



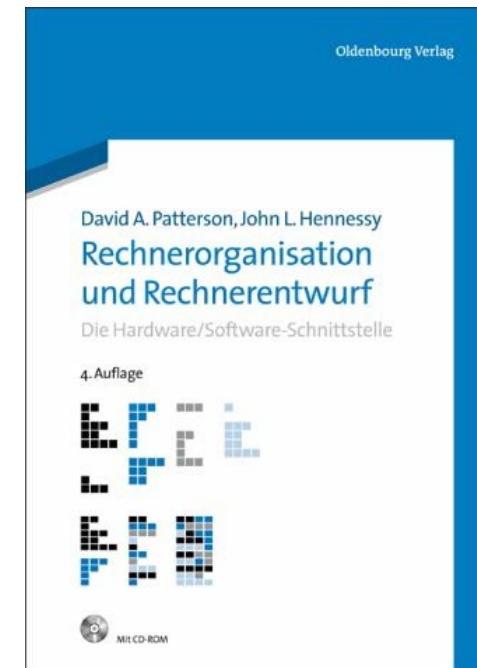
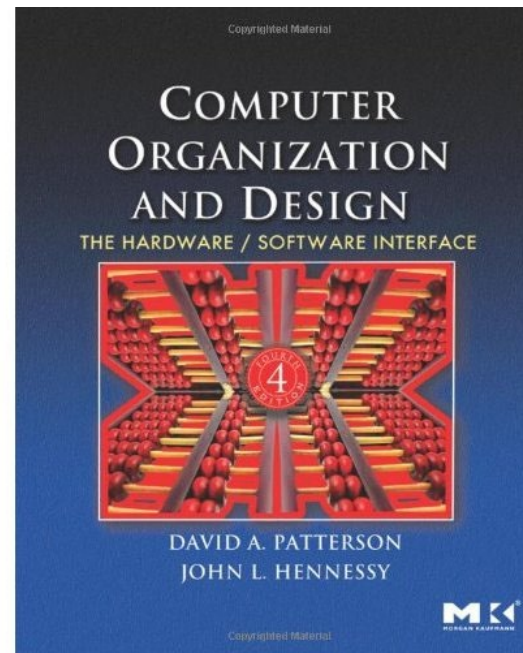
Informatik der Systeme – Chapter 5: Instructions – Language of the Computer

Prof. Dr. Michael Menth

<https://kn.inf.uni-tuebingen.de>



- ▶ 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"



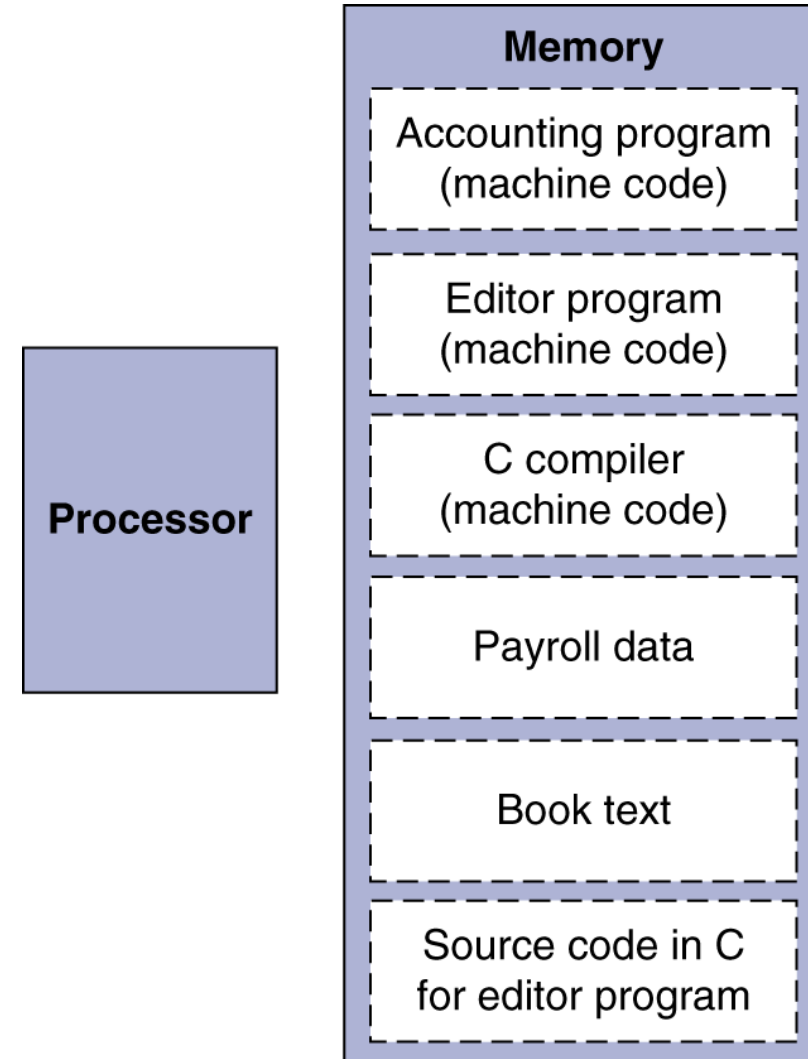


- ▶ Instruction set architecture
- ▶ The MIPS ISA
- ▶ Calling procedures in MIPS
- ▶ Dealing with character sets
- ▶ Translating and starting a program
- ▶ Pointers in C
- ▶ Effect of compiler optimization
- ▶ Other ISAs
- ▶ Concluding remarks



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





- ▶ 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

- 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)
- x86: Internally, CISC instructions are decoded to RISC instructions
- ARM processors in smart phones and tablet computers such as the iPad



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



- ▶ Used as the example throughout the lecture

- ▶ **M**icroprocessor without **I**nterlocked **P**ipeline **S**tages
 - „interlocked“ = ineinander verzahnt, gekuppelt
 - RISC-style ISA, developed since 1981 by John Hennessy at Stanford University
 - From 1984 on: MIPS Computer Systems Inc., today MIPS Technologies



- ▶ Large share of embedded core market (historically)
 - For example: PlayStation 1 and Nintendo 64 gaming consoles

- ▶ Typical of many modern ISAs
 - Many similarities with ARM or RISC-V
 - Scientifically modern and elegant

- ▶ Didactically simple
 - Very good book and simulator!
 - Easy to understand

- ▶ What about x86?
 - Really hard to understand
 - Many (many, many...) legacy parts for backward compatibility
 - Very complex (> 1000 instructions, special purpose extensions)



► 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

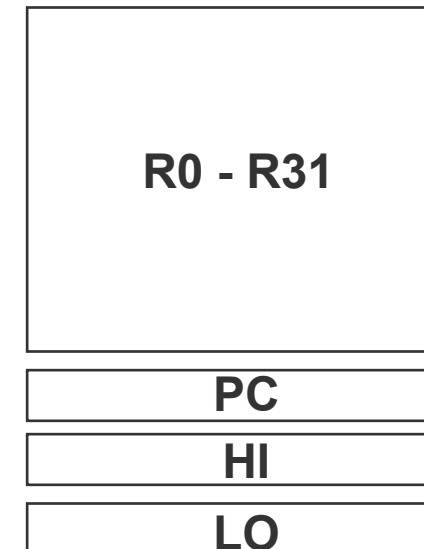


► Instruction categories

- Computational
- Load/store
- Jump and branch
- Floating point
 - Coprocessor
- Memory management
- Special

- Registers \$HI, \$LO used for multiplication and division

Registers (32 bits wide)



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

op	rs	rt	rd	sa	funct
-----------	-----------	-----------	-----------	-----------	--------------

R format

op	rs	rt	immediate
-----------	-----------	-----------	------------------

I format

op	jump target
-----------	--------------------

J format



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	no
\$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

