EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

FACULTY OF
SCIENCE
Communication Networks

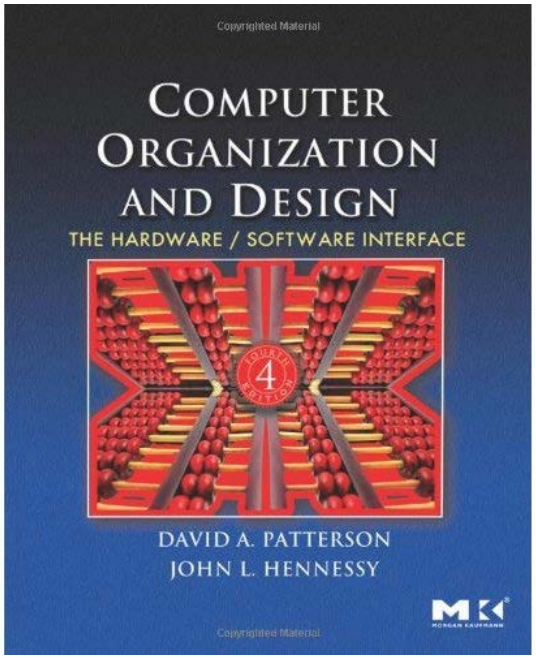
Informatik der Systeme – Chapter 5: Instructions – Language of the Computer

Prof. Dr. Michael Menth

http://kn.inf.uni-tuebingen.de

Acknowledgements

- This set of slides is an adaptation of Prof. Mary Jane Irwin‘s lecture notes, http://www.cse.psu.edu/research/mdl/mji/
- Adapted from Patterson & Hennessy: “Computer Organization and Design”, 4th Edition, © 2008, MK
- German translation: Patterson & Hennessy: “Rechnerorganisation und Rechnerentwurf”

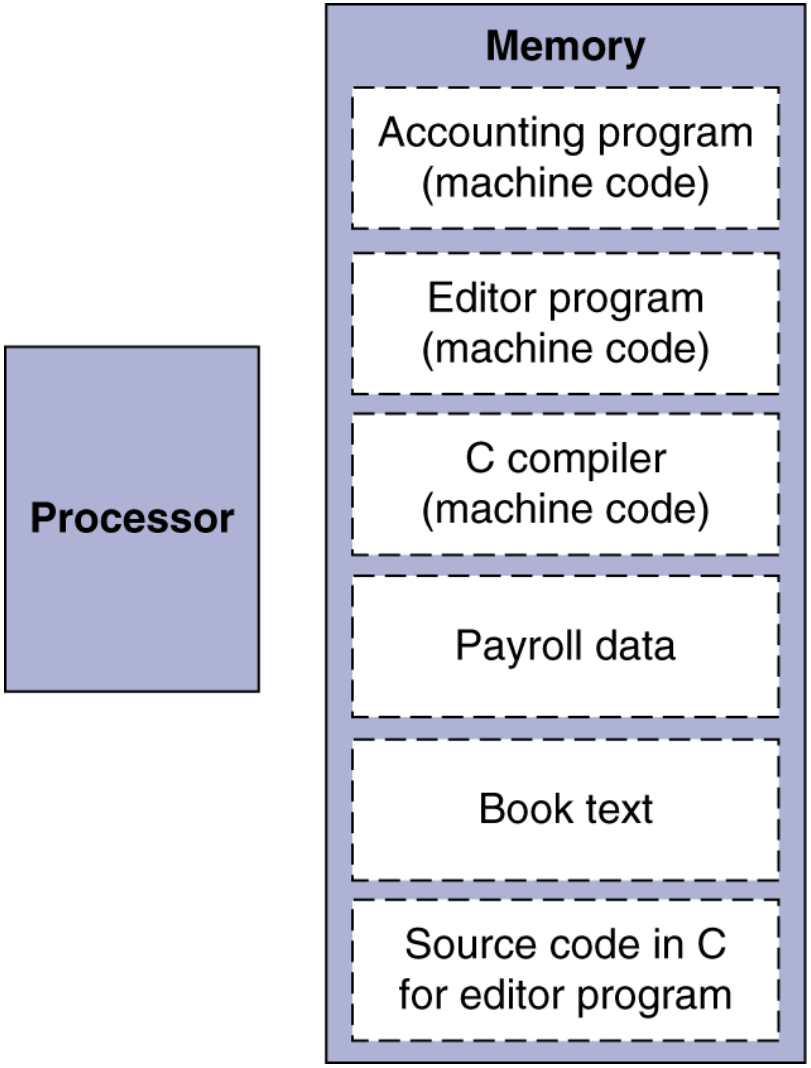


Overview

- Instruction set architecture
- The MIPS ISA
- Calling procedures in MIPS
- Dealing with character sets
- Translating and starting a program
- Effect of compiler optimization
- Other ISAs
- Concluding remarks

Stored Program Computers (Von Neumann Architecture)

- Instructions just like data
  - Represented in binary
  - Stored in memory
  - Programs can be shipped as files of binary numbers
- Programs can operate on programs
  - E.g., compilers, linkers, …
- Binary compatibility
  - Allows compiled programs to work on different computers compatible with the same ISA
  - Leads industry to align around a small number of ISAs



Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets, in particular embedded devices

CISC and RISC

CISC

- Complex Instruction Set Computer
- Single instructions can execute several low-level operations (such as a load from memory, an arithmetic operation, and a memory store) and/or are capable of multi-step operations or addressing modes within single instructions
- E.g. IBM's System/360 through z/Architecture, PDP-11, VAX, Motorola 68k, and x86 family until 80386

RISC

- Reduced Instruction Set Computer
- CPU design strategy based on the insight that simplified instructions can provide higher performance if this simplicity enables much faster execution of each instruction.
- E.g.DEC Alpha, AMD 29k, ARC, ARM, Atmel AVR, MIPS, PA-RISC, Power (including PowerPC), and SPARC
- x86: RISC core with CISC emulation
- ARM processors in smart phones and tablet computers such as the iPad

Overview

- Instruction set architecture
- The MIPS ISA
- Calling procedures in MIPS
- Dealing with character sets
- Translating and starting a program
- Effect of compiler optimization
- Other ISAs
- Concluding remarks

The MIPS Instruction Set

- Used as the example throughout the lecture
- **M**icroprocessor without **I**nterlocked **P**ipeline **S**tages
  - RISC-style ISA, developed since 1981 by John Hennessy at Standford University
  - From 1984 on: MIPS Computer Systems Inc., today MIPS Technologies
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Typical of many modern ISAs

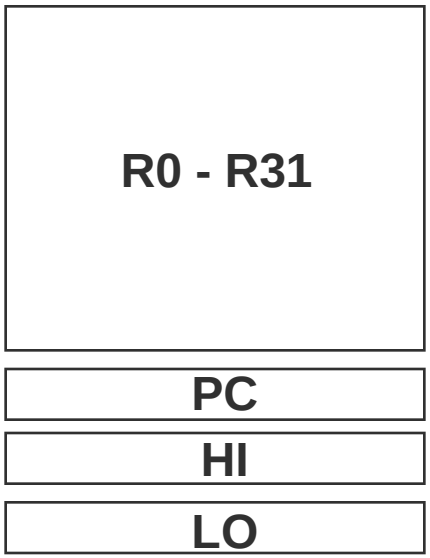
MIPS (RISC) Design Principles

- Simplicity favors regularity
  - Fixed size instructions
  - Small number of instruction formats
  - Opcode always the first 6 bits
- Smaller is faster
  - Limited instruction set
  - Limited number of registers in register file
  - Limited number of addressing modes
- Make the common case fast
  - Arithmetic operands from the register file (load-store machine)
  - Allow instructions to contain immediate operands
- Good design demands good compromises
  - Three instruction formats

MIPS-32 ISA

- Instruction Categories
  - Computational
  - Load/Store
  - Jump and Branch
  - Floating Point
    - Coprocessor
  - Memory Management
  - Special

Registers (32 bits wide)



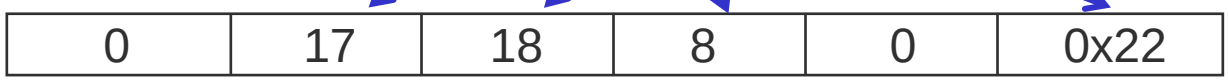
3 Instruction Formats (machine code representation): all 32 bits wide

Aside: MIPS Register Convention

Name	Register Number	Usage	Preserve on call?
$zero	0	constant 0 (hardware)	n.a.
$at	1	reserved for assembler	n.a.
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	yes
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return addr (hardware)	yes

MIPS Arithmetic Instructions

- Each arithmetic instruction in MIPS assembly language performs one operation
- Each specifies exactly three operands that are all contained in the datapath's register file (`$t0,$s1,$s2`)
- Format: destination ← source1 op source2
- Example: `sub $t0, $s1, $s2` (assembler code)
- Conversion into machine language (with example values)



MIPS Instruction Formats and Fields

- MIPS instruction formats define special instruction fields
- **R**-Format
  - For instructions with up to 3 **R**egisters

  - op 6-bits opcode that specifies the operation
  - rs 5-bits register file address of the first source operand
  - rt 5-bits register file address of the second source operand
  - rd 5-bits register file address of the result's destination
  - shamt 5-bits shift amount (for shift instructions)
  - funct 6-bits function code augmenting the opcode
- **I**-Format (later)
  - For instructions with **I**mmediate operands (explicit numbers)
- **J**-Format (later)
  - For **J**ump instructions

R-Format Example

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

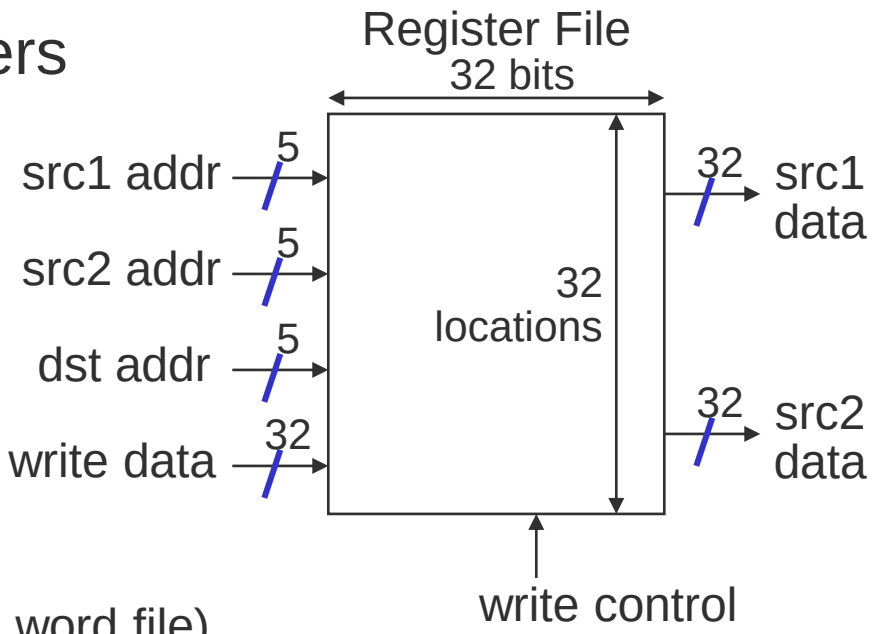
```
add $t0, $s1, $s2
```

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$$00000010001100100100000000100000_2 = 02324020_{16}$$

