182.690 RECHNERSTRUKTUREN - INSTRUCTIONS

Thomas Polzer
tpolzer@ecs.tuwien.ac.at
Institut für Technische Informatik

Instruction Set

- The collection of all instructions of a certain computer
- Different computers have different instruction sets
  - But many aspects common
- Early computers:
  - Very simple instruction sets
  - Simplified implementation
- Intermediate: Complex instruction set
- Modern computers
  - Back to simple instruction sets

MIPS-32 Instruction Set

- Used as the example throughout the book
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Typical example of many modern ISAs

MIPS-32 ISA

- 296 pages!
- Instruction Categories
  - Computational
  - Load/Store
  - Jump and Branch
  - Floating Point (coprocessor)
  - Memory Management
  - Special
- 3 Instruction Formats: all 32 bits wide

MIPS-32 Instruction Classes Distribution

- Frequency of MIPS instruction classes for SPEC2006

Instruction Class	Frequency	
	Integer	Floating point
Arithmetic	16%	**48%**
Data transfer	**35%**	**36%**
Logical	12%	4%
Cond. Branch	**34%**	8%
Jump	2%	0%

Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination

```
add a, b, c   ≡   a = b + c
```

- All arithmetic operations have this form
- Design Principle 1: Simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Arithmetic Operations (Example 1)

- C code:

```
a = b + c;
d = a – e;
```

- Compiled MIPS code:

```
add a, b, c
sub d, a, e
```

Arithmetic Operations (Example 2)

- C code:

```
f = (g + h) - (i + j);
```

- Compiled MIPS code:

```
add t0, g, h    # t0←g+h
add t1, i, j    # t1←i+j
sub f, t0, t1   # f←t0-t1
```

t0 and t1 are temporary results

Register Operands

- Arithmetic instructions:
  - Register operands
- MIPS has a 32 × 32-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data = a “word”
- Assembler names (convention)
  - `$t0, $t1, …, $t9` for temporary values
  - `$s0, $s1, …, $s7` for saved variables
- Design Principle 2: Smaller is faster
  - (main memory: millions of locations)

MIPS-32 Register File

- Faster than main memory
- Easier for a compiler to use
  - e.g., (A*B) – (C*D) – (E*F) multiplies in any order vs. stack
- Can hold variables
  - code density improves (registers named with fewer bits than memory locations)

Register Operand Example

- C code:

```
f = (g + h) - (i + j);
//f,   g,   h,   i,   j     →
//$s0, $s1, $s2, $s3, $s4
```

- Compiled MIPS code:

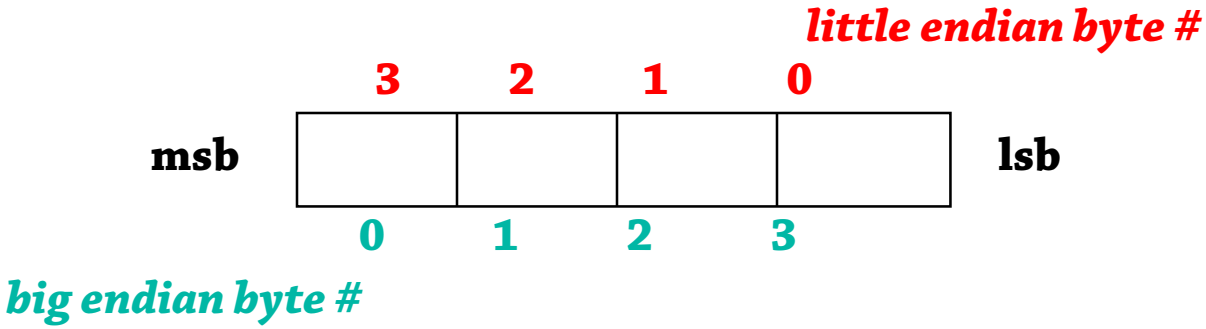
```
add $t0, $s1, $s2 # $t0←$s1+$s2
add $t1, $s3, $s4 # $t1←$s3+$s4
sub $s0, $t0, $t1 # $s0←$t0+$t1
```