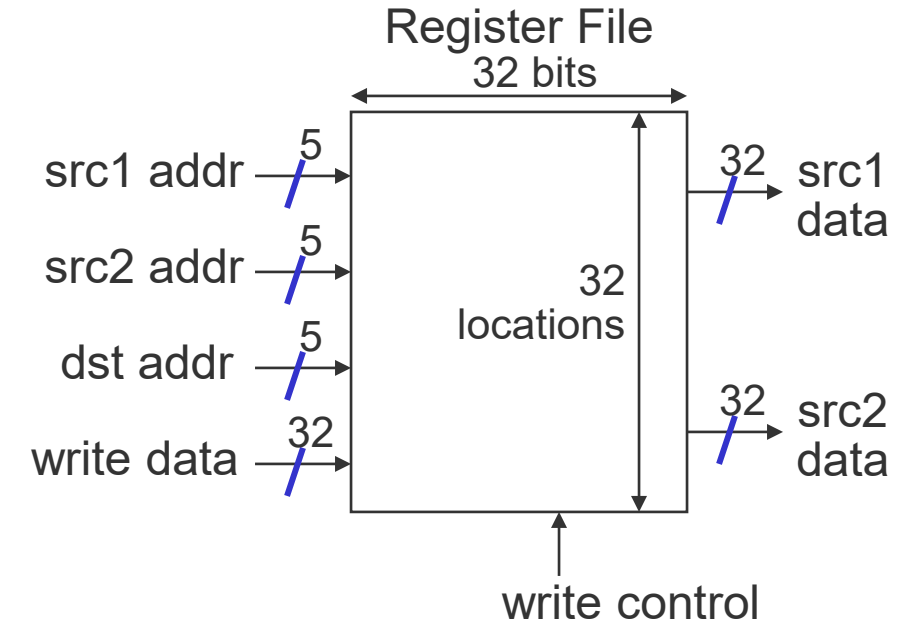


► Holds thirty-two 32-bit registers

- Two read ports and
- One write port

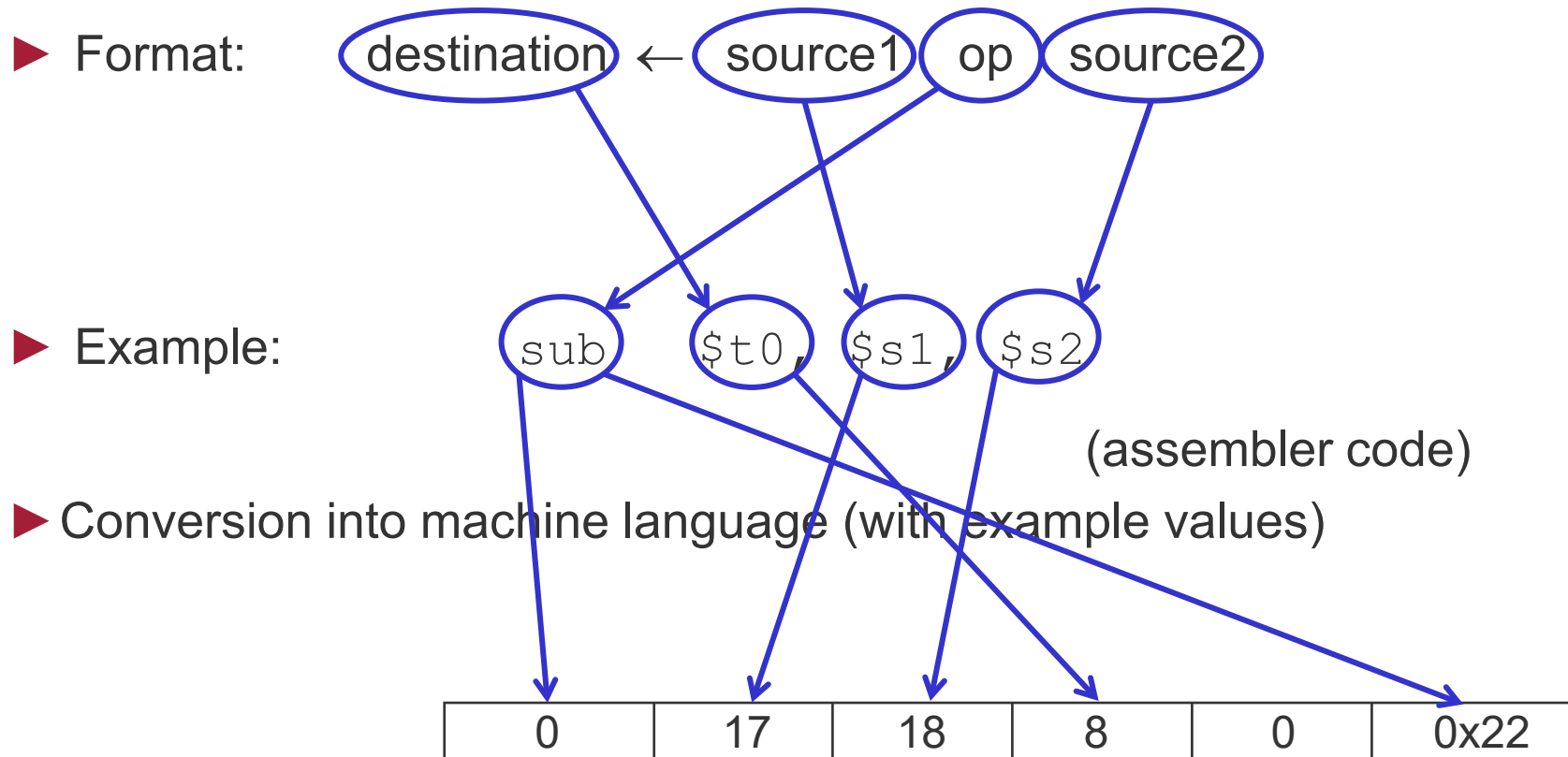
► Registers are

- 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., $(A \cdot B) - (C \cdot D) - (E \cdot F)$ can do multiplies in any order vs. stack
- Can hold variables so that
 - Code density improves (since registers are named with fewer bits than a memory location)





- ▶ 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)



► MIPS instruction formats define special instruction fields

► **R**-Format

- For instructions with up to 3 **R**egisters

op	rs	rt	rd	shamt	funct
----	----	----	----	-------	-------

- 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

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

add \$t0, \$s1, \$s2

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

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



- Motivation: small constants often used in typical code

- Immediate instructions

- Example

`addi $t0, $s2, 4` $\# \$t0 = \$s2 + 4$

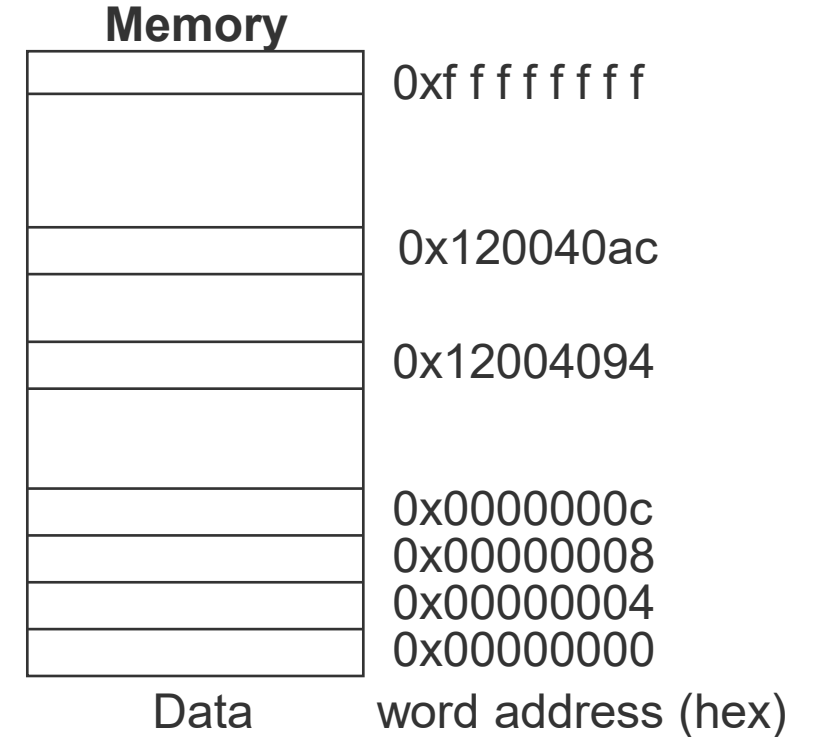
- I-Format



- Constant kept inside the instruction itself
 - Constant values limited to the range from -2^{15} to $+2^{15}-1$
- Other approaches to deal with constants
 - Use hard-wired register \$zero for constant 0 (32 bits wide)
 - Create other 32-bit-wide value in register (see later)



- ▶ Most architectures address individual **bytes** in memory
- ▶ **Alignment restriction**
 - Memory address of a **word** must be on natural word boundaries (a multiple of 4 in MIPS-32)
 - Only whole words can be read/written from memory
- ▶ To apply arithmetic operations
 - Load values from memory into registers
 - Store result from register to memory
- ▶ Used for composite data (variables, arrays, structures, dynamic data, ...)





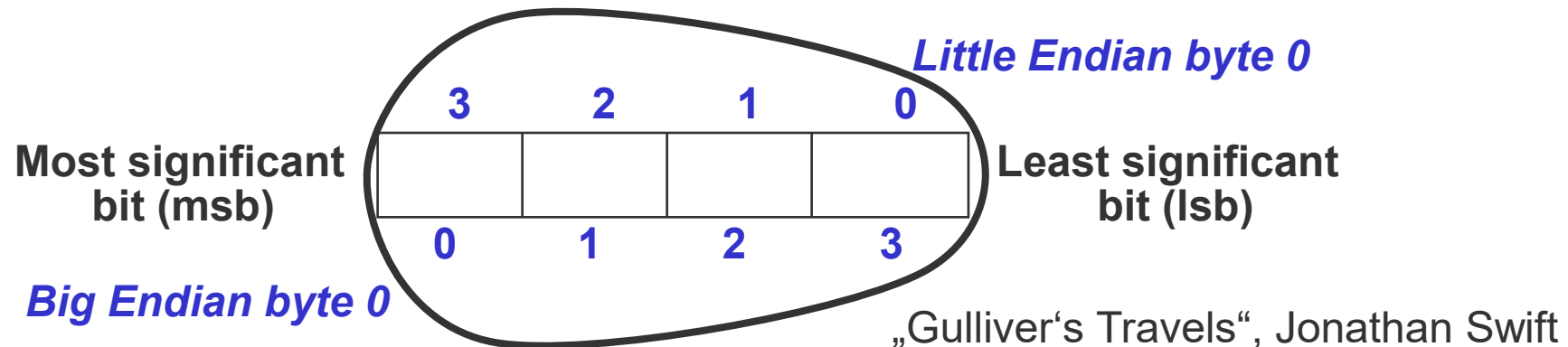
How to number bytes within a word?

► Big Endian

- Leftmost byte is word address
- Examples: IBM 360/370, Motorola 68k, MIPS, Sparc
- Is also „network byte order“ (RFC2360)

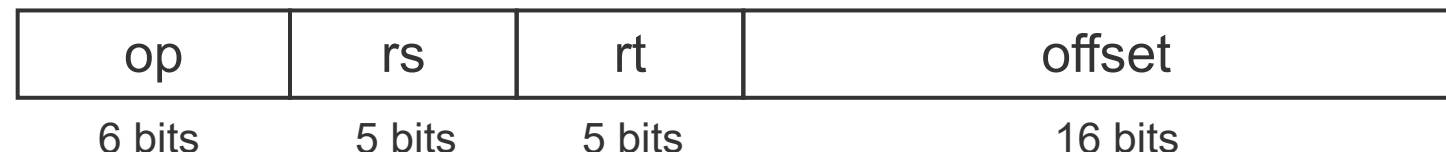
► Little Endian

- Rightmost byte is word address
- Examples: Intel 80x86, DEC Vax, DEC Alpha



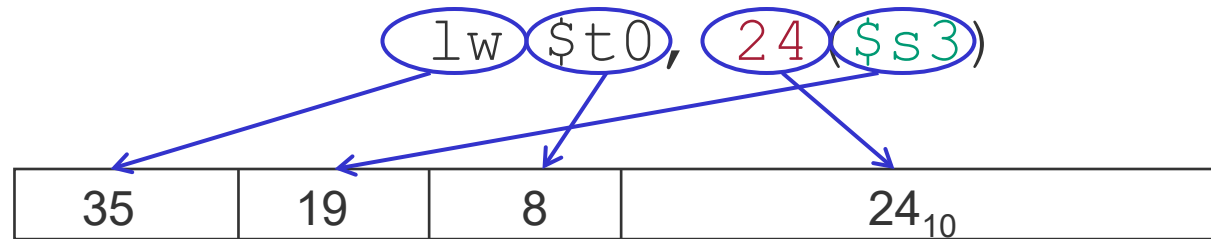


- ▶ MIPS has two basic **data transfer instructions** for accessing memory
 - `lw $t0, 4($s3) #load word from memory`
Data is loaded from memory into register `$t0`
 - `sw $t0, 8($s3) #store word to memory`
Data is stored from register `$t0` into memory
- ▶ The memory address – a 32-bit address – is formed by adding the contents of the **base address register** (`$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
- ▶ **I-Format** – for immediate arithmetic and load/store instructions
 - `rs`: register for base addr; memory addr = base addr + offset
 - `rt`: destination or source register number