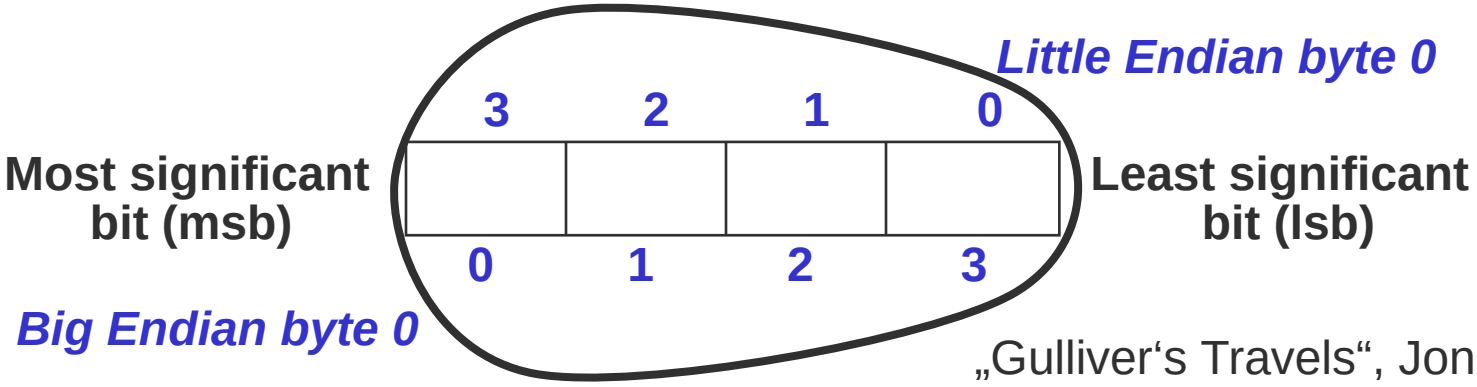
MIPS Register File

- Holds thirty-two 32-bit registers
  - Two read ports and
  - One write port
- Registers are
  - Faster than main memory
    - But register files with more locations are slower (e.g., a 64 word file could be as much as 50% slower than a 32 word file)
    - Read/write port increase impacts speed quadratically
  - Easier for a compiler to use
    - E.g., (A·B) – (C·D) – (E·F) can do multiplies in any order vs. stack
  - Can hold variables so that
    - Code density improves (since register are named with fewer bits than a memory location)



Memory

- Used for composite data (variables, arrays, structures, dynamic data, …)
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Most architectures address individual bytes in memory
- Alignment restriction – the memory address of a word must be on natural word boundaries (a multiple of 4 in MIPS-32)
- Big Endian: leftmost byte is word address
  - Examples: IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Little Endian: rightmost byte is word address
  - Examples: Intel 80x86, DEC Vax, DEC Alpha (Windows NT)



„Gulliver‘s Travels“, Jonathan Swift

MIPS Memory Access Instructions

- MIPS has two basic data transfer instructions for accessing memory

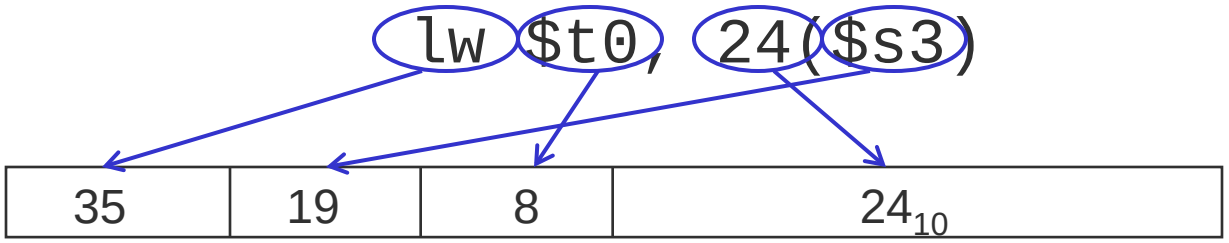
```
lw    $t0, 4($s3) #load word from memory
sw    $t0, 8($s3) #store word to memory
```

- I-Format – for immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Memory address: sum of offset and base address in rs

- The data is loaded into (lw) or stored from (sw) a register in the register file ($t0) – a 5 bit address
- The memory address – a 32 bit address – is formed by adding the contents of the base address register ($s3) to the 16 bit offset value
  - A 16-bit field meaning access is limited to memory locations within a region of $\pm 2^{13}$ or 8,192 words ($\pm 2^{15}$ or 32,768 bytes) of the address in the base register

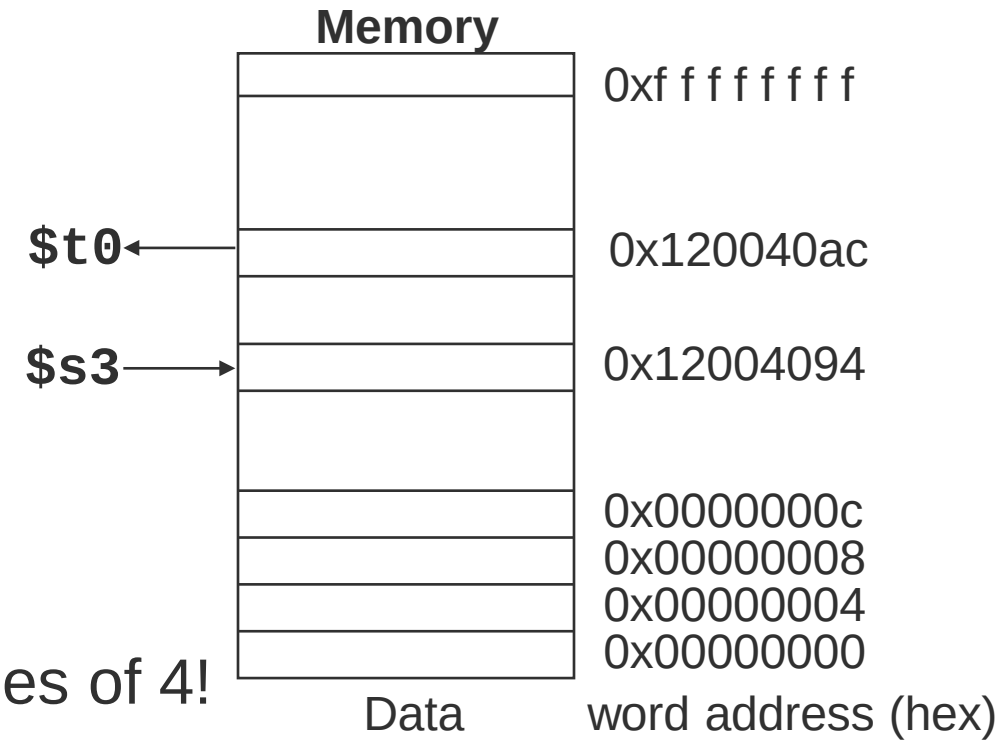
Example: Load Instruction

- Load/store instructions use I-Format



24_{10} + $s3 =

. . . 0001 1000
+ . . . 1001 0100
. . . 1010 1100 =
0x120040ac



Word addresses are multiples of 4!

Aside: Loading and Storing Bytes

- MIPS provides special instructions to move bytes

```
lb    $t0, 1($s3)  #load byte from memory
sb    $t0, 6($s3)  #store byte to  memory
```

- What 8 bits get loaded and stored?
  - Load byte places the byte from memory in the rightmost 8 bits of the destination register
    - What happens to the other bits in the register?
  - Store byte takes the byte from the rightmost 8 bits of a register and writes it to a byte in memory
    - What happens to the other bits in the memory word?

MIPS Immediate Instructions

- Small constants are used often in typical code
- Possible approaches?
  - Put “typical constants” in memory and load them
  - Create hard-wired registers (like $zero) for constants like 1
  - Have special instructions that contain constants !

```
addi   $s0, $s0, 4 #$s0 = $s0 + 4
slti   $t0, $s2, 15  #$t0 = 1 if $s2<15
```

  - Immediate instructions use I-Format

- The constant is kept inside the instruction itself!
  - Immediate format limits values to the range from -2^{15} to $+2^{15}-1$

Aside: How About Larger Constants?

- We'd also like to be able to load a 32 bit constant into a register, for this we must use two instructions
- A new "load upper immediate" instruction

```
lui $t0, 1010101010101010
```

16	0	8	1010101010101010_2

- Then must get the lower order bits right, use

```
ori $t0, $t0, 1010101010101010
```

1010101010101010	0000000000000000
0000000000000000	1010101010101010
1010101010101010	1010101010101010

Logical Operations

- Instructions for bitwise manipulation
  - Useful for extracting and inserting groups of bits (e.g. bytes or 8-bit characters) in a 32-bit word
  - Shift left logical (sll): shift left and fill with 0 bits
  - Shift right logical (srl): shift right and fill with 0 bits
  - Useful for fast multiplication of unsigned numbers by 2^i

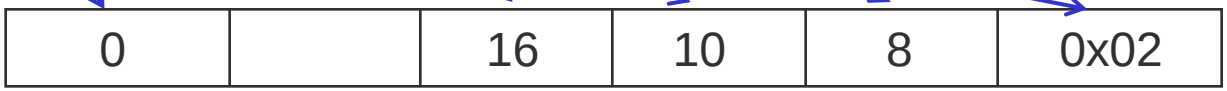
Operation	C	Java	MIPS
Shift left	<<	<<	`sll`
Shift right	>>	>>	`srl`
Bitwise AND	&	&	`and, andi`
Bitwise OR	|	|	`or, ori`
Bitwise NOT	~	~	`nor`

MIPS Shift Operations

- Shifts move all the bits in a word left or right

```
sll $t2, $s0, 8    #$t2 = $s0 << 8 bits
srl $t2, $s0, 8    #$t2 = $s0 >> 8 bits
```

- Shift operations use R-Format

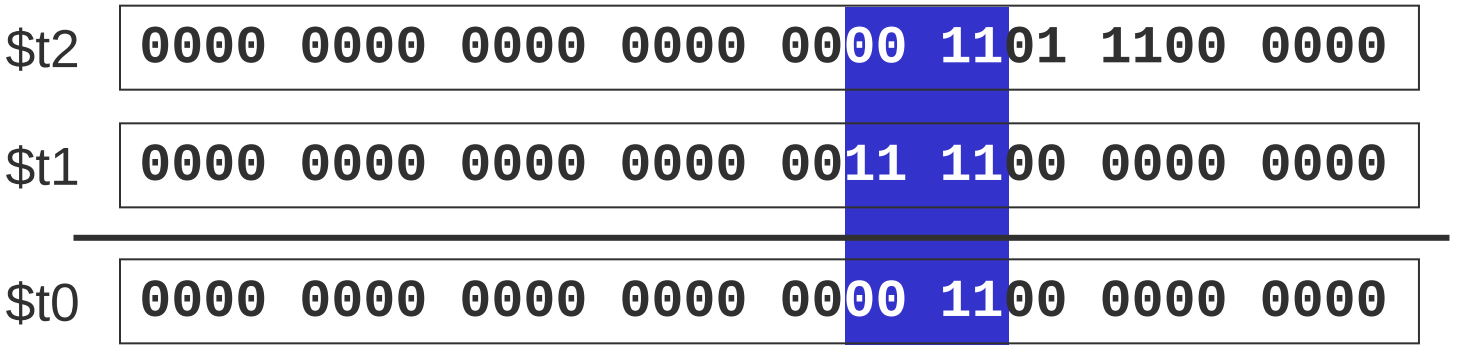


  - Shift amount (shamt) = 8
  - Notice that a 5-bit shamt field is enough to shift a 32-bit value 2^5 – 1 or 31 bit positions

AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

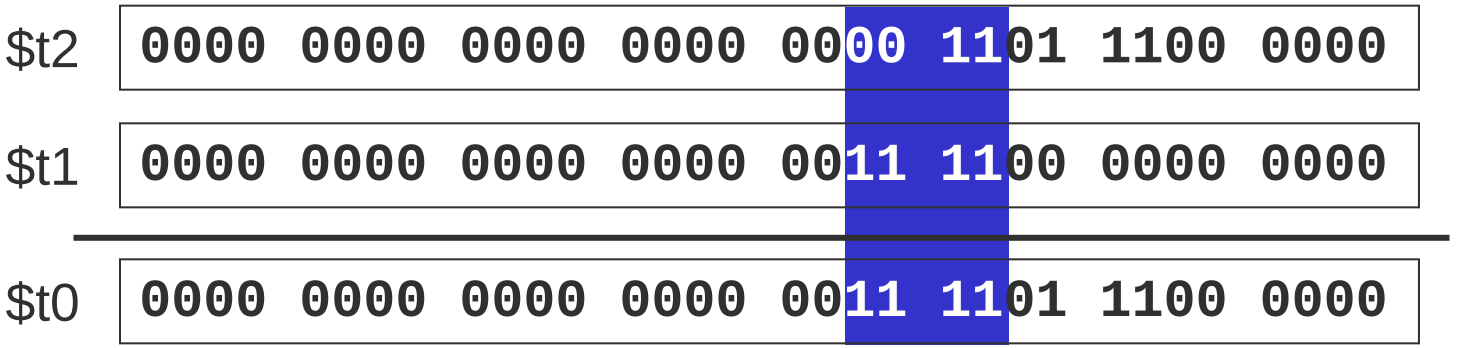
```
and $t0, $t1, $t2
```



OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

```
or $t0, $t1, $t2
```



NOT Operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction
  - a NOR b == NOT (a OR b)

