`* `multu $`*`rs`*`, $`*`rt`* multiplies rs and rt (as signed/unsigned integers), puts result into `lo` and `hi`
- `mflo $`*`rd`* and `mfhi $`*`rd`* move contents of `lo` and `hi` into register rd
- Also instructions madd, msub, etc. to multiply and add/sub the result to `lo` and `hi`
- Divide operations use `lo` and `hi` to store the quotient and remainder respectively.

Load/Store Instructions

- Instructions to access memory
  - `lw  $rd, <imm>($rs)`
    loads contents of memory address rs+imm into rd
  - `sw  $rs, <imm>($rt)`
    stores register rs into memory address rt+imm
- Only one addressing mode! `<imm>($rs)`
- Traps (fails) if rs+imm is not word-aligned
  - Other instructions to load bytes and half-words

Comparisons

- `slt  $rd, $rs, $rt`
  - Set Less Than
  - rd := (rs < rt), treating rs and rt as signed integers
- `slti  $rd, $rs, <imm16>`
  - Set Less Than Immediate
  - rd := (rs < imm16), treating rs and imm16 as signed integers
- Additionally, unsigned versions: `sltu`, `sltiu`
  - i.e., treating operands as unsigned integers
- Assembler provides pseudo-instructions for `seq`, `sge`, `sgeu`, `sgt`, `sne`, …

Conditional Branching

- `beq $rs,$rt,<imm16>`
  - if $rs == $rt then pc := pc + imm16
- `bne $rs,$rt,<imm16>`
- `b <imm16>` == `beq $0,$0,<imm16>`
- `bgez $rs, <imm16>`
  - if $rs ≥ 0 then pc := pc + imm16
- Also `bgtz`, `blez`, `bltz`
- Pseudo instructions:
  - `b<comp> $rs,$rt, <imm16>`

Labels

- Writing offsets for branches is difficult!
- Assembler lets us use symbolic **labels** instead
- Put a label on an instruction and then can branch to it:

```
LOOP: ...
      bne $3, $2, LOOP
```

- Assembler figures out actual offsets.
  - (How would you implement that?)

Unconditional Jumps

- `j <imm26>`
  - Jump
  - pc := (imm26 << 2)
- `jr  $rs`
  - Jump register
  - pc := $rs
- `jal <imm26>`
  - Jump and link. Used for calling functions. Puts the return address into $31
  - $31 := pc+4 ; pc := (imm26 << 2)
- Also, `jalr` and a few others.
- Again, in practice, we use labels:

```
fact:   ...
main:   ...
        jal fact
```

Other Instructions

- Floating-point (separate registers $fi)
- Traps
- OS-trickery

Back to example

```
int sum(int n) {
  int s = 0;
  for (; n != 0; n--)
    s += n;
  return s;
}
```

```
sum:    ori   $2,$0,$0
        b     test
loop:   add   $2,$2,$4
        subi  $4,$4,1
test:   bne   $4,$0,loop
        jr    $31
```

```
int main() {
  return sum(42);
}
```

```
main:   ori   $4,$0,42
        move  $17,$31
        jal   sum
        jr    $17
```

Slightly better

```
int sum(int n) {
  int s = 0;
  for (; n != 0; n--)
    s += n;
  return s;
}
```

```
sum:    ori   $2,$0,$0
        b     test
loop:   add   $2,$2,$4
        subi  $4,$4,1
test:   bne   $4,$0,loop
        jr    $31
```

```
int main() {
  return sum(42);
}
```

```
main:   ori   $4,$0,42
        jal   sum
```

SPIM Simulator

- We will program to the MIPS *virtual machine* which is provided by the assembler.
  - Lets us use macro instructions, labels, etc.
    - (but we must leave a scratch register for the assembler to do its work)
  - Lets us ignore delay slots.
    - (but then we pay the price of not scheduling those slots.)
- More information about SPIM and the MIPS instruction set in

  "Assemblers, Linkers, and the SPIM Simulator"
  by James Larus
  http://spimsimulator.sourceforge.net/HP_AppA.pdf