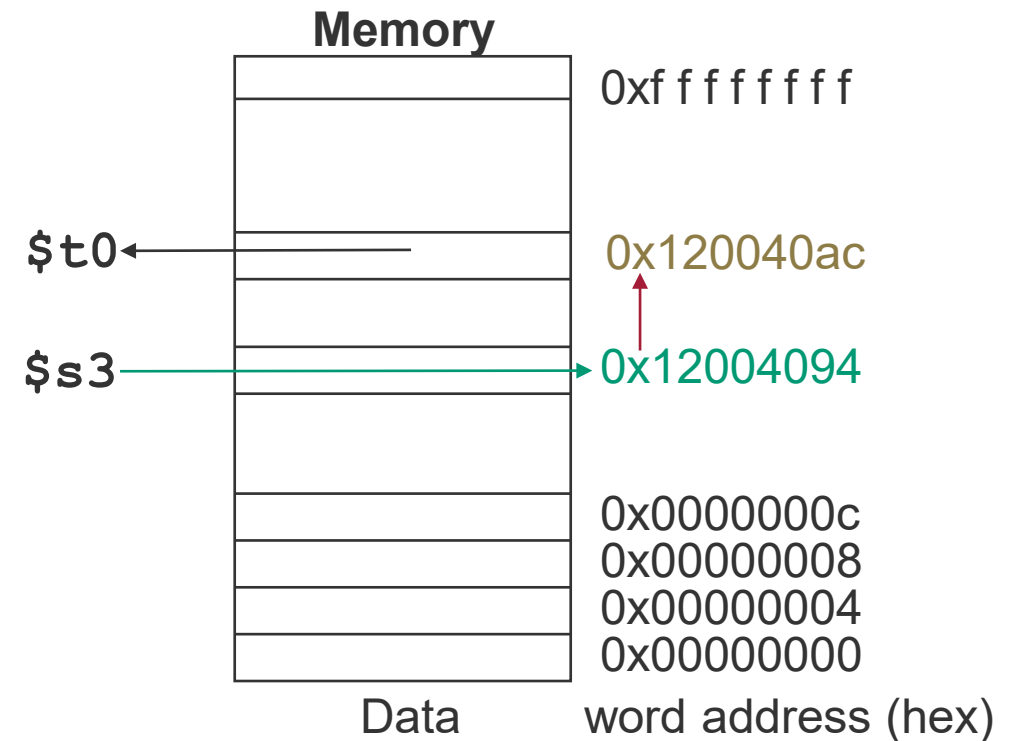
► Load/store instructions use I-Format



$$24_{10} + \$s3 =$$

$$\begin{array}{r}
 \dots 0001\ 1000 \\
 + \dots \underline{1001\ 0100} \\
 \dots 1010\ 1100 = \\
 0x120040ac
 \end{array}$$

Word addresses are multiples of 4!





► MIPS provides special instructions to move **bytes**

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

0x28	19	8	16 bit offset
------	----	---	---------------

► 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?



► Instructions for bitwise word 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
Bitwise AND	&	&	and, andi
Bitwise OR			or, ori
Bitwise NOT	~	~	nor
Shift left	<<	<<	sll
Shift right	>>	>>	srl

- 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
- 5-bit shamt field enough to shift a 32-bit value ($32=2^5$)
 - Between 0 and 31 bit positions



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

and \$t0, \$t1, \$t2

\$t2	0000	0000	0000	0000	00	00	11	01	1100	0000
\$t1	0000	0000	0000	0000	00	11	11	00	0000	0000
\$t0	0000	0000	0000	0000	00	00	11	00	0000	0000



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

or \$t0, \$t1, \$t2

\$t2	0000	0000	0000	0000	00	00	11	01	1100	0000
\$t1	0000	0000	0000	0000	00	11	11	00	0000	0000
\$t0	0000	0000	0000	0000	00	11	11	01	1100	0000



► Useful to invert bits in a word

- Change 0 to 1, and 1 to 0

► MIPS has NOR 3-operand instruction

- $a \text{ NOR } b == \text{NOT} (a \text{ OR } b)$

`nor $t0, $t1, $zero`

Register 0: always
read as 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

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

- Logical operations with two source operands use **R-Format**

0	9	10	8	0	0x24
---	---	----	---	---	------

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 immediate operand use **I-Format**

0x0D	9	8	0xFF00
------	---	---	--------

andi \$t0, \$t1, 0xFF00 #\$t0 = \$t1 & ff00

ori \$t0, \$t1, 0xFF00 #\$t0 = \$t1 | ff00

What does ANDI do
with the 16 most-
significant bits?



- ▶ Two instructions needed
- ▶ 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
------------------	------------------



► MIPS conditional branch instructions

```
bne $s0, $s1, Lbl1    #go to Lbl1 if $s0≠$s1
beq $s0, $s1, Lbl1    #go to Lbl1 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:” marks the corresponding instruction in code
- Lbl1 in the instruction is approximately the offset to the current instruction address (more next slide)

► Conditional branch uses I-Format

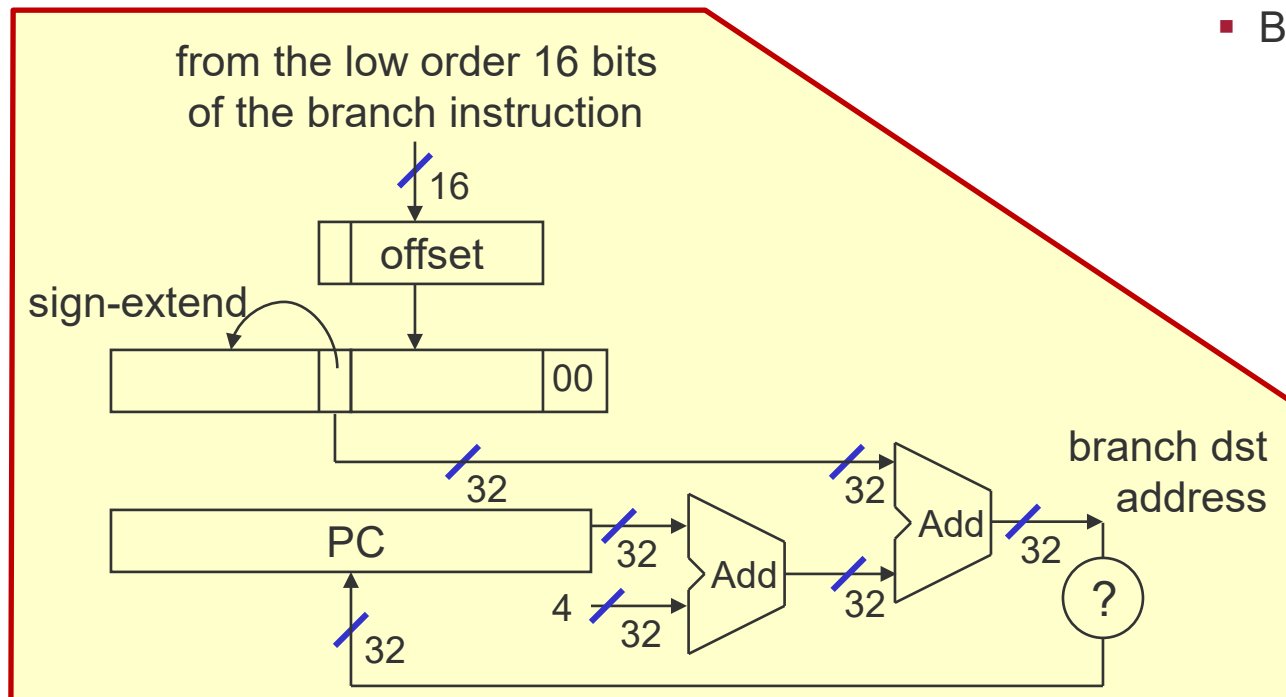
0x05	16	17	16 bit offset
------	----	----	---------------



Specifying Branch Destinations

- ▶ Example: `bne $s0, $s1, Lbl1`
- ▶ Special register: program counter PC
 - Holds memory address of next instruction to be executed
 - Usually PC is automatically incremented during “fetch cycle”: $PC = PC + 4$

- ▶ Branch
 - Requires $PC = PC + 4 + 4 * (\text{value of Lbl1})$, i.e., the label denotes an offset within the instruction flow
 - 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



► More complex branch instructions

- `blt`, `bge`, ...
- Hardware for `<`, `≥`, ... slower than for `=`, `≠`
- Combining with branch involves more work per instruction, requiring a slower clock – all instructions penalized!
- Solution: use 2 instructions instead, `slt` required

► 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 ...
```



► Emulate other conditional branches using

- `slt`
- `beq` or `bne`, and
- registers `$zero` and `$at`

► Examples

- 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`)



► 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`
 - $-1 < +1 \Rightarrow \$t0 = 1$
- `sltu $t0, $s0, $s1 # unsigned`
 - $+4,294,967,295 > +1 \Rightarrow \$t0 = 0$

► Low-cost checking for “index out of bounds” for arrays

- Array with n elements: $0 \leq i < n$ (i : index)

- MIPS code

```

i in $s1, n in $s2
sltu $t0, $s1, $s2           # $t0 = 1 if
                              # $s1 < $s2 (max)
                              # and $s1 >= 0 (min)

beq $t0, $zero, IOOB         # go to IOOB if
                              # $t0 = 0

```

► First instruction checks both conditions

- Negative integers in two’s complement look like large numbers in unsigned notation.
- Therefore, an unsigned comparison of $i < n$ checks if $i < n$ and if $i \geq 0$.



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

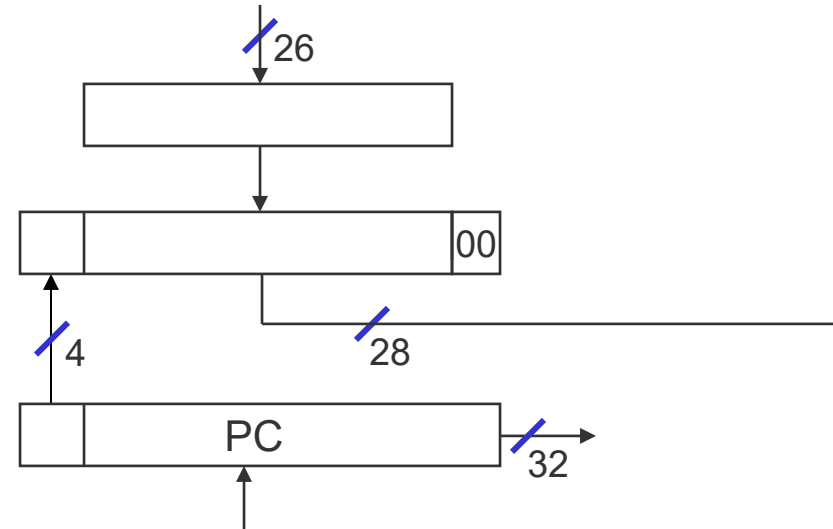
```
j    label           # go to label
```