Register 0: always read as zero

```
nor $t0, $t1, $zero
```

$zero	0000 0000 0000 0000 0000 0000 0000 0000
$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	1111 1111 1111 1111 1100 0011 1111 1111

MIPS Logical Operations

- There are a number of bit-wise logical operations in the MIPS ISA

```
and $t0, $t1, $t2     #$t0 = $t1 & $t2
or  $t0, $t1, $t2     #$t0 = $t1 | $t2
nor $t0, $t1, $t2     #$t0 = not($t1 | $t2)
```

  - Logical operations with two source operands use R-Format

0	9	10	8	0	0x24

```
andi $t0, $t1, 0xFF00 #$t0 = $t1 & ff00
ori  $t0, $t1, 0xFF00 #$t0 = $t1 | ff00
```

  - Logical operations with immediate operand use I-Format

0x0D	9	8	0xFF00

MIPS Control Flow Instructions

- MIPS conditional branch instructions

```
bne $s0, $s1, Lbl     #go to Lbl if $s0≠$s1
beq $s0, $s1, Lbl     #go to Lbl if $s0=$s1
```

- Example

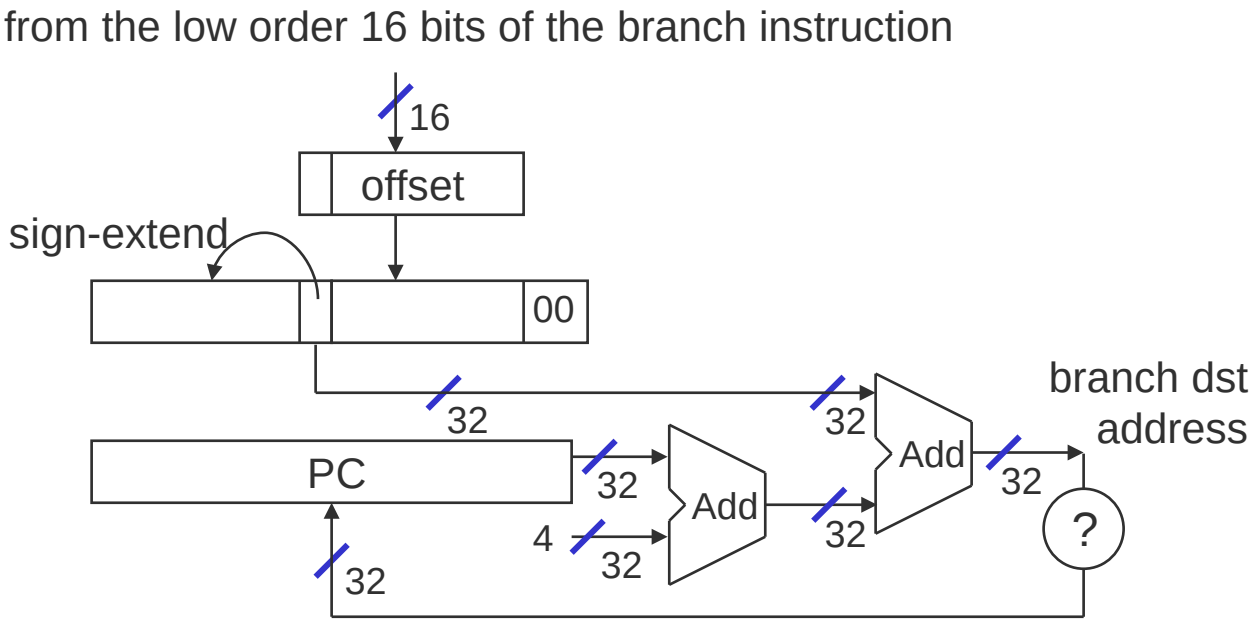
```
        if (i==j) h = i + j;
        bne $s0, $s1, Lbl1
        add $s3, $s0, $s1
Lbl1:   ...
```

- Assembler takes care of fixing the label
  - Lbl1 in the instruction is approximately the offset to the current instruction address (more next slide)
  - “Lbl1:” marks the corresponding instruction in code
- Conditional branch uses I-Format

0x05	16	17	16 bit offset

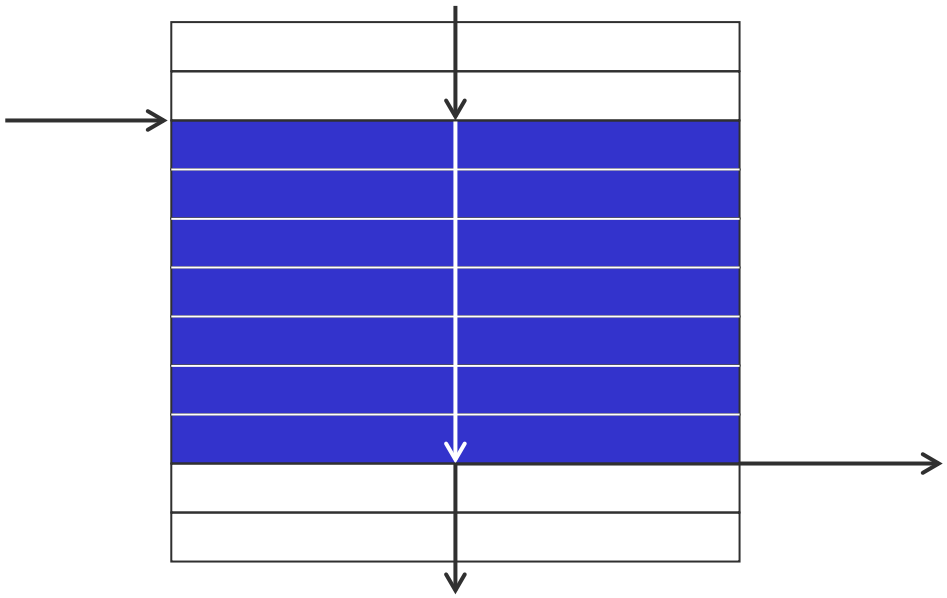
Specifying Branch Destinations

- Example: `bne $s0, $s1, Lbl1`
- Means: next instruction address is (PC+4)+4*(value of Lbl1)
  - PC register automatically incremented (PC+4) during fetch cycle
- Limits the branch distance to -2^{15} to $+2^{15}-1$ (word) instructions from the (instruction after the) branch instruction
  - But most branches are local anyway



Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)
- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks



More Complex Branch Instructions

- We have `beq`, `bne`, but what about other kinds of branches (e.g., branch-if-less-than)?
- Set on less than instruction

```
slt $t0, $s0, $s1      # if $s0 < $s1    then
                       # $t0 = 1         else
                       # $t0 = 0
```

  - Instruction format (R format):

0	16	17	8		0x24

- Immediate operand available

```
slti $t0, $s0, 25      # if $s0 < 25 then $t0=1 ...
```

Signed vs. Unsigned Comparison

- Unsigned versions of `slt` and `slti` (u stands for unsigned)

```
sltu $t0, $s0, $s1    # if $s0 < $s1 then $t0=1 ...
sltiu $t0, $s0, 25    # if $s0 < 25 then $t0=1 ...
```

- Example
  - $s0 = **1111 1111 1111 1111 1111 1111 1111 1111**
  - $s1 = **0000 0000 0000 0000 0000 0000 0000 0001**
  - `slt  $t0, $s0, $s1  # signed`
    - –1 < +1 ⇒ $t0 = 1
  - `sltu $t0, $s0, $s1  # unsigned`
    - +4,294,967,295 > +1 ⇒ $t0 = 0

Aside: More Branch Instructions

- Why not `blt`, `bge`, etc? Hardware for <, ≥, … slower than =, ≠
  - Combining with branch involves more work per instruction, requiring a slower clock – all instructions penalized!
- Can use `slt`, `beq`, `bne`, and the fixed value of 0 in register `$zero` to create other conditions
  - Less than

```
blt $s1, $s2, Label
```

```
slt  $at, $s1, $s2        # $at set to 1 if
bne  $at, $zero, Label    # $s1 < $s2
```

  - Less than or equal to `ble $s1, $s2, Label`
  - Greater than `bgt $s1, $s2, Label`
  - Greater than or equal to `bge $s1, $s2, Label`
- Such branches are included in the instruction set as pseudo instructions
  - Recognized (and expanded) by the assembler
  - Therefore, the assembler needs a reserved register (`$at`)

Bounds Check Shortcut

- Treating signed numbers as if they were unsigned gives a low-cost way of checking if $0 \leq x < y$ (index out of bounds for arrays)
  - $y \rightarrow$ `$t2`, $x \rightarrow$ `$s1`

```
sltu $t0, $s1, $t2     # $t0 = 1 if
                       # $s1 < $t2 (max)
                       # and $s1 >= 0 (min)
beq $t0, $zero, IOOB   # go to IOOB if
                       # $t0 = 0
```

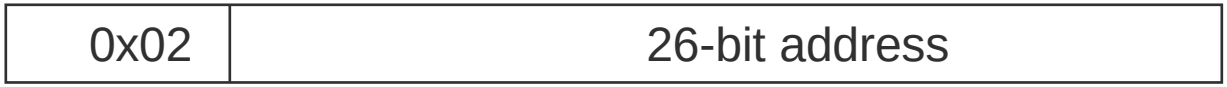
- The key is that negative integers in two's complement look like large numbers in unsigned notation. Thus, an unsigned comparison of $x < y$ also checks if x is negative as well as if x is less than y.

Other Control Flow Instructions

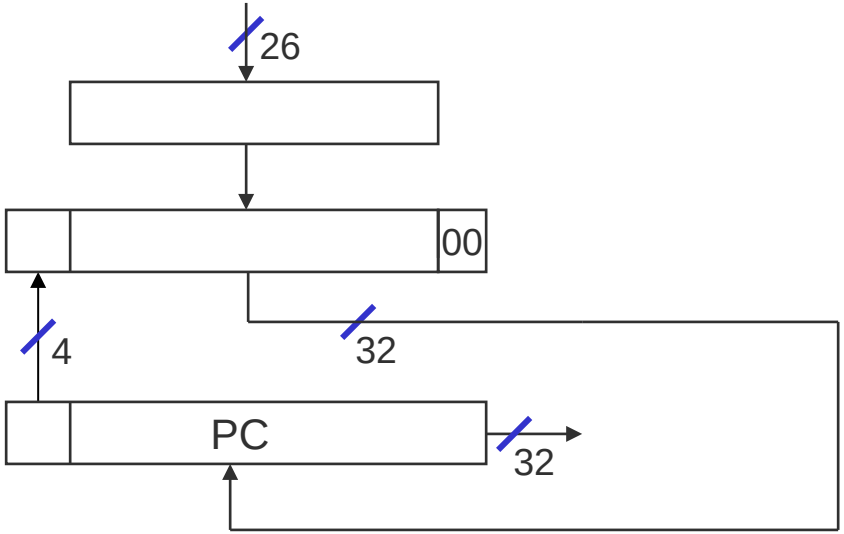
- MIPS also has an unconditional branch instruction or jump instruction:

```
j   label                # go to label
```