Memory Operands

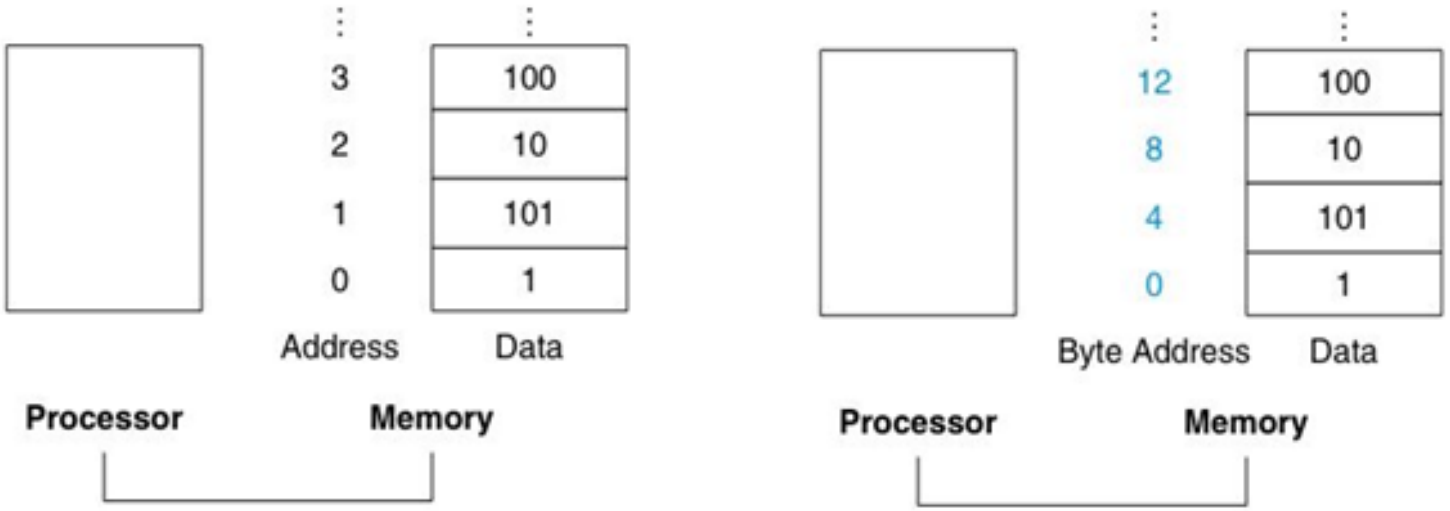
- Main memory used for composite data
  - Arrays, structures, dynamic data
- For arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Address must be a multiple of 4
- MIPS is Big Endian

Big Endian / Little Endian

- Big Endian:
  - Leftmost byte is word address
    - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Little Endian:
  - Rightmost byte is word address
    - Intel 80x86, DEC Vax, DEC Alpha



Memory Address vs. Byte Address



Memory Operand (Example 1)

- C code:

```
g = h + A[8];
//g: $s1, h: $s2
//base address of A: $s3
```

- Compiled MIPS code:


```
lw  $t0, 32($s3)  # load word
add $s1, $s2, $t0
```

//Index 8 requires offset of 32 (4 bytes per word)

Memory Operand (Example 2)

- C code:

```
A[12] = h + A[8];
//h: $s2, base address of A: $s3
```

- Compiled MIPS code:

```
lw  $t0, 32($s3)    # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)    # store word
```

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!

Immediate Operands

- Constant data specified in an instruction

```
addi $s3, $s3, 4
```

- No subtract immediate instruction
  - Just use negative constant:

```
addi $s2, $s1, -1
```

- Design Principle 3:
  Make the common case fast
  - Small constants common (therefore only 16 bit!)
  - Immediate operand avoids load instruction

Arithmetic Operations (Example 1)

- C code:

```
a = b + 128;
c = b – 128;
```

- Compiled MIPS code:

```
addi $s0, $s1, 128
addi $s2, $s1, -128
```

The Constant Zero

- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - e.g., move between registers

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

Unsigned Binary Integers

- Given an n-bit number

$$x = x^{n-1}\,2^{n-1} + x^{n-2}\,2^{n-2} + \ldots + x^{1}\,2^{1} + x^{0}\,2^{0}$$

- Range: 0 to $+2^n - 1$
- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bit

  0 to +4,294,967,295

2s-Complement Signed Integers

- Given an n-bit number

$$x = -x^{n-1}\,2^{n-1} + x^{n-2}\,2^{n-2} + \ldots + x^{1}\,2^{1} + x^{0}\,2^{0}$$

- Range: -2^{n-1} to $+2^{n-1}-1$
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^{2} + 0\times2^{1} + 0\times2^{0}$
    $= -2{,}147{,}483{,}648 + 1073741824 + \ldots + 4 + 0 + 0 =$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 32 bits

$$-2{,}147{,}483{,}648 \text{ to } +2{,}147{,}483{,}647$$

2s-Complement Signed Integers

- Bit 31 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- $-(-2^n-1)$ can't be represented
- Non-negative numbers:
  - Same unsigned and 2s-complement representation
- Some specific numbers
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111

Signed Negation

- Complement and add 1
  - Complement means $1 \rightarrow 0$, $0 \rightarrow 1$

$$x + \bar{x} = 1111\ldots111_2 = -1$$
$$\bar{x} + 1 = -x$$

- Example: negate +2
  - $+2 = 0000\ 0000\ \ldots\ 0010_2$
  - $-2 = 1111\ 1111\ \ldots\ 1101_2 + 1 = 1111\ 1111\ \ldots\ 1110_2$

Sign Extension

- Representing a number using more bits
  - Preserve the numeric value