- Jump instruction uses the **J-Format**



from the low order 26 bits of the jump instruction

- Implementation and exact semantics



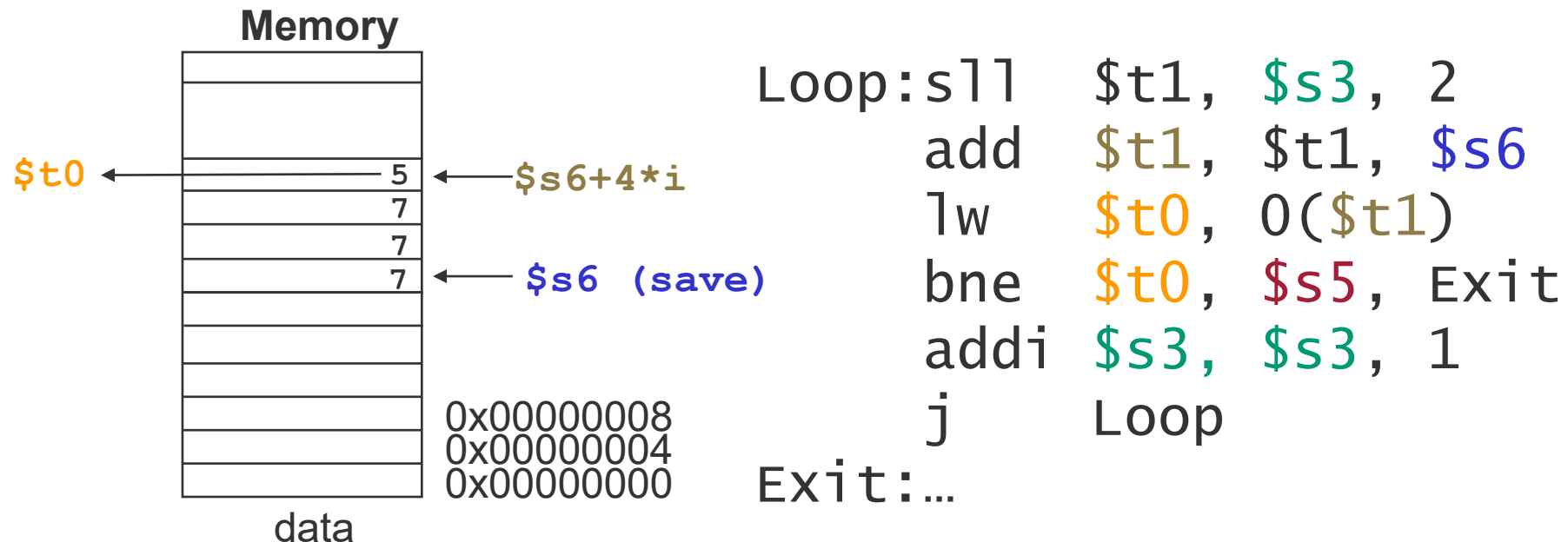


► C code:

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

► Compiled MIPS code

- address of save in $\$s6$, $k=7$ in $\$s5$, i in $\$s3$





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:    ...

```



► Register operands

- Direct access
- Used by arithmetic instructions
- Only 32 registers to enable fast access in implementations

► 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

► Memory operands

- Only accessible via load and store instructions
 - ⇒ Compilers should use registers for variables as much as possible
- Only spill to memory for less frequently used variables
- Register optimization is important!



Summary of Addressing Modes in MIPS

► Immediate addressing

- `addi $s0 $s1 1`

► Register addressing

- `add $s0 $s1 $s2`

► Base addressing

- `sw $s0 4($s3)`

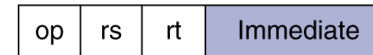
► PC-relative addressing

- `bne $s0 $s1 loop`

► Pseudo-direct addressing

- `j loop`

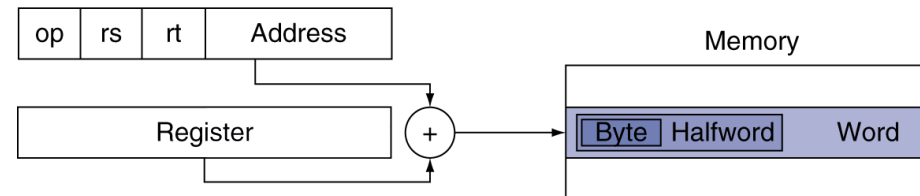
1. Immediate addressing



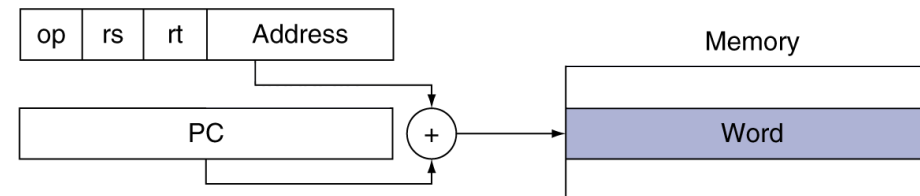
2. Register addressing



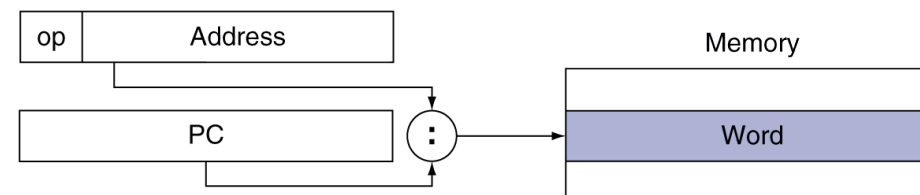
3. Base addressing



4. PC-relative addressing



5. Pseudodirect addressing





► Must support the arithmetic/logic operations of the ISA

add, addi, addiu, addu

sub, subu

mult, multu

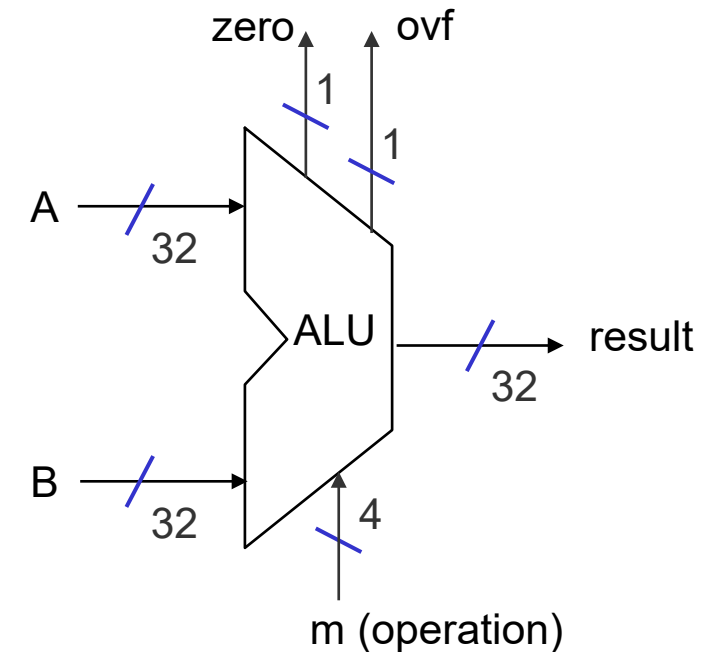
div, divu

and, andi, nor, or, ori, xor, xori

beq, bne, slt, slti, sltiu, sltu

► With special handling for

- Sign extension – addi, addiu, slti, sltiu
- Zero extension – andi, ori, xori
- Overflow detection – add, addi, sub
- No overflow handling applied for unsigned operations





- ▶ 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

► Consider 2's complement numbers with 4 digits $\Rightarrow$ values $[-8; 7]$

- Addition works like for binary numbers, but disregard last carry-over
https://de.wikipedia.org/wiki/Zweierkomplement#Addition_und_Subtraktion

► $A > 0, B > 0$

- $A + B > 0$

$$\begin{array}{rcl}
 0100 & = & 4 \\
 + 0011 & = & 3 \\
 \hline
 0111 & = & 7
 \end{array}$$

- $A + B < 0$

$$\begin{array}{rcl}
 0100 & = & 4 \\
 + 0100 & = & 4 \\
 \hline
 1000 & = & -8 \neq 8
 \end{array}$$

► $A < 0, B < 0$

- $A + B < 0$

$$\begin{array}{rcl}
 1100 & = & -4 \\
 + 1100 & = & -4 \\
 \hline
 11000 & = & -8
 \end{array}$$

- $A + B > 0$

$$\begin{array}{rcl}
 1100 & = & -4 \\
 + 1011 & = & -5 \\
 \hline
 10111 & = & +7 \neq -9
 \end{array}$$



Support for Floating Point Numbers

- ▶ Requires floating point unit (FPU)
 - Co-processor
 - Chip separated from CPU for specialized purposes
 - Other uses for co-processors
 - Graphics acceleration
 - AI acceleration (very recent)
 - Today: FPU integrated on CPU
 - Cheaper than co-processor
 - Works w/ modern caches
- ▶ MIPS instructions support features from up to 4 co-processors
 - 1 for memory management
 - Exception and trap handling (≠ memory management unit)
 - 1 FPU
 - 2 optionals co-processors for for special uses

- ▶ FPU in MIPS
 - Lecture assumes co-processor
 - Modern MIPS processors have CPU-integrated FPU
 - Either case: floating point operations cannot use general purpose registers
- ▶ 32 special registers $\$f0-\$f31$ for floating point operations
 - Used only for floating point numbers
 - Every register contains a 32 bit single precision number
 - Numbers are stored in compliance with IEEE 754
 - Two registers ($\$f0-\$f1$, $\$f2-\$f3$, ...) can be used combined for double precision 64 bit numbers



Instructions for Floating Point Numbers

► Loading from address

- `l.s fd, offset(rbase)`
- `".s"` stands for single precision

► Storing to address

- `s.s fs, offset(rbase)`

► Arithmetic operations

- `add/sub.s fd, fs, ft`
- `mul/div.s fd, fs, ft`
- `sqrt.s fd, fs`

► Support for double precision 64 bit numbers

- Instructions with suffix `".d"`
- Two registers per number
- Only even register numbers can be used

► Move float register content

- `mov.s fd, fs`

► Move from general register to float register

- `mtc1 rs, fd`
- "move to co-processor 1"
- Source and destination position are reversed!

► Move to general register from float register

- `mfc1 rd, fs`
- "move from co-processor 1"

► Only bit patterns moved w/o conversion or type checking

- Exceptions raised by using arithmetic w/ invalid register content or invalid operation
 - IEEE 754 describes what has to be done



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



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

► 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**



► Procedure **return** with