- Jump instruction uses the J-Format



from the low order 26 bits of the jump instruction



Compiling Loop Statements

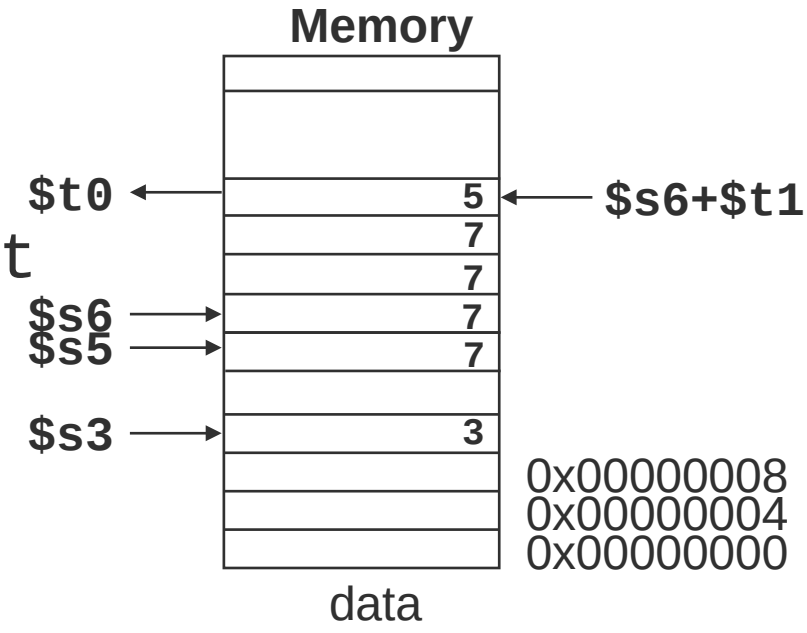
- C code:

```
while (save[i] == k) i += 1;
```

  - i in $s3, k in $s5, address of save in $s6

- Compiled MIPS code:

```
Loop:sll  $t1, $s3, 2
     add  $t1, $t1, $s6
     lw   $t0, 0($t1)
     bne  $t0, $s5, Exit
     addi $s3, $s3, 1
     j    Loop
Exit:…
```



Aside: Conditional Branching Far Away

- What if a conditional branch destination is further away than can be captured in 16 bits?
- The assembler comes to the rescue – it inserts an unconditional jump to the branch target and inverts the condition
- Example:

```
        beq   $s0, $s1, L1
```

- Becomes:

```
        bne   $s0, $s1, L2
        j     L1
L2:     …
```

Instructions for Accessing Procedures

- MIPS procedure call instruction

```
jal   ProcedureAddress  #jump and link
```

  - Saves PC+4 in register `$ra` to have a link to the next instruction for the procedure return
  - Uses the J-Format

0x03	26 bit address

- Then can do procedure return with a

```
jr    $ra                    #return
```

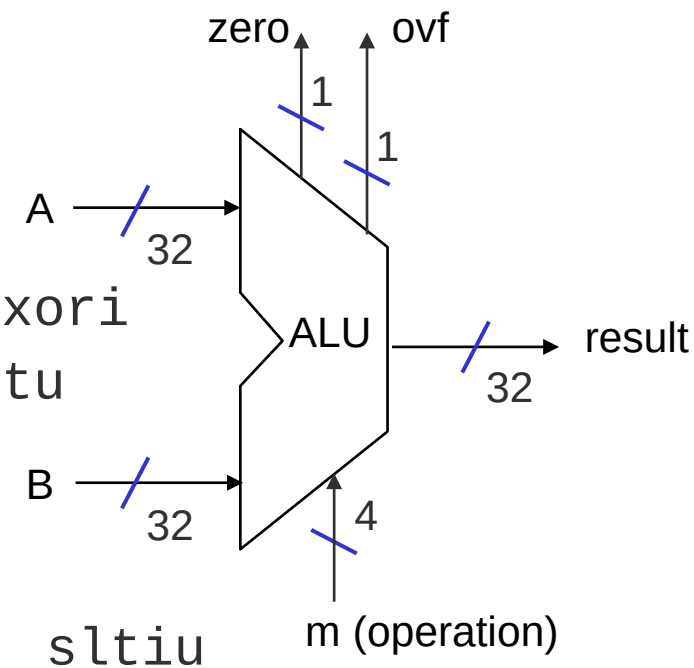
  - Can also be used for computed jumps
    - e.g., for case/switch statements
  - Uses the R-Format

0	31				0x08

MIPS Arithmetic Logic Unit (ALU)

- Must support the arithmetic/logic operations of the ISA

```
add, addi, addiu, addu
sub, subu
mult, multu, div, divu
sqrt
and, andi, nor, or, ori, xor, xori
beq, bne, slt, slti, sltiu, sltu
```



- With special handling for
  - Sign extend – `addi, addiu, slti, sltiu`
  - Zero extend – `andi, ori, xori`
  - Overflow detection – `add, addi, sub`

Dealing with Overflow

- Overflow occurs when the result of an operation cannot be represented in 32-bits, i.e., when the sign bit contains a value bit of the result and not the proper sign bit
- When adding operands with different signs or when subtracting operands with the same sign, overflow can never occur

Operation	Operand A	Operand B	Result indicating overflow
A + B	≥ 0	≥ 0	< 0
A + B	< 0	< 0	≥ 0
A - B	≥ 0	< 0	< 0
A - B	< 0	≥ 0	≥ 0

- MIPS signals overflow with an exception (aka interrupt) – an unscheduled procedure call where the EPC contains the address of the instruction that caused the exception

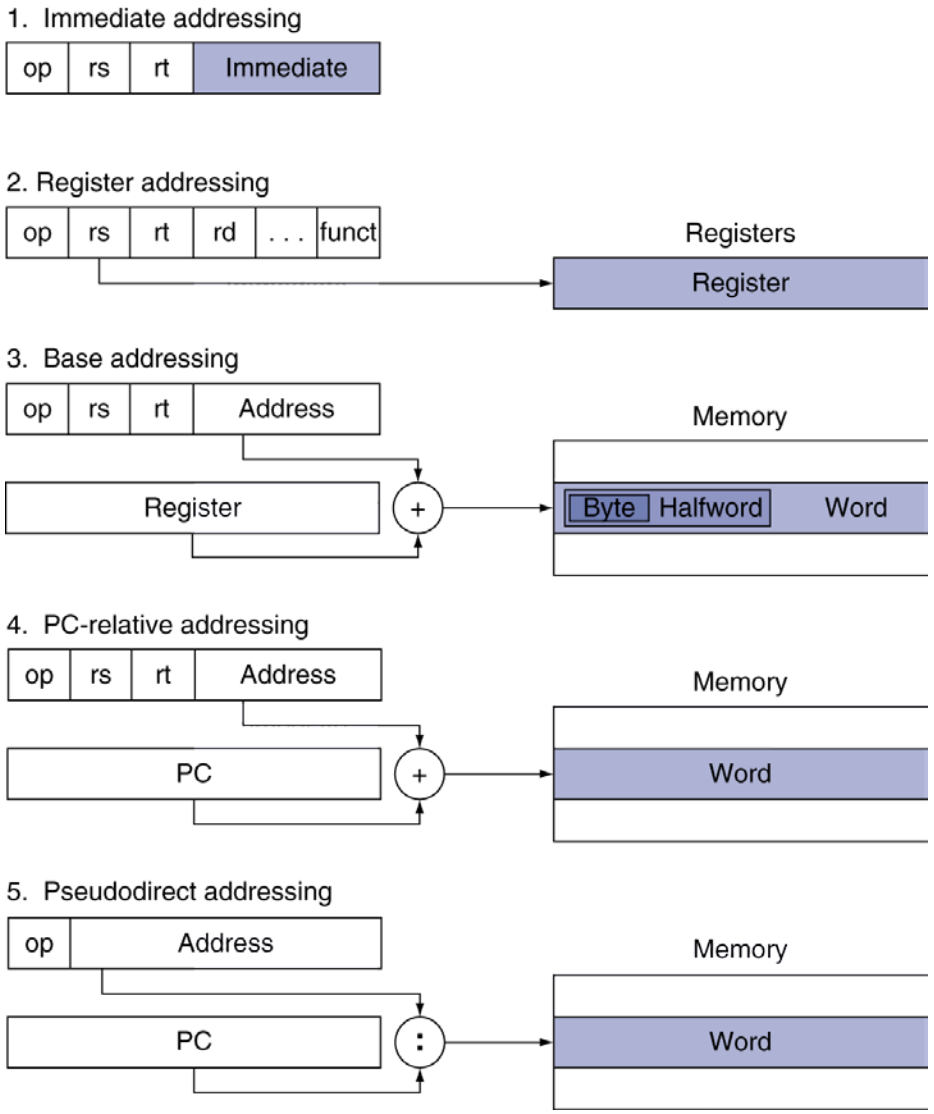
Summary of Operands in MIPS

- Register operands
  - Direct access
  - Used by arithmetic instructions
  - Only 32 registers to enable fast access in implementations
- Memory operands
  - Only accessible via load and store instructions
    - ⇒ Compilers must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!
- Immediate operands
  - Explicit numbers in instructions
- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
  - Useful for common operations

```
add $t2, $s1, $zero
```

Summary of Addressing Modes in MIPS

- Immediate addressing
  - `addi $s0 $s1 1`
- Register addressing
  - `add $s0 $s1 $s2`
- Base addressing
  - `sw $s0 $s3(4)`
- PC-relative addressing
  - `bne $s0 $s1 loop`
- Pseudo-direct adressing
  - `j loop`



Overview

- Instruction set architecture
- The MIPS ISA
- Calling procedures in MIPS
- Dealing with character sets
- Translating and starting a program
- Effect of compiler optimization
- Other ISAs
- Concluding remarks

Six Steps in Execution of a Procedure

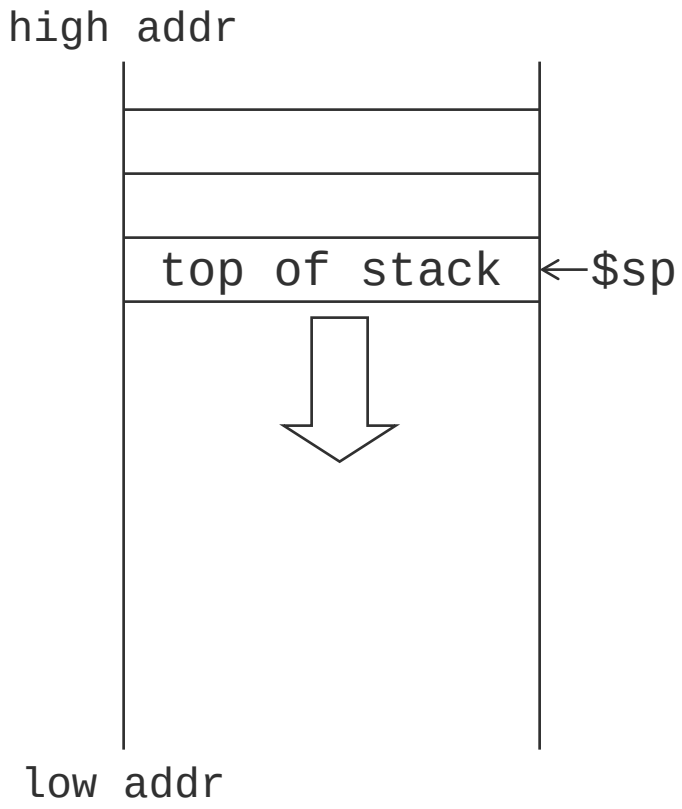
1. Main routine (caller) places parameters in a place where the procedure (callee) can access them
   - `$a0` – `$a3`: four argument registers
2. Caller transfers control to the callee
3. Callee acquires the storage resources needed
4. Callee performs the desired task
5. Callee places the result value in a place where the caller can access it
   - `$v0` – `$v1`: two value registers for result values
6. Callee returns control to the caller
   - `$ra`: one return address register to return to the point of origin

Spilling Registers into Memory

- What if the callee needs to use more registers than allocated to argument and return values?
  - Callee uses a stack – a last-in-first-out queue
- Stack pointer `$sp` (`$29`)
  - One of the general registers
  - Used to address the stack
  - Stack "grows" from high address to low address
  - Add data onto the stack – push data on stack at new `$sp`