```
jr     $ra                #return
```

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





► 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 “grows” from high address to low address

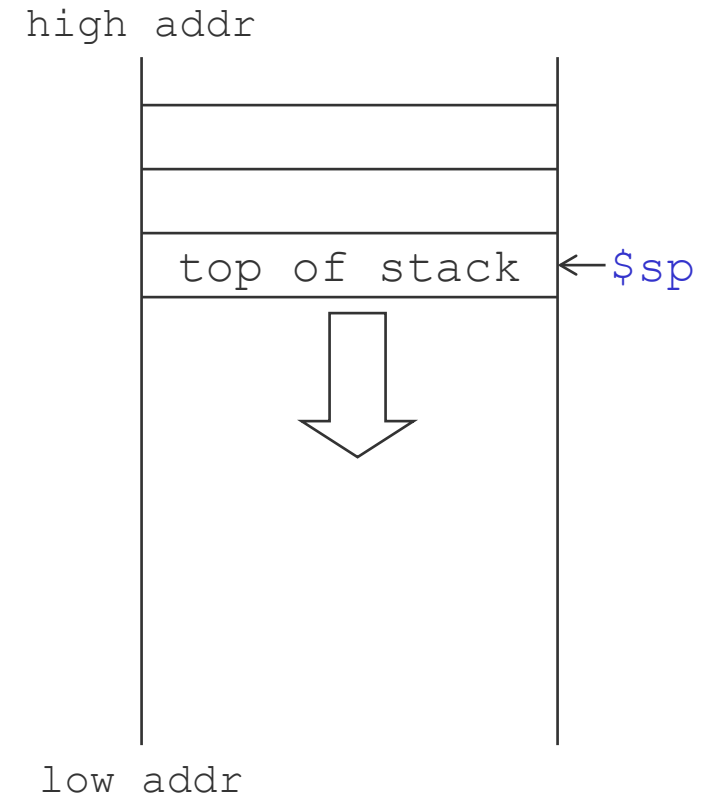
► **Stack pointer** $\$sp$ ($\$29$)

- One of the general registers
- Used to address the stack
 - Points to last word used
- 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 $\$sp$

$$\$sp = \$sp + 4$$





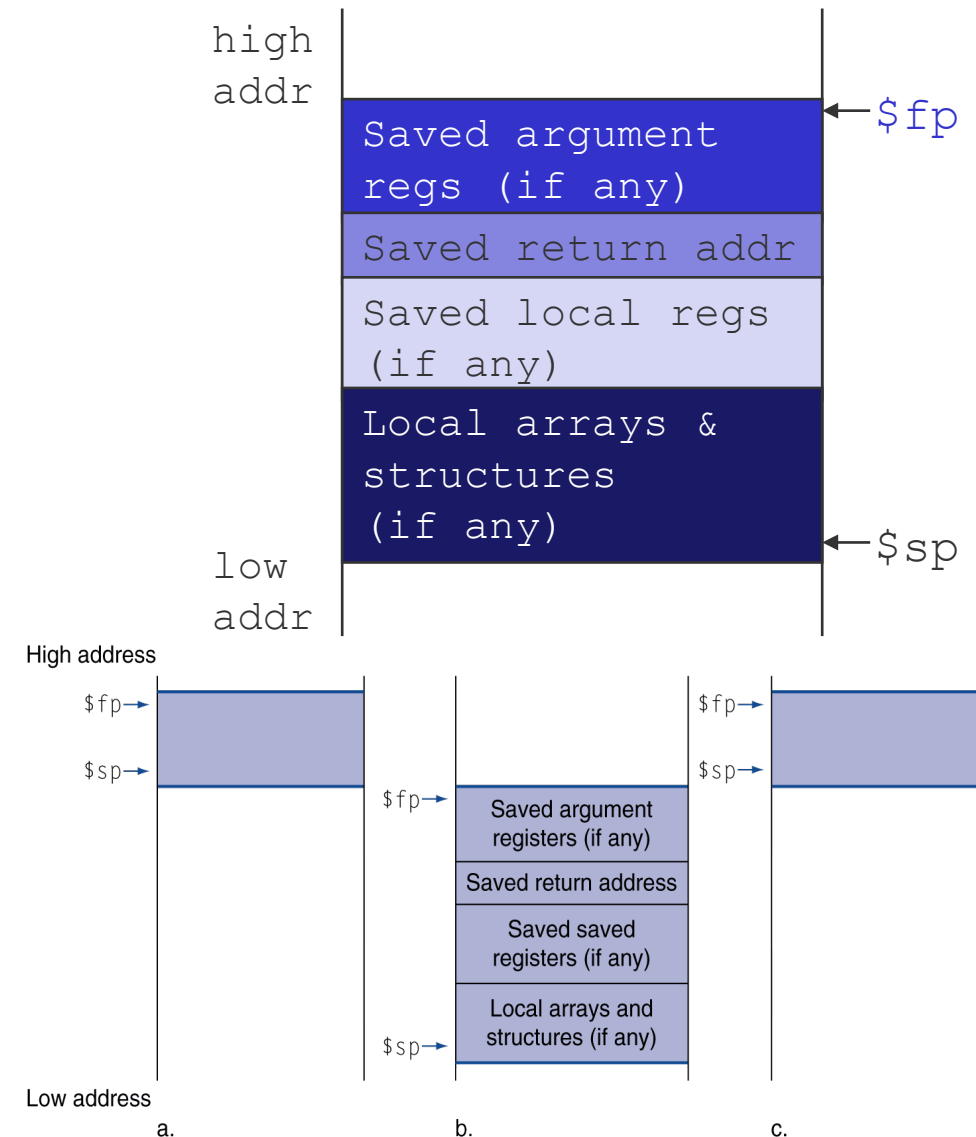
Aside: Allocating Space on the Stack

► 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 procedure frame
- Provides a stable “base” register for the procedure
 - \$fp may be initialized using \$sp on a call and
 - \$sp may be restored using \$fp on a return
- Right figure: before, during, and after procedure call





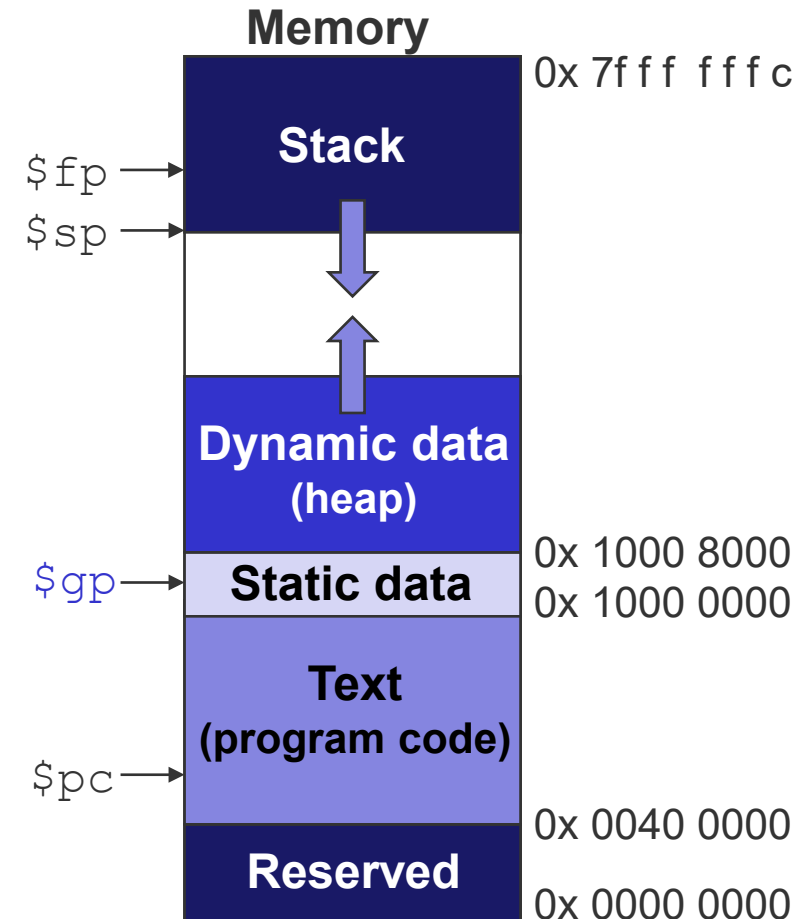
Aside: Allocating Space on the Heap

► 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()`





► Calling program

- Saves any arguments $\$ax$ and temporaries $\$tx$ needed after the call
- Must restore them after return from procedure

► Called 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)
- Initially also saves its return address on the stack



► 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
```



► 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   # if so, result is 1
    addi $v0, $zero, 1    #   pop 2 items from stack
    addi $sp, $sp, 8      #   and return
    jr   $ra
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
```



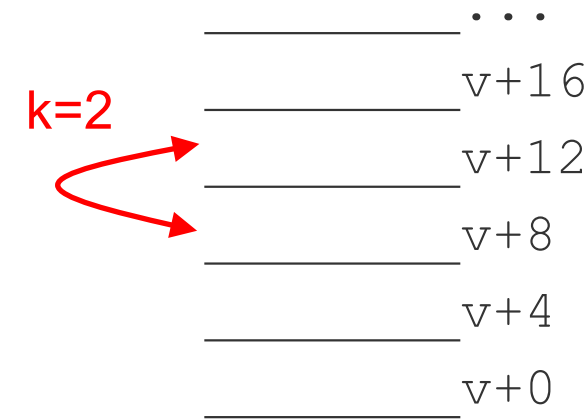
Runtime Example (Requires Animation)

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



- Illustrates use of assembly instructions for a C (insertion) 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;
}
```



► 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($\$t1$)	# $\$t0 = v[k]$
lw $\$t2$, 4($\$t1$)	# $\$t2 = v[k+1]$
sw $\$t2$, 0($\$t1$)	# $v[k] = \$t2$ ($v[k+1]$)
sw $\$t0$, 4($\$t1$)	# $v[k+1] = \$t0$
jr $\$ra$	# return to calling routine



► 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.



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

- ▶ $i=1$
 - $j=0$: $v=[3,4,2,1]$

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

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



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



► 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.



Example: Procedure Body of “sort” in MIPS

► v in $\$a0$, n in $\$a1$, i in $\$s0$, j in $\$s1$, deviations from the book in red.

	move $\$s2, \$a0$	# save $\$a0$ into $\$s2$
	move $\$s3, \$a1$	# save $\$a1$ into $\$s3$
for1tst:	addi $\$s0, \$zero, 1$	# $i = 1$
	slt $\$t0, \$s0, \$s3$	# $\$t0 = 1$ if $\$s0 < \$s3$ ($i < n$)
	beq $\$t0, \$zero, exit1$	# go to exit1 if $\$s0 \geq \$s3$ ($i \geq n$)
for2tst:	addi $\$s1, \$s0, -1$	# $j = i - 1$
	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 $\$t4 \geq \$t3$
	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



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



► Byte-encoded character sets

■ ASCII

- American Standard Code for Information Interchange
- 128 characters: 95 graphic, 33 control
- Uses only 7 bits

■ ISO/IEC 8859 standard

- Provides 15 character sets to support European languages plus Turkish, Arabic, Hebrew, and Thai
- Extension of ASCII to 8 bits
- Latin-1 (aka ISO 8859-1): 256 characters, ASCII, +96 more graphic characters, specialized for Western European languages, now better use UTF-8

■ ANSI

- Other extension of ASCII to 8 bits
- Similar to Latin-1

► EBCDIC (Extended Binary Coded Decimal Interchange Code)

- Mostly used on mainframes



Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	NUL (null)	32	SPACE	64	@	96	`
1	SOH (start of heading)	33	!	65	A	97	a
2	STX (start of text)	34	"	66	B	98	b
3	ETX (end of text)	35	#	67	C	99	c
4	EOT (end of transmission)	36	\$	68	D	100	d
5	ENQ (enquiry)	37	%	69	E	101	e
6	ACK (acknowledge)	38	&	70	F	102	f
7	BEL (bell)	39	'	71	G	103	g
8	BS (backspace)	40	(	72	H	104	h
9	TAB (horizontal tab)	41	)	73	I	105	i
10	LF (NL line feed, new line)	42	*	74	J	106	j
11	VT (vertical tab)	43	+	75	K	107	k
12	FF (NP form feed, new page)	44	,	76	L	108	l
13	CR (carriage return)	45	-	77	M	109	m
14	SO (shift out)	46	.	78	N	110	n
15	SI (shift in)	47	/	79	O	111	o
16	DLE (data link escape)	48	0	80	P	112	p
17	DC1 (device control 1)	49	1	81	Q	113	q
18	DC2 (device control 2)	50	2	82	R	114	r
19	DC3 (device control 3)	51	3	83	S	115	s
20	DC4 (device control 4)	52	4	84	T	116	t
21	NAK (negative acknowledge)	53	5	85	U	117	u
22	SYN (synchronous idle)	54	6	86	V	118	v
23	ETB (end of trans. block)	55	7	87	W	119	w
24	CAN (cancel)	56	8	88	X	120	x
25	EM (end of medium)	57	9	89	Y	121	y
26	SUB (substitute)	58	:	90	Z	122	z
27	ESC (escape)	59	;	91	[	123	{
28	FS (file separator)	60	<	92	\	124	
29	GS (group separator)	61	=	93	]	125	}
30	RS (record separator)	62	>	94	^	126	~
31	US (unit separator)	63	?	95	_	127	DEL

Source:

<https://www.cs.cmu.edu/~pattis/15-1XX/common/handouts/ascii.html>



► Unicode

- Most of the world's alphabets, plus symbols
- 17x16-bit character sets (1114112 codepoints, not all defined)
 - 1st plane: U+0000 ... U+FFFF (Basic Multilingual Plane, BMP)
 - 2nd plane: U+10000 ... U+1FFFF
 - ...
 - 17th plane: U+100000 ... U+10FFFF
- ISO name for Unicode: Universal (Coded) Character Set (UCS)

► UTF (Unicode Transformation Format or USC Transformation Format)

- Variable-length encoding
- Represent all Unicode characters
- Several variants: UTF-8, UTF-16, UTF-32, UTF-EBCDIC, ...

► UTF-8/16/32

- Code unit size 8/16/32 bit
- Some codepoints require several code units for coding (1-4 bytes / 2 or 4 bytes, 4 bytes)
- UTF-8 equals ASCII in first 128 characters



► UTF-8

- GNU/Linux, Unix
- E-Mail, WWW, other Internet-related services
- Used by 88.9% of all websites in April 2017 (<https://de.wikipedia.org/wiki/UTF-8>)

► UTF-16

- Java, .NET
 - `String.length()` in Java returns the number of code units in the `String` $\neq$ number of characters
 - `java.lang.String.codePointCount(): public int codePointCount(int beginIndex, int endIndex)` yields number of codepoints in `[beginIndex, endIndex)`
- Windows, OS X

► Punycode

- RFC 3492, encodes internationalized domain names containing non-ASCII characters with ASCII characters, <https://en.wikipedia.org/wiki/Punycode>
- Example: "München" encoded as "Mnchen-3ya"

► Character sets matter for LaTeX

- <http://www.schlosser.info/seltsame-umlaute-in-latex-zeichenkodierung-pruefen/>



- ▶ 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
 - „Hello World!\0“
 - 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



► 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)`



► 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')
}
```



► 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

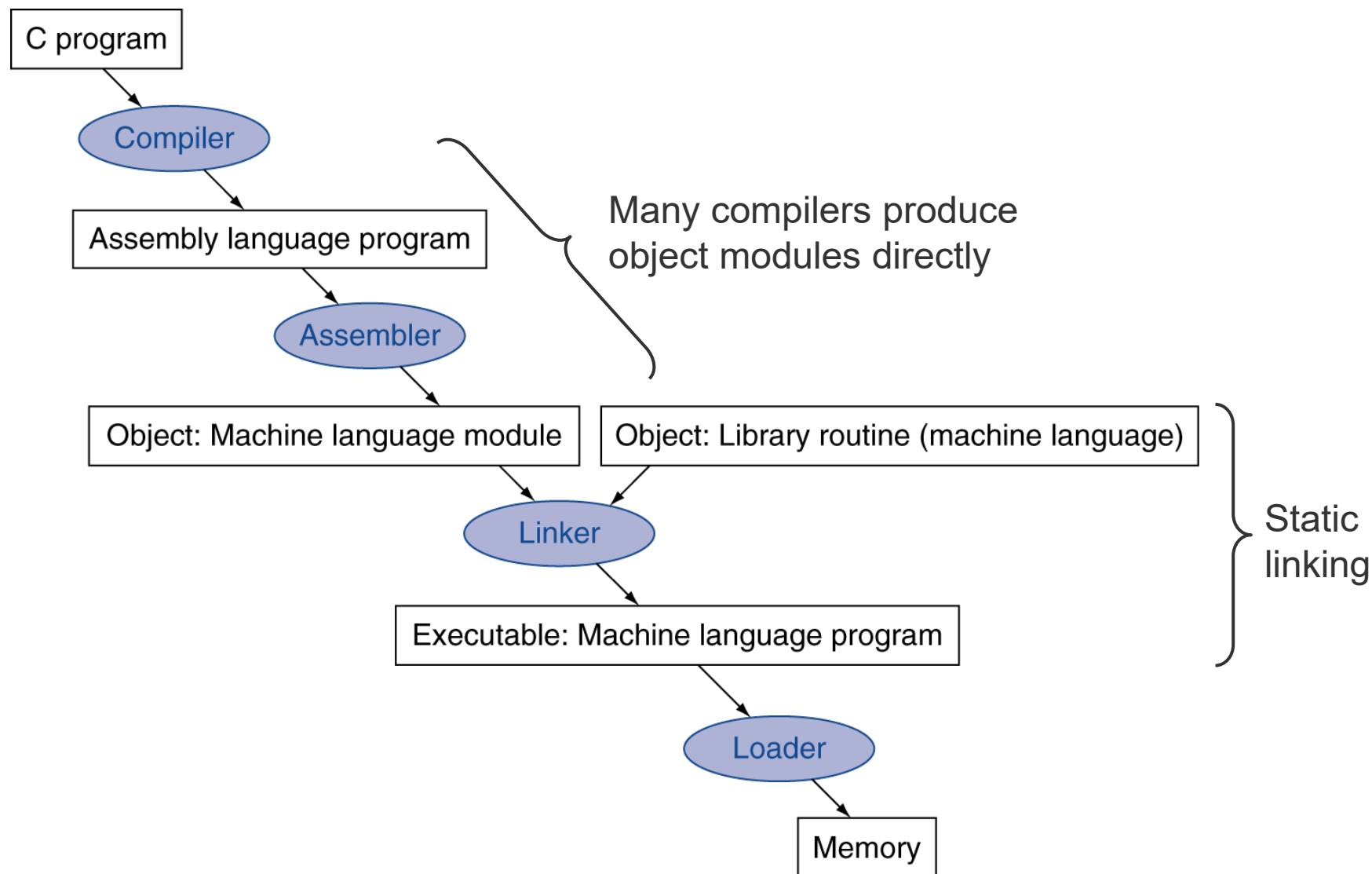
# start of do ... while statement
do: addi  $s0, $s0, 1      # i = i + 1 (bytes, not words!)
    add   $t1, $s0, $a1    # addr of y[i] in $t1
    lbu   $t2, 0($t1)      # $t2 = y[i]
    add   $t3, $s0, $a0    # addr of x[i] in $t3
    sb    $t2, 0($t3)      # x[i] = y[i]
    bne   $t2, $zero, do   # 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

```



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





► 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



► Assembler (or compiler)

- Translates program into object modules including machine instructions
- Provides information for building a complete program from the pieces

► Object module

- Header
 - Describes 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



► 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)

- Merges segments
- Resolve labels (determine their addresses)
- Patch location-dependent and external labels/refs
- Produces an executable image

► 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



► 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”



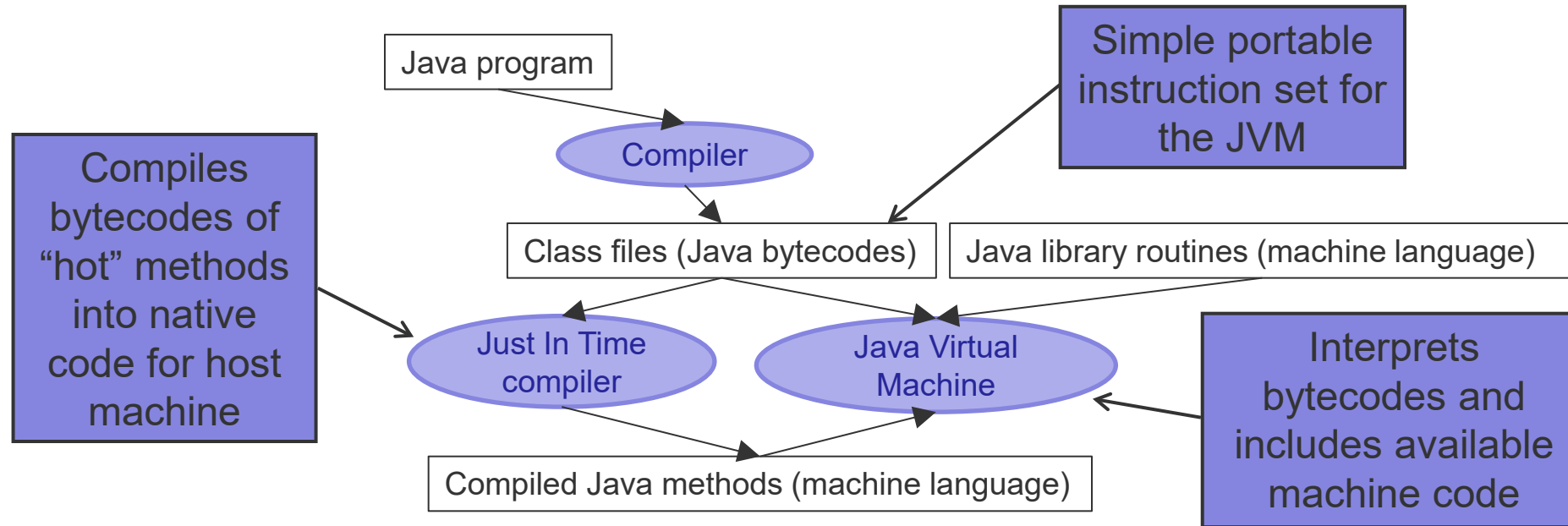
- ▶ Only link/load library procedure when it is called
 - Requires procedure code to be relocatable
 - Avoids image bloat caused by static linking of all (transitively) referenced libraries
 - Automatically picks up new library versions
 - Object module of such a library: dynamically linked library (DLL)