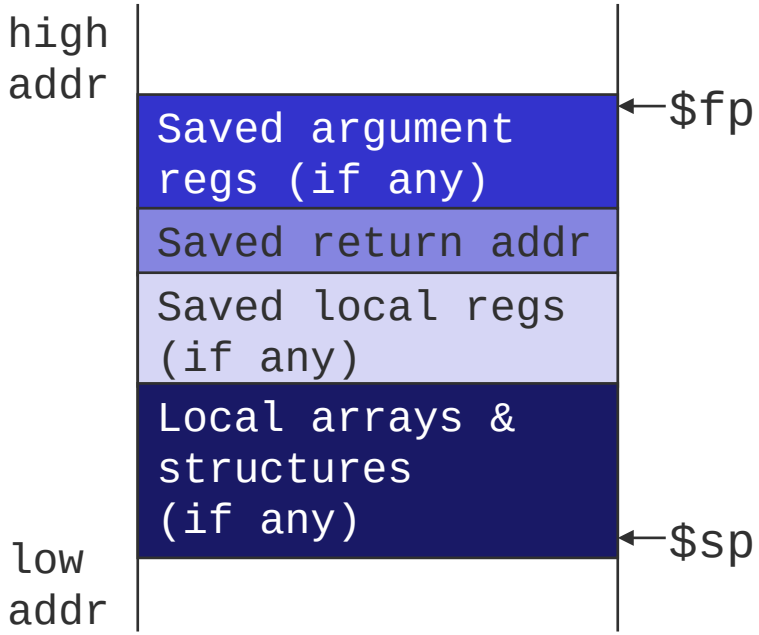
    ```
    $sp = $sp – 4
    ```
  - Remove data from the stack – pop data from stack at `$sp`

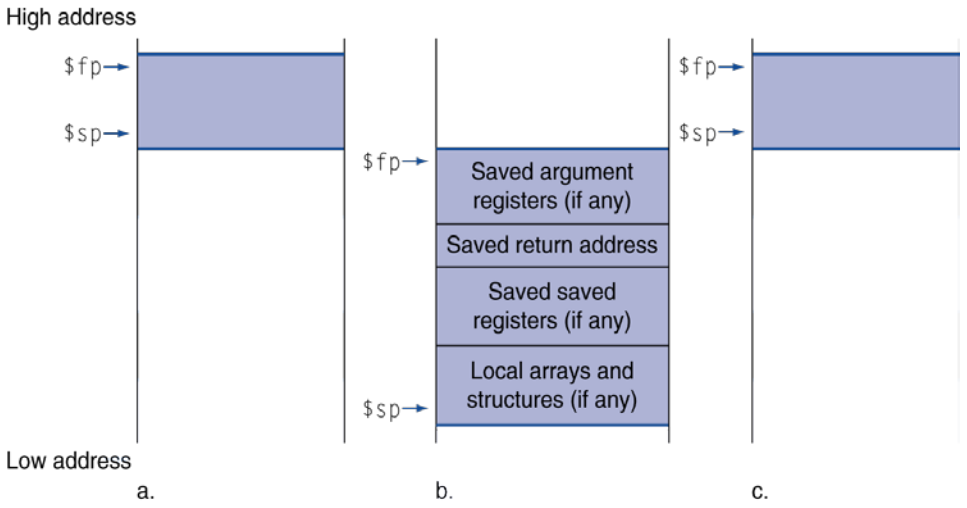
    ```
    $sp = $sp + 4
    ```



Aside: Allocating Space on the Stack

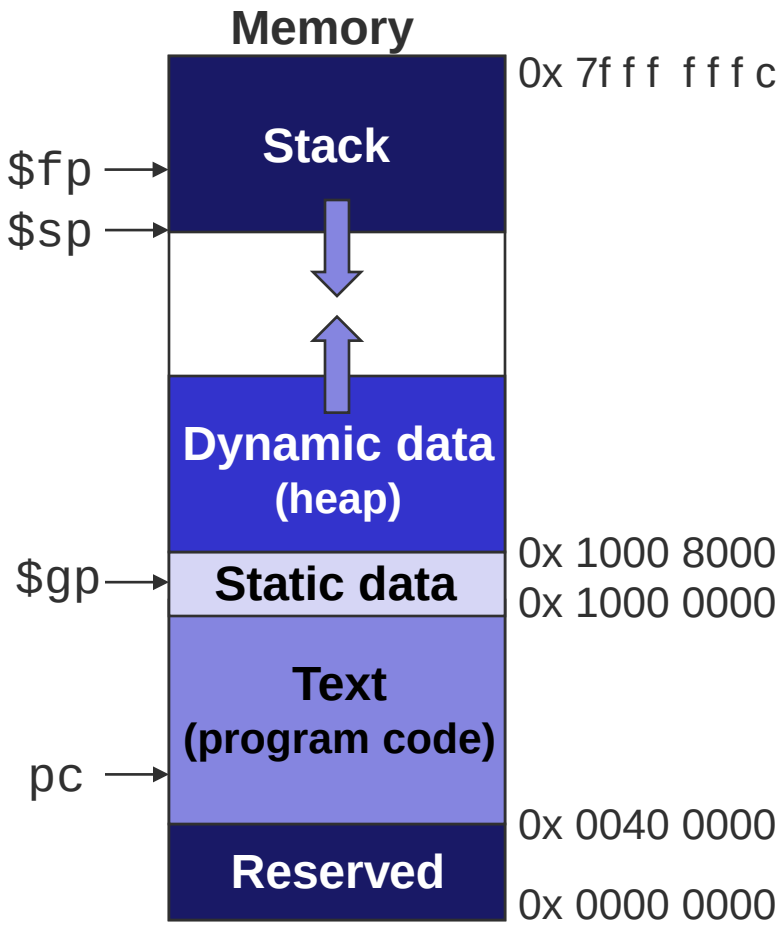
- Procedure frame
  - The segment of the stack containing a procedure's saved registers and local variables
  - Aka activation record
- The frame pointer (`$fp`)
  - Points to the first word of the frame of a procedure
  - Provides a stable "base" register for the procedure
    - `$fp` is initialized using `$sp` on a call and
    - `$sp` is restored using `$fp` on a return
  - Right figure: before, during, and after procedure call





Aside: Allocating Space on the Heap

- Static data segment
  - For constants and other static variables (e.g., arrays)
  - Global pointer `$gp` used to address static data with positive and negative 16-bit offset (`lw, sw`)
- Dynamic data segment
  - For structures that grow and shrink (e.g., linked lists)
  - Aka heap
  - Dealing with dynamic data in C
    - Allocate space on the heap with `malloc()` (memory allocation)
    - Free it with `free()`



Leaf- and Non-Leaf Procedures

- Procedures
  - Must save register values `$sx` on the stack if it makes use of these registers
  - Must restore them before return
  - Can overwrite arguments `$ax` and temporaries `$tx`
- Leaf procedure
  - Does not call other procedures
- Non-leaf procedure
  - Calls other procedures (nested calls)
  - For nested call, caller needs to save on the stack
    - Its return address
    - Any arguments `$ax` and temporaries `$tx` needed after the call
  - Registers restored from the stack after the call

Leaf Procedure Example

- C code

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;}
```

- MIPS code
  - Arguments g, …, j in `$a0, …, $a3`
  - f in `$s0` (hence, need to save $s0 on stack)
  - Result in `$v0`

```
leaf_example:
  addi $sp, $sp, -4      # Save $s0 on stack
  sw   $s0, 0($sp)
  add  $t0, $a0, $a1     # Procedure body
  add  $t1, $a2, $a3
  sub  $s0, $t0, $t1
  add  $v0, $s0, $zero   # Result
  lw   $s0, 0($sp)       # Restore $s0
  addi $sp, $sp, 4
  jr   $ra               # Return
```

Non-Leaf Procedure Example

- C code

```
int fact (int n){
  if (n < 1) return 1;
  else return n * fact(n - 1);}
```

Recursive procedure: special case of a non-leaf procedure that calls itself again.

- MIPS code: argument n in $a0, result in $v0

```
fact:
    addi $sp, $sp, -8      # adjust stack for 2 items
    sw   $ra, 4($sp)       # save return address
    sw   $a0, 0($sp)       # save argument
    slti $t0, $a0, 1       # test for n < 1
    beq  $t0, $zero, L1
    addi $v0, $zero, 1     # if so, result is 1
    addi $sp, $sp, 8       #   pop 2 items from stack
    jr   $ra               #   and return
L1: addi $a0, $a0, -1      # else decrement n
    jal  fact              # recursive call
    lw   $a0, 0($sp)       # restore original n
    lw   $ra, 4($sp)       #   and return address
    addi $sp, $sp, 8       # pop 2 items from stack
    mul  $v0, $a0, $v0     # multiply to get result
    jr   $ra               # and return
```

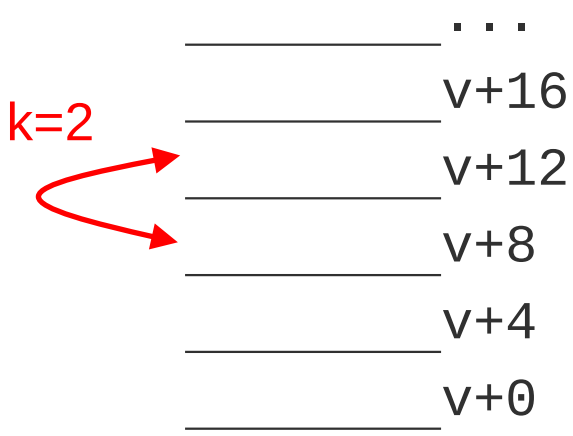
Runtime Example (Requires Animation)

Function calls	Stack occupation	Stack Pointer	Calculation of return values
		<- $sp	$v0=6
fact(3)	$ra		$v0=$v0*$a0=2*3=6
	$a0=3	<- $sp	$v0=2
fact(2)	$ra		$v0=$v0*$a0=1*2=2
	$a0=2	<- $sp	$v0=1
fact(1)	$ra		$v0=$v0*$a0=1*1=1
	$a0=1	<- $sp	$v0=1
fact(0)	$ra		$v0=1
	$a0=0	<- $sp	

Example: “swap” in C

- Illustrates use of assembly instructions for a C bubble (exchange) sort function
- Swap procedure (leaf)

```
void swap(int v[], int k)
{
  int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```



Example: “swap” in MIPS Assembler

- v in $a0, k in $a1, temp in $t0

```
swap: sll $t1, $a1, 2   # $t1 = k * 4
      add $t1, $a0, $t1 # $t1 = v+(k*4)
                        #   (address of v[k])
      lw $t0, 0($t1)    # $t0 (temp) = v[k]
      lw $t2, 4($t1)    # $t2 = v[k+1]
      sw $t2, 0($t1)    # v[k] = $t2 (v[k+1])
      sw $t0, 4($t1)    # v[k+1] = $t0 (temp)
      jr $ra            # return to calling routine
```

Example: “sort” in C

- Non-leaf (calls swap)
  - v: array to be sorted
  - n: size of array to be sorted

```
void sort (int v[], int n)
  {
    int i, j;
    for (i = 1*; i < n; i += 1) {
      for (j = i – 1;
           j >= 0 && v[j] > v[j + 1];
           j -= 1) {
        swap(v,j);
      }
    }
  }
```

*Book says i=0, but i=1 should be more efficient.