- ▶ 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 interprets Java bytecode
- ▶ Slower than machine code
 - Today for most applications not important
- ▶ 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



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

- ▶ `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 an 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)`

...	...
<u>a [4] = 4</u>	a + 16
<u>a [3] = 5</u>	a + 12
<u>a [2] = 7</u>	a + 8
<u>a [1] = 2</u>	a + 4
<u>a [0] = 3</u>	a + 0



- ▶ 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 this is the job of the compiler
 - Optimization method: “elimination of induction variables” converts usage of array indexing to usage of pointers
 - Recommendation
 - Write more readable code using array indexing
 - Compilers are much better in optimizing on machine level



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

<pre>void clear1(int array[], int size) { int i; for (i = 0; i < size; i += 1) array[i] = 0; }</pre>	<pre>void clear2(int *array, int size) { int *p; for (p = &array[0]; p < &array[size]; p = p + 1) *p = 0; }</pre>
<pre> 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</pre>	<pre> 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</pre>

MIPS code uses „do while“ instead of „for“ loop, makes a difference for size=0



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



- ▶ 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



```
int sum = 0;
for (i = 1; i <= 10; i++) {
    sum = sum + i;
}
```



```
sum = 55;
```

- ▶ Substitute simple computations with result at compile time

```
sum = sum * 4;
```



```
sum = sum << 2;
```

- ▶ Substitute hard operations with easy ones

```
for (i = 1; i <= 10; i++) {
    sum = sum + i;
    sum2 = sum2 + (2+i)*3;
}
```

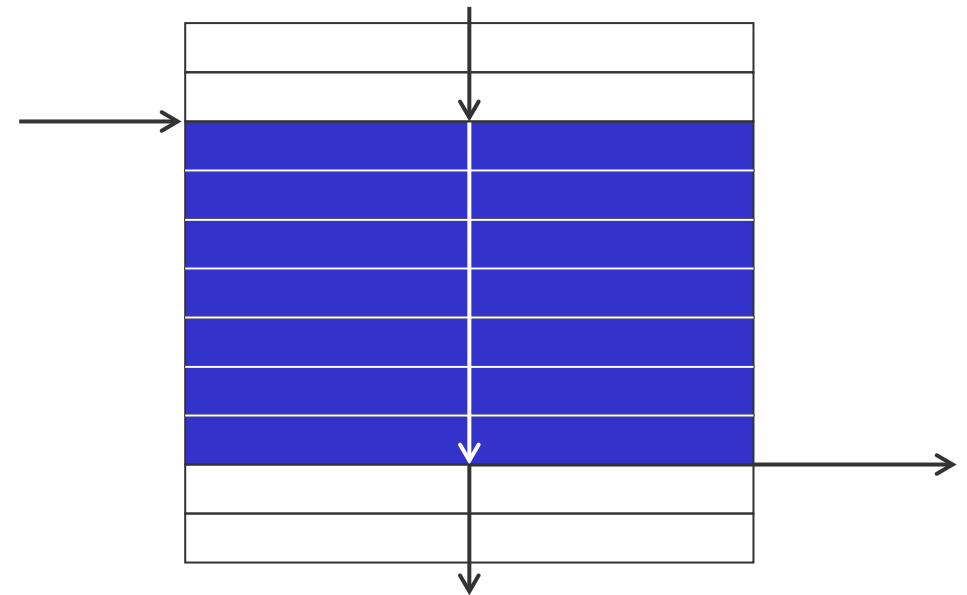


```
for (i = 1; i <= 10; i++) {
    sum = sum + i;
}
for (i = 1; i <= 10; i++) {
    sum2 = sum2 + (2+i)*3;
}
```

- ▶ Loop distribution
 - We come back to this when we learn about caching
 - Also useful if register number too small



- ▶ 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





```

for (i = 0; i < 20; i++) {
    sum = sum + power(i, 2);
}
    
```



```

for (i = 0; i < 5; i++) {
    sum = sum + power((i*4), 2);
    sum = sum + power((i*4)+1, 2);
    sum = sum + power((i*4)+2, 2);
    sum = sum + power((i*4)+3, 2);
}
    
```

- ▶ Many optimization techniques have drawbacks
- ▶ Substitute loop body, such that multiple iterations are performed at once
 - Saves jump operations
 - We will see why this is useful when learning about pipelining
 - Increases size of machine code
 - Bad on embedded platforms
 - More „identical“ instructions to store in L1 Cache
 - We come back to this when we learn about caching



Hands-on on compiler optimization

- ▶ GCC = Gnu Compiler Collection
 - Available for all platforms
 - Open source (GPL)
- ▶ Supports four coarse-grained optimization levels
 - O0, O1, O2 and O3
 - Also supports fine-grained on/off switching of optimization techniques
- ▶ Also some special levels
 - Os for minimal code size
 - Og for minimal compilation time and debugging

- ▶ O0
 - No optimization, only for debugging
- ▶ O1
 - Produces smaller **and** faster code
- ▶ O2
 - Some more features than O1, but no compromise on size, recommended
- ▶ O3
 - Even more features active
 - Expensive in compilation time and code size
 - Often worse than O2
 - Higher number $\neq$ better

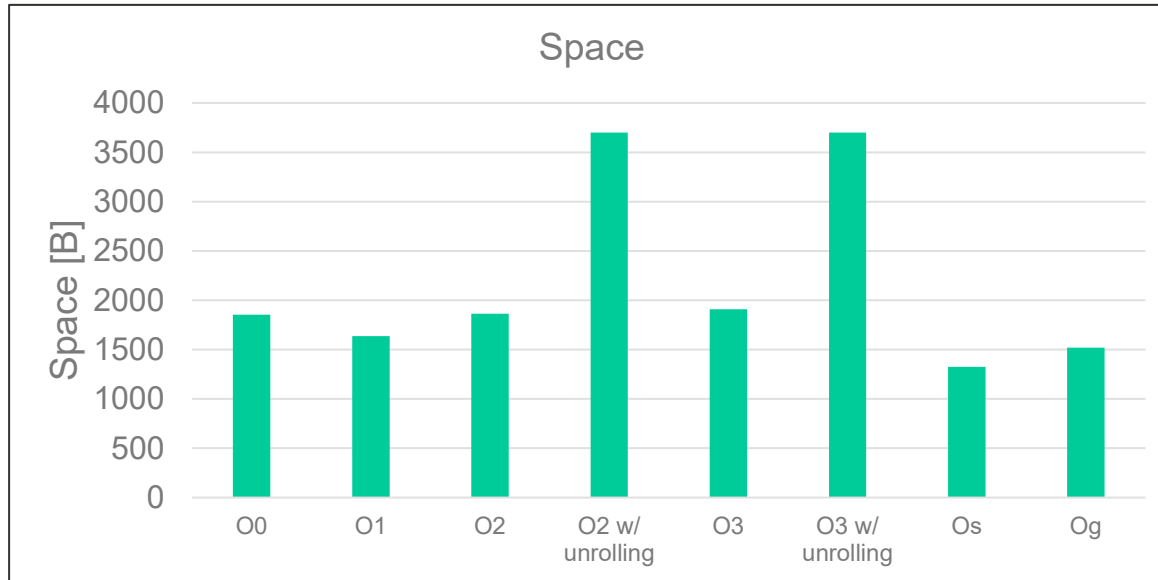


```

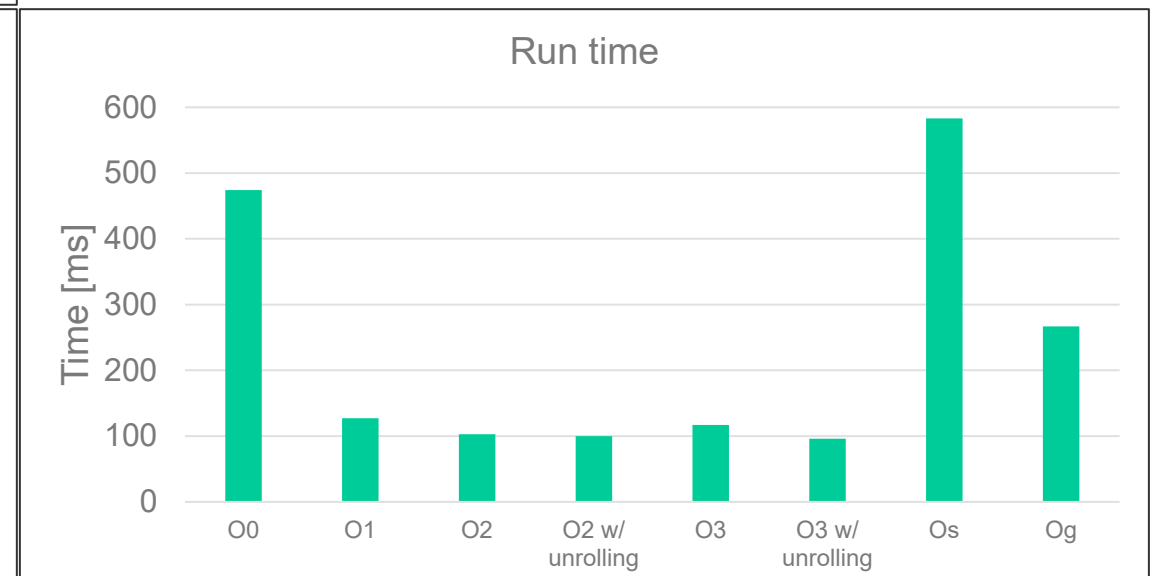
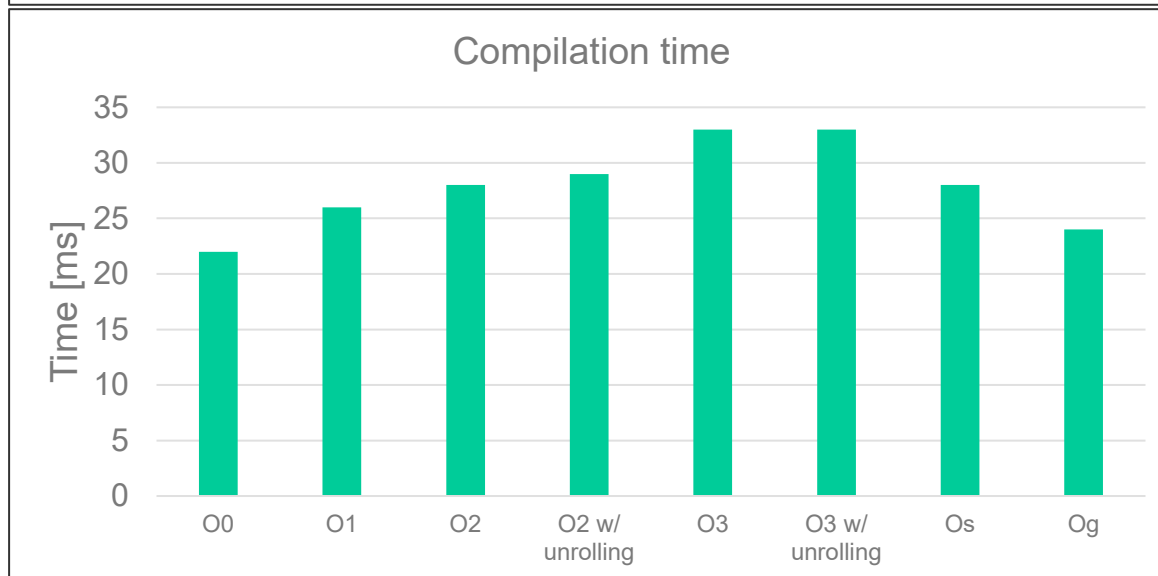
1  #include <stdio.h>
2
3  double power (double b, int e) {
4      double x = 1.0;
5      int j;
6
7      for (j = 1; j <= e; j++) {
8          x = b * x;
9      }
10
11     return x;
12 }
13
14 int main (void) {
15     double sum = 0;
16     int i;
17
18     for (i = 1; i <= 1000000000; i++) {
19         sum = sum + power(i, i % 5);
20     }
21
22     printf("sum = %f\n", sum);
23     return 0;
24 }

```

- ▶ **power** is a candidate for inlining
- ▶ Computations with floating point numbers
 - Special instructions could be used (AVX, MMX, SSE, ...)
- ▶ Loops could be unrolled to save jumps
- ▶ Loop could also be substituted
 - Detecting this is expensive and in general, there is no algorithm which can do this!