Runtime Example

- n=4 array elements; array positions: 0, 1, 2, 3
- v=[4,3,2,1] (example)

- i=1
  - j=0: v=[3,4,2,1]

- i=2
  - j=1: v=[3,2,4,1]
  - j=0: v=[2,3,4,1]

- i=3
  - j=2: v=[2,3,1,4]
  - j=1: v=[2,1,3,4]
  - j=0: v=[1,2,3,4]

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Example: Procedure Body of "sort" in MIPS

- v in `$a0`, n in `$a1`, i in `$s0`, j in `$s1`

```
          move $s2, $a0            # save $a0 into $s2
          move $s3, $a1            # save $a1 into $s3
          addi $s0, $zero, 1       # i = 1
for1tst:  slt  $t0, $s0, $s3       # $t0 = 1 if $s0 < $s3 (i < n)
          beq  $t0, $zero, exit1   # go to exit1 if $s0 ≥ $s3 (i ≥ n)
          addi $s1, $s0, -1        # j = i - 1
for2tst:  slti $t0, $s1, 0         # $t0 = 1 if $s1 < 0 (j < 0)
          bne  $t0, $zero, exit2   # go to exit2 if $s1 < 0 (j < 0)
          sll  $t1, $s1, 2         # $t1 = j * 4
          add  $t2, $s2, $t1       # $t2 = v + (j * 4)
          lw   $t3, 0($t2)         # $t3 = v[j]
          lw   $t4, 4($t2)         # $t4 = v[j + 1]
          slt  $t0, $t4, $t3       # $t0 = 1 if $t4 < $t3 (v[j+1]<v[j])
          beq  $t0, $zero, exit2   # go to exit2 if $t4 ≥ $t3
          move $a0, $s2            # 1st param of swap is v (old $a0)
          move $a1, $s1            # 2nd param of swap is j
          jal  swap                # call swap procedure
          addi $s1, $s1, -1        # j -= 1
          j    for2tst             # jump to test of inner loop
exit2:    addi $s0, $s0, 1         # i += 1
          j    for1tst             # jump to test of outer loop
```

Move params

Outer loop

Inner loop

Pass params & call

Inner loop

Outer loop

Example: Full Procedure of “sort” in MIPS

```
sort:   addi $sp,$sp, –20      # make room on stack for 5 registers
        sw $ra, 16($sp)        # save $ra on stack
        sw $s3,12($sp)         # save $s3 on stack
        sw $s2, 8($sp)         # save $s2 on stack
        sw $s1, 4($sp)         # save $s1 on stack
        sw $s0, 0($sp)         # save $s0 on stack
        …                      # procedure body
        …
exit1:  lw $s0, 0($sp)         # restore $s0 from stack
        lw $s1, 4($sp)         # restore $s1 from stack
        lw $s2, 8($sp)         # restore $s2 from stack
        lw $s3,12($sp)         # restore $s3 from stack
        lw $ra,16($sp)         # restore $ra from stack
        addi $sp,$sp, 20       # restore stack pointer
        jr $ra                 # return to calling routine
```

Overview

- Instruction set architecture
- The MIPS ISA
- Calling procedures in MIPS
- Dealing with character sets
- Translating and starting a program
- Effect of compiler optimization
- Other ISAs
- Concluding remarks

Character Data

- Byte-encoded character sets
  - ASCII: 128 characters, 95 graphic, 33 control
  - Latin-1 (aka ISO 8859-1): 256 characters, ASCII, +96 more graphic characters
- Unicode (16 bit)
  - Most of the world's alphabets, plus symbols
  - 17x16-bit character sets (1114112 codepoints)
  - Java uses 16 bit Unicode (UTF-16)
  - UTF: Unicode Transformation Format, also UCS Transformation Format (UCS = Universal Character Set)
    - Variable-length encoding: special characters require several bytes (1-4 bytes)
    - Represent all Unicode characters
    - Several variants: UTF-8, UTF-16, UTF-32, UTF-EBCDIC, …
- EBCDIC (Extended Binary Coded Decimals Interchange Code)
  - Mostly used on mainframes

String Representations

- Array of characters, first position(s) used to indicate array length
  - Used in Java
  - First word indicates length
  - 16-bit Unicode characters used
- Structure, consisting of a length field and an array of that length
- Array of characters, end of string determined by value 0
  - Example: ASCII value 0 means symbol „null“ while value 48 means character „0“ $\Rightarrow$ character „0“ can be used in strings
  - Principle used in C
  - Character representation depends on machine and operating system
    - ASCII, OEM, ANSI, ISO-Latin-1, Unicode, EBCDIC,…
    - Most of them use 1 byte for character representation

Byte/Halfword Operations

- Could use bitwise operations

- MIPS byte/halfword load/store
  - String processing is a common case
  - Sign extend to 32 bits in rt

```
lb rt, offset(rs)
lh rt, offset(rs)
```

  - Zero extend to 32 bits in rt

```
lbu rt, offset(rs)
lhu rt, offset(rs)
```

  - Store just rightmost byte/halfword

```
sb rt, offset(rs)
sh rt, offset(rs)
```

String Copy Example in C

- C code (naïve) for copying null-terminated strings
  - Copies string `y[]` into string `x[]`
  - `do … while(cond)` in C language corresponds to
  - `repeat … until(not(cond))` in other programming languages

```
void strcpy (char x[], char y[]){
   int i;
   i=-1;
   do{
       i=i+1;
       x[i]=y[i];
   }while (y[i]!='\0')
}
```

String Copy Example in MIPS

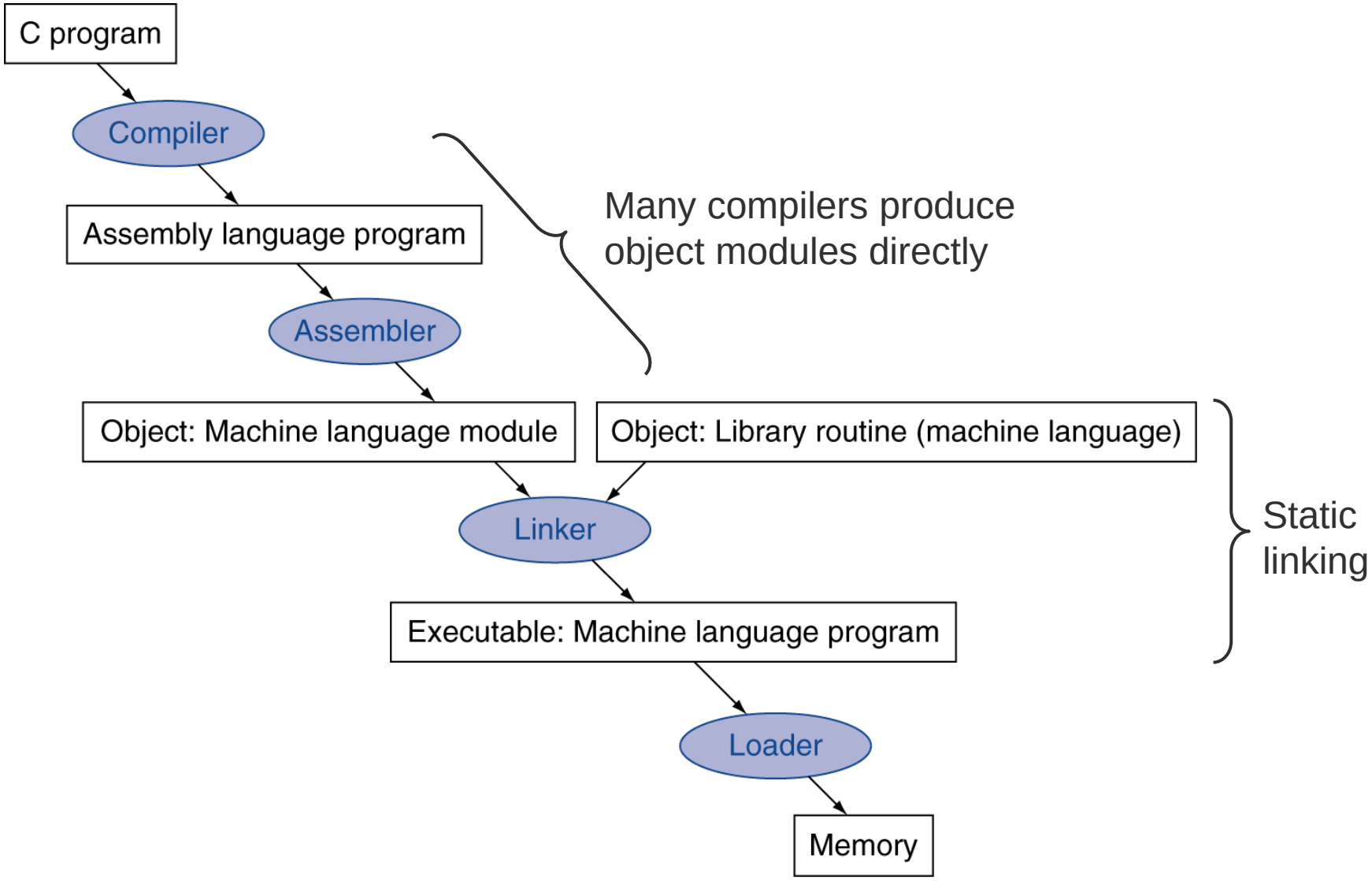
- Addresses of `x, y` in `$a0, $a1`, value of `i` in `$s0`
- Character `'\0'` has value `0` in ASCII

```
strcpy:
    addi $sp, $sp, -4       # adjust stack for 1 item
    sw   $s0, 0($sp)        # save $s0
    addi  $s0, $zero, -1    # i = -1
# start of do … while statement
do: addi $s0, $s0, 1        # i = i + 1 (bytes, not words!)
    add  $t1, $s0, $a1      # addr of y[i] in $t1
    lbu  $t2, 0($t1)        # $t2 = y[i]
    add  $t3, $s0, $a0      # addr of x[i] in $t3
    sb   $t2, 0($t3)        # x[i] = y[i]
    bne  $t2, $zero, do     # continue loop if y[i] != 0
# end of do … while statement
    lw   $s0, 0($sp)        # restore saved $s0
    addi $sp, $sp, 4        # pop 1 item from stack
    jr   $ra                # and return
```

Overview

- Instruction set architecture
- The MIPS ISA
- Calling procedures in MIPS
- Dealing with character sets
- Translating and starting a program
- Effect of compiler optimization
- Other ISAs
- Concluding remarks

Translation and Startup



Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one

- Pseudoinstructions
  - Can be used in assembly language
  - Need to be translated into other instructions for translation into machine code
  - Example

```
move $t0, $t1          →    add $t0, $zero, $t1
blt  $t0, $t1, L       →    slt $at, $t0, $t1
                            bne $at, $zero, L
```

  - `$at` (register 1): assembler temporary

Producing an Object Module

- Assembler (or compiler)
  - Translates program into machine instructions
  - Provides information for building a complete program from the pieces

- Object module
  - Header
    - Described contents of object module
  - Text segment
    - Translated instructions (machine code)
  - Static data segment
    - Data allocated for the life of the program
  - Relocation info
    - For contents that depend on absolute location of loaded program
  - Symbol table
    - Remaining labels (e.g. external refs)
  - Debug info
    - For associating with source code

Linking Object Modules

- Programs consist of several procedures
- Procedures may be compiled and assembled independently
- These object modules need to be linked together
- Problem
  - Machine code needs to be relocated
  - External labels pointing to other procedures are not yet resolved
- That‘s the job of the linker or link editor (Binder)
- Linker produces an executable image
  - Merges segments
  - Resolve labels (determine their addresses)
  - Patch location-dependent and external labels/refs
- Could leave location dependencies for fixing by a relocating loader
- But with virtual memory, no need to do this
  - Program can be loaded into absolute location in virtual memory space

Loading a Program

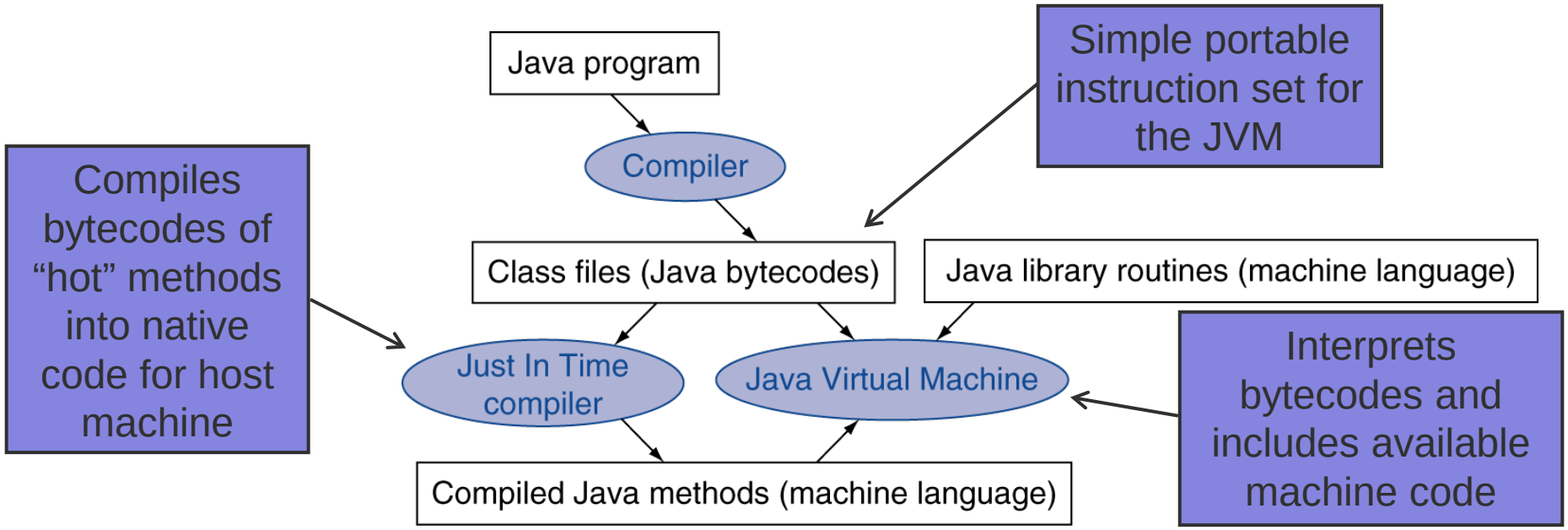
- Load from image file on disk into memory

1. Read header to determine segment sizes
2. Create virtual address space
3. Copy text and initialize data into memory
   - Or set page table entries so they can be faulted in
4. Set up arguments on stack
5. Initialize registers (including $sp, $fp, $gp)
6. Jump to startup routine
   - Copy arguments to $a0, … and call main
   - When main returns, do syscall “exit”

Dynamic Linking

- Only link/load library procedure when it is called
  - Requires procedure code to be relocatable
  - Avoids image bloat caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions
  - Object module of such a library: dynamically linked library (DLL)
- First DLLs: all procedures of a library linked, requiring again large memory space
- Lazy linkage: links procedures at first call
  - More effort at first call
  - Less memory space needed

Starting Java Applications



- Interpreter = program simulating an ISA
- JVM interpretes Java bytecode
- Much slower than machine code

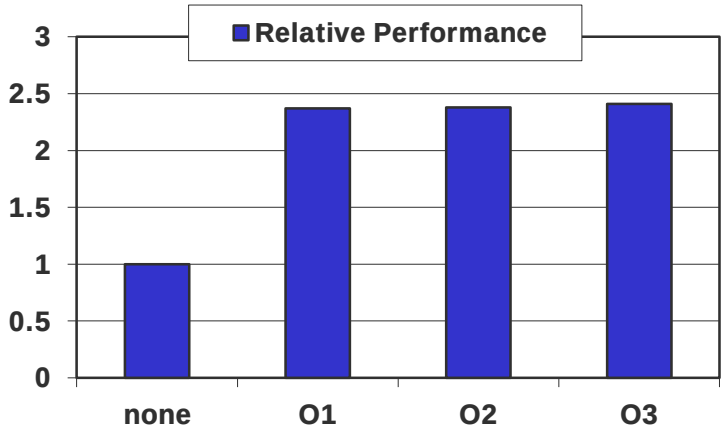
- JIT compiles Java bytecode into machine code during interpretation and saves it for future execution
- JVM also calls machine code from libraries and machine code bytecode compiled by JIT

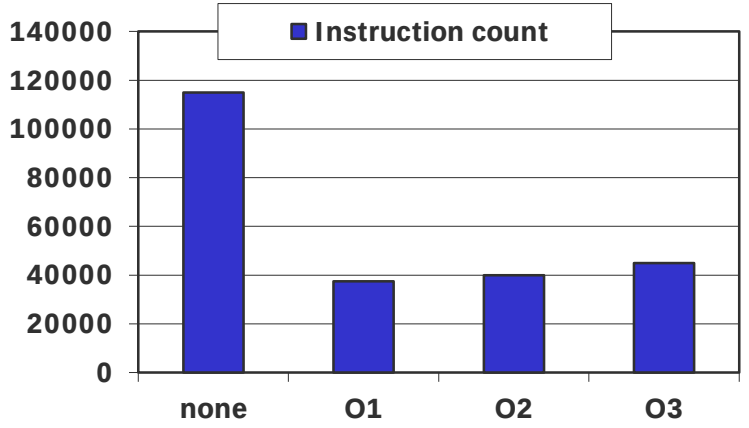
Overview

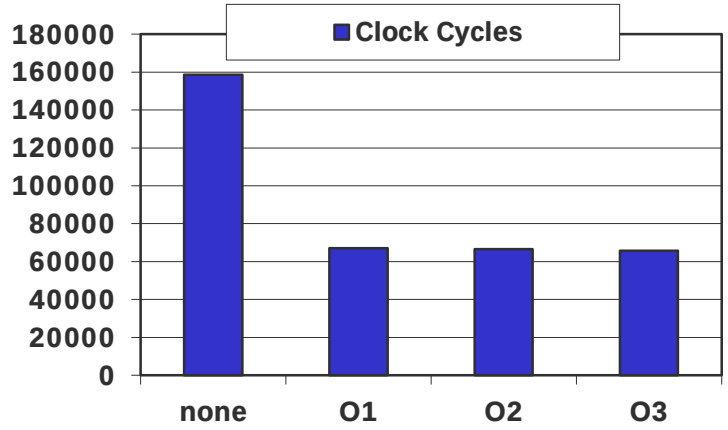
- Instruction set architecture
- The MIPS ISA
- Calling procedures in MIPS
- Dealing with character sets
- Translating and starting a program
- Effect of compiler optimization
- Other ISAs
- Concluding remarks

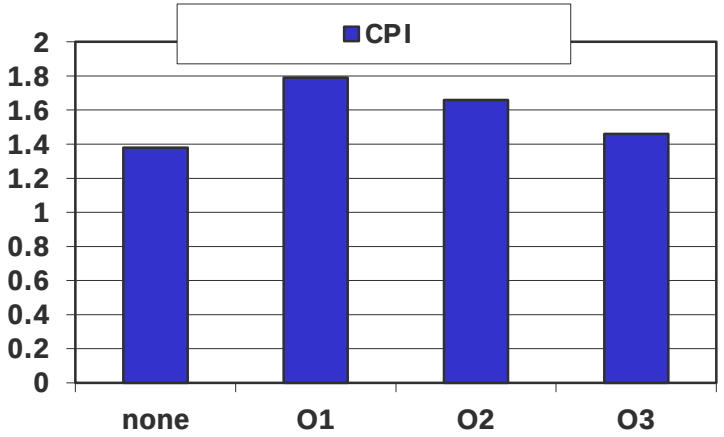
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux





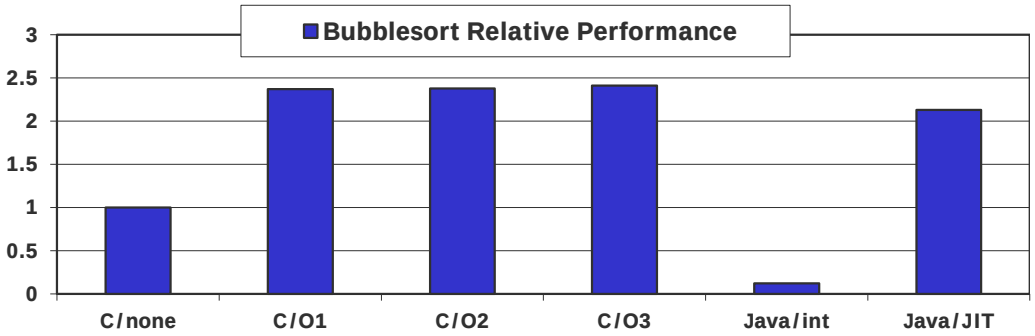


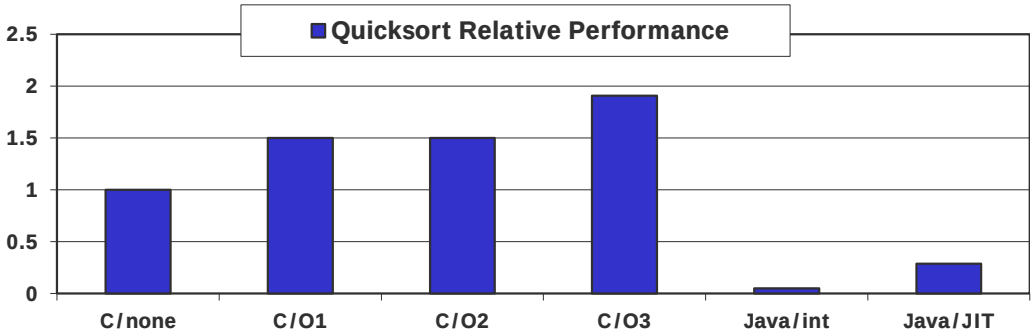


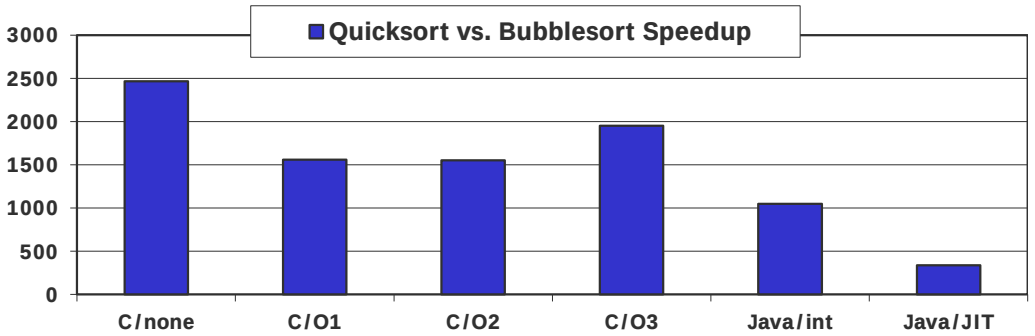
- The unoptimized code has the best CPI, the O1 version has the lowest instruction count, but the O3 version is the fastest. Why?

Effect of Language and Algorithm

- Sort 100,000 words
  - with the array initialized to random values
- on a Pentium 4
  - with a 3.06 GHz clock rate,
  - a 533 MHz system bus,
  - with 2 GB of DDR SDRAM,
  - using Linux version 2.4.20







Lessons Learned

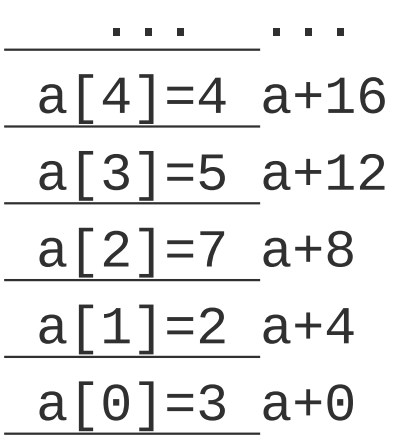
- Instruction count and CPI are not good performance indicators in isolation
- Compiler optimizations are sensitive to the algorithm
- Java/JIT compiled code is significantly faster than JVM interpreted
  - Comparable to optimized C in some cases
- Nothing can fix a dumb algorithm!