- ▶ O3 slightly worse than O2 in every aspect
- ▶ Os produces much smaller code
- ▶ Og is compromise of debugging and small optimizations



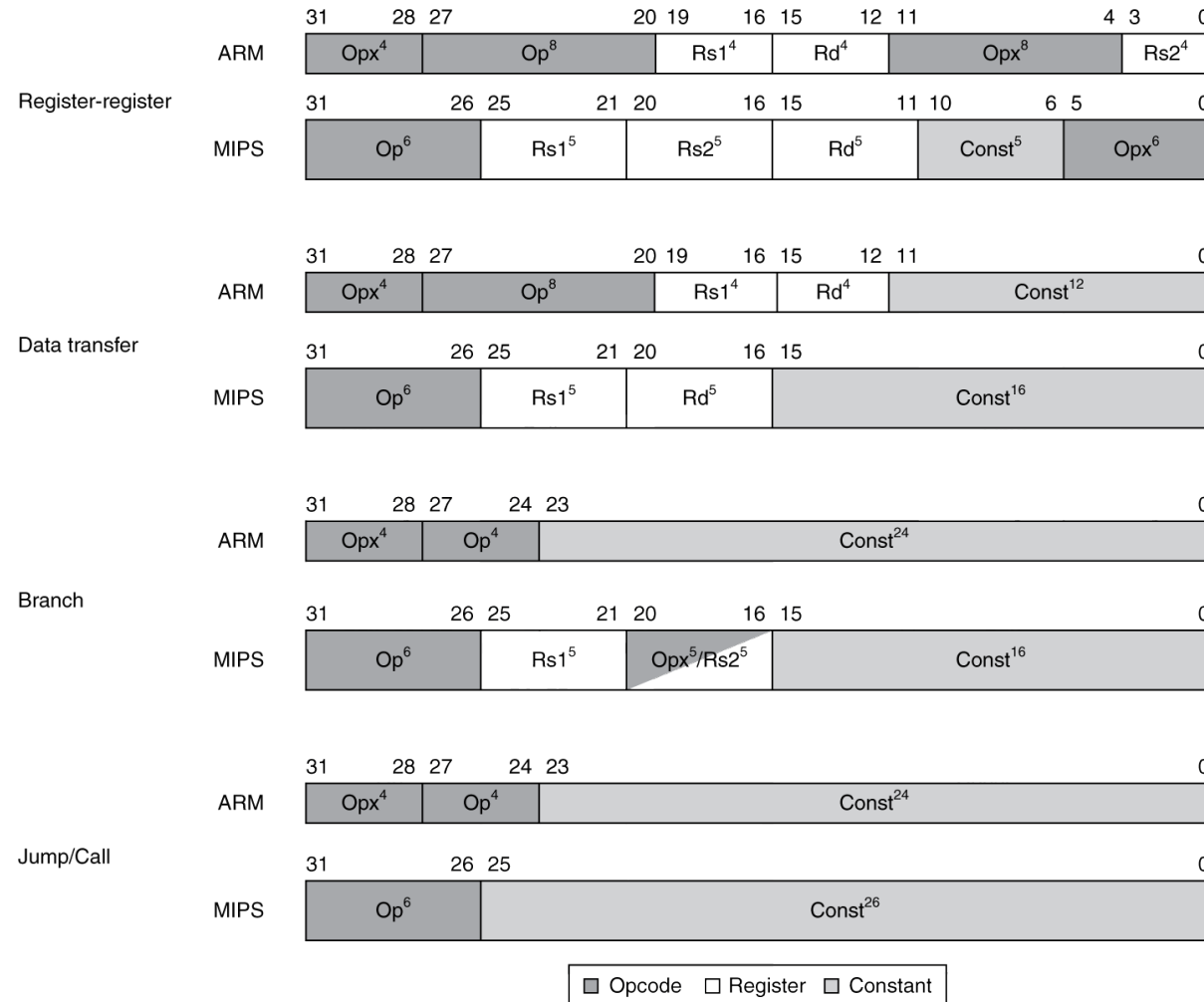


- ▶ Instruction set architecture
- ▶ The MIPS ISA
- ▶ Calling procedures in MIPS
- ▶ Dealing with character sets
- ▶ Translating and starting a program
- ▶ Pointers in C
- ▶ Effect of compiler optimization
- ▶ Other ISAs
- ▶ Concluding remarks



- ▶ 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



► ARM uses different principle for branch instructions



- ▶ 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



- ▶ 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
- ▶ 80486 (1989): pipelined, on-chip caches and FPU
 - Compatible competitors: AMD, Cyrix, ...



History of Intel Processors (x86 ISA) (2)

► P5 (Pentium) Architecture

- Pentium (1993)
 - 64-bit datapath
 - The infamous FDIV bug
 - Superscalar (more than one instruction per clock cycle possible)
- Pentium MMX (1997)
 - Multimedia extensions for increased computation capacity

► P6 Architecture (1995)

- Pentium Pro (1995)
- Pentium II (Pentium Pro + MMX, 1997)

- Pentium III (SSE: Streaming SIMD Extensions, 1999)

► Netburst (2000)

- Pentium IV: many problems (e.g., very long pipeline, high cpu clock, heat), abandoned

► PM Architecture (2003)

- Pentium M (2003) builds on P6, designed by Intel Israel originally for notebooks
- Intel Core (2006) after great success of Pentium M, adopted for desktop processors



Generations of Intel's Micro Architectures

- ▶ Core Architecture (1st generation), late 2006
 - Intel Core2, 64 bit support, first to offer VT-x
- ▶ Nehalem, late 2008
 - Front side bus (connection of CPU to system) replaced by QPI
- ▶ Sandy Bridge, 2010
 - Intel Core i Sandy (2010) AVX (advanced vector extensions)
 - Intel Core i Ivy (late 2011)
- ▶ Haswell, 2013
 - 22 nm, AVX2
- ▶ Skylake, late 2015
 - 14 nm
- ▶ Palm Cove, 2018
 - AVX-512 instruction set
- ▶ Sunny Cove
 - Low-power, mobile only
- ▶ Cypress Cove
 - Small update, fewer cycles for an L1 hit
- ▶ Wallow Cove
 - New manufacturing process
- ▶ Golden Cove
 - More instructions per cycle (IPC)
- ▶ Raptor Cove
 - Increased L2, L3 cache sizes



▶ Hyperthreading (HT)

- Multiple threads (lightweight processes) can be processed on one processor core
- Virtually doubles number of processor cores on Intel Core

▶ GPU

- Graphics processing unit (Grafikchip)

▶ vPro

- Enables remote management (e.g. remote BIOS configuration)

▶ TSX-NI

- Hardware transactional memory (batch-processed memory access, up to 40% faster)

▶ Turbo-Boost

- CPU automatically increases clock rate under load (overclocking)

▶ ACS

- Allows isolated iommu groups (for advanced hardware virtualization)

▶ VT-x, VT-d

- Hardware-based virtualization support

▶ AES-NI

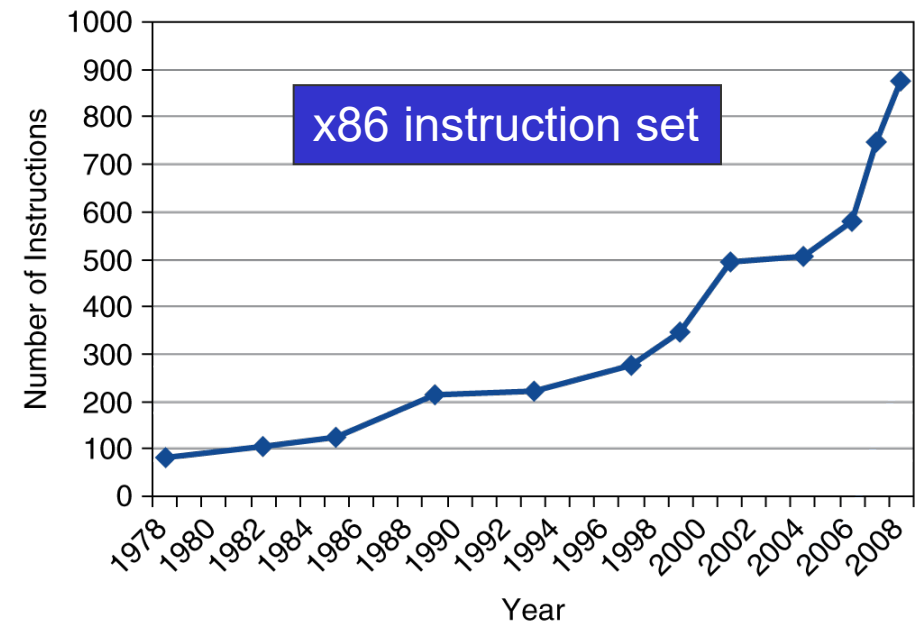
- AES: standard for symmetric encryption
- AES acceleration in hardware



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



- ▶ Powerful instructions $\Rightarrow$ 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 $\Rightarrow$ more errors and less productivity
- ▶ Backward compatibility
 - $\Rightarrow$ instruction set doesn't change
 - But they do accrete more instructions





- ▶ 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 is popped
 - But address can be passed back via an argument
 - Don't do that!



► Layers of software/hardware

- Compiler, assembler, hardware

► RISC ISAs

- Typical example: MIPS

► Design principles

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