Inline Expansion or Inlining

- Replaces a function call with the body of the function to be called (the callee)
- A compiler may do that before translation to assembler code
- Saves the effort of
  - Passing arguments and return values
  - Jumping
  - Saving and restoring registers
- May improve time usage at runtime
- May possibly increase the final size of the binary executable
  - Larger source code
- Widely used optimization technique in compilers

Pointers in C

- int b; c=b+5
  - b is a variable for an integer
  - &b is the memory address of that variable
- int *x; x=&b; c=*x+5;
  - x is a variable for a address of a memory location holding an integer (**pointer**)
  - *x returns the value of that location (**dereferencing**)
- Pointer arithmetic
  - „sizeof(int)“ is size of an integer in bytes
  - x++: pointer x is increased by sizeof(int), not the value of *x!
- Connection with arrays: int a[]
  - Several items of same type consecutively stored in memory
  - int *x=a; base address array
  - Value and address of i-th position in array
    - a[i], &a[i]; *(x+i), (x+i)



Arrays vs. Pointers

- Indexing general arrays (e.g. of structs) involves
  - Multiplying index by element size
  - Adding to array base address
- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity
- More efficient code possible with pointers (see next slide)
- But compiler optimization
  - “strength reduction” effects usage of shift instead of multiply by 2^n
  - “elimination of induction variables” converts usage of array indexing to usage of pointers
- Recommendation
  - Write more readable code using array indexing
  - Let the compiler do the tricky stuff

Example: Clearing an Array

- Arrays and pointers lead to almost the same assembly code
- But blue instructions are required to be in the loop only for arrays

```
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

```
clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p < &array[size];
       p = p + 1)
    *p = 0;
}
```

```
       move $t0,$zero   # i = 0
loop1: sll $t1,$t0,2    # $t1 = i * 4
       add $t2,$a0,$t1  # $t2 =
                        #   &array[i]
       sw $zero, 0($t2) # array[i] = 0
       addi $t0,$t0,1   # i = i + 1
       slt $t3,$t0,$a1  # $t3 =
                        #   (i < size)
       bne $t3,$zero,loop1 # if (…)
                           # goto loop1
```

```
       move $t0,$a0     # p = & array[0]
       sll $t1,$a1,2    # $t1 = size * 4
       add $t2,$a0,$t1  # $t2 =
                        #   &array[size]
loop2: sw $zero,0($t0)  # Memory[p] = 0
       addi $t0,$t0,4   # p = p + 4
       slt $t3,$t0,$t2  # $t3 =
                        #(p<&array[size])
       bne $t3,$zero,loop2 # if (…)
                           # goto loop2
```

Overview

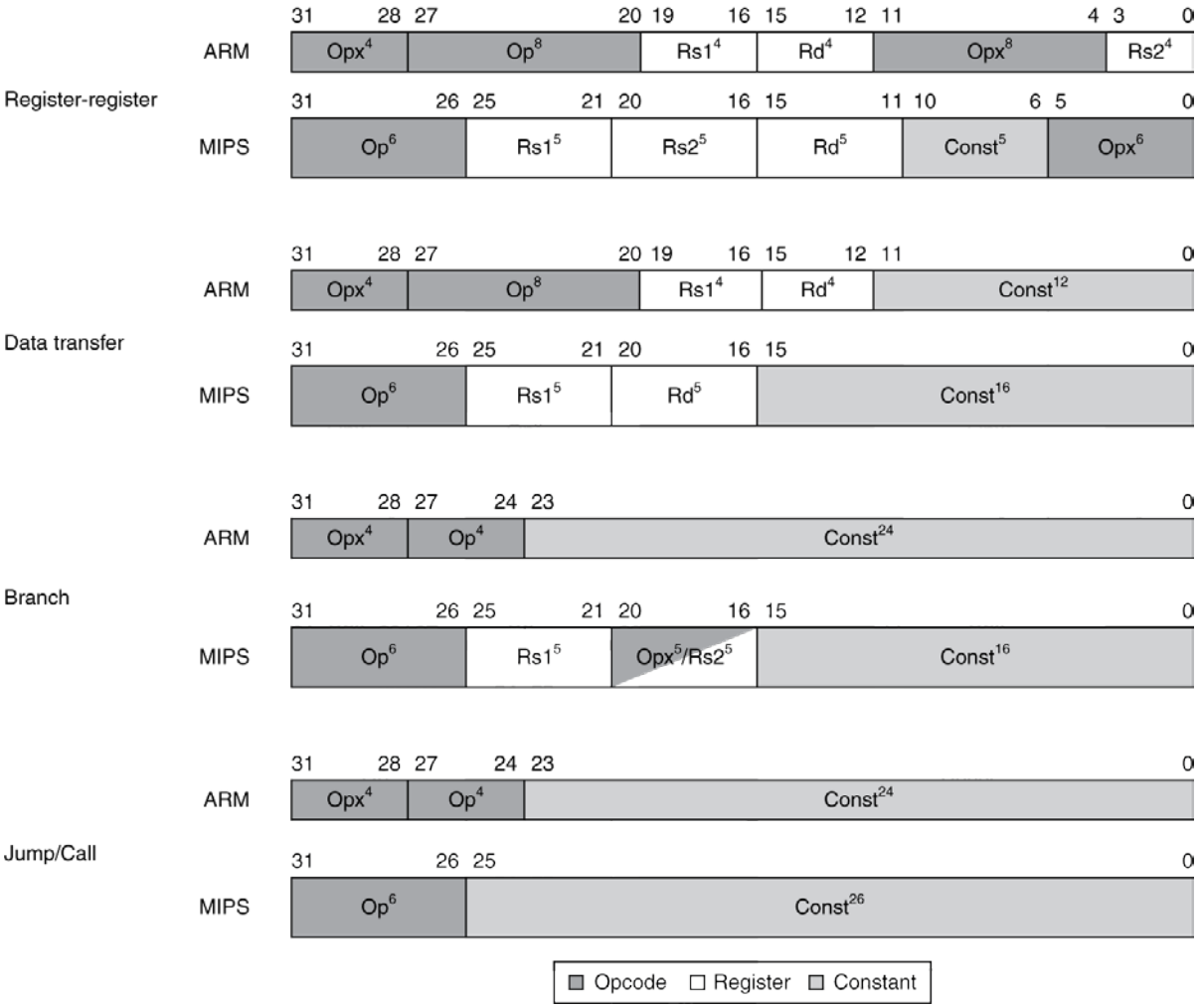
- Instruction set architecture
- The MIPS ISA
- Calling procedures in MIPS
- Dealing with character sets
- Translating and starting a program
- Effect of compiler optimization
- Other ISAs
- Concluding remarks

ARM & MIPS Similarities

- ARM (Advanced RISC Machines)
- The most popular embedded core, widely used in smartphones
- Basic set of instructions similar to MIPS

	ARM	MIPS
Date announced	1985	1985
Instruction size	32 bits	32 bits
Address space	32-bit flat	32-bit flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Registers	15 × 32-bit	31 × 32-bit
Input/output	Memory mapped	Memory mapped

Instruction Encoding



- ARM uses different principle for branch instructions

The Intel x86 ISA

- IA-32: Intel Architecture 32 bit (32 bit x86 architecture)
  - Ex. 80386 / 386 / i386
- General purpose registers (GPRs) from 80386 on
- Words
  - 16 bit = word, 32 bit = double word, 64 bit = quad word
- Instructions
  - Significantly more than in MIPS
  - More complex
  - Varying length instruction format (1 – 15 bytes)
- Complex instruction set makes implementation difficult
  - Hardware translates instructions to simpler microoperations
    - Simple instructions: 1–1
    - Complex instructions: 1–many
  - Microengine similar to RISC
  - Market share makes this economically viable
- Comparable performance to RISC
  - Compilers avoid complex instructions
- Evolution with backward compatibility
  - If Intel didn't extend with compatibility, its competitors would!
  - Technical elegance ≠ market success

History of the Intel x86 ISA

- 8080 (1974): 8-bit microprocessor
  - Accumulator, plus 3 index-register pairs
- 8086 (1978): 16-bit extension to 8080
  - Complex instruction set (CISC)
- 8087 (1980): floating-point coprocessor
  - Adds FP instructions and register stack
- 80286 (1982): 24-bit addresses, MMU
  - Segmented memory mapping and protection
- 80386 (1985): 32-bit extension (now IA-32)
  - Additional addressing modes and operations
  - Paged memory mapping as well as segments
- i486 (1989): pipelined, on-chip caches and FPU
  - Compatible competitors: AMD, Cyrix, …
- Pentium (1993): superscalar, 64-bit datapath
  - Later versions added MMX (Multi-Media eXtension) instructions
  - The infamous FDIV bug

History of the Intel x86 ISA

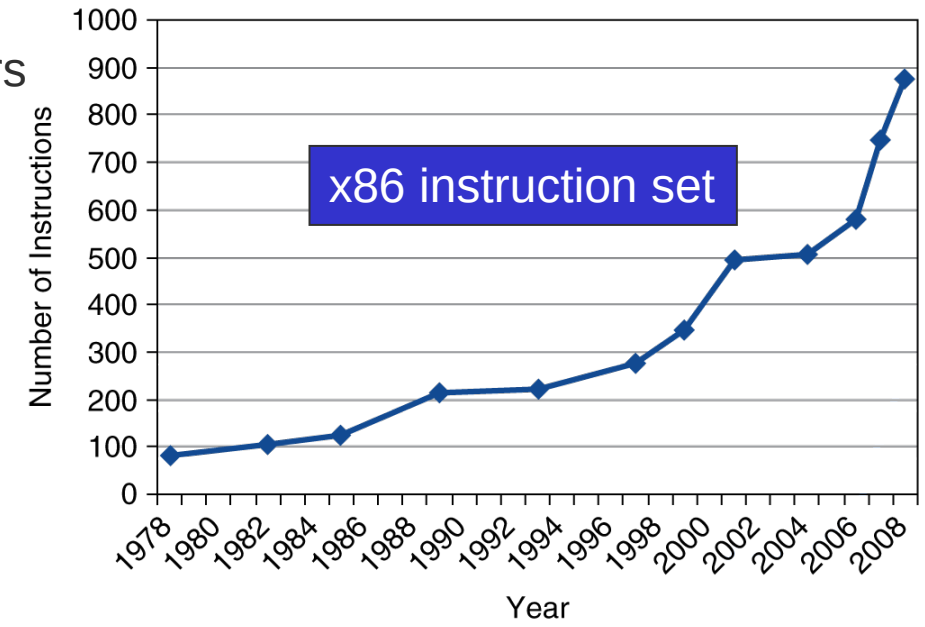
- Pentium Pro (1995), Pentium II (1997)
  - New microarchitecture (see Colwell, *The Pentium Chronicles*)
- Pentium III (1999)
  - Added SSE (Streaming SIMD Extensions) and associated registers
- Pentium 4 (2001)
  - New microarchitecture
  - Added SSE2 instructions
- AMD64 (2003): extended architecture to 64 bits
- EM64T – Extended Memory 64 Technology (2004)
  - AMD64 adopted by Intel (with refinements)
  - Added SSE3 instructions
- Intel Core (2006)
  - Added SSE4 instructions, virtual machine support
- AMD64 (announced 2007): SSE5 instructions
  - Intel declined to follow, instead…
- Advanced Vector Extension (announced 2008)
  - Longer SSE registers, more instructions
- …

Overview

- Instruction set architecture
- The MIPS ISA
- Calling procedures in MIPS
- Dealing with character sets
- Translating and starting a program
- Effect of compiler optimization
- Other ISAs
- Concluding remarks

Fallacies

- Powerful instruction ⇒ higher performance
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions
- Use assembly code for high performance
  - But modern compilers are better at dealing with modern processors
  - More lines of code ⇒ more errors and less productivity
- Backward compatibility ⇒ instruction set doesn't change
  - But they do accrete more instructions



Pitfalls

- Sequential words are not at sequential addresses
  - Increment by 4, not by 1!
- Pointers to local variables (e.g. in procedures)
  - Become invalid when stack popped
  - But address can be passed back via an argument
  - Don’t do that!

Concluding Remarks

- Design principles
  1. Simplicity favors regularity ⇒ fixed instruction formats
  2. Smaller is faster ⇒ only a few registers
  3. Make the common case fast ⇒ PC-relative addressing
  4. Good design demands good compromises ⇒ immediate addressing for small constants help to keep fixed length instruction formats

- Layers of software/hardware
  - Compiler, assembler, hardware
- MIPS: typical of RISC ISAs
- X86: not typical RISC ISA (more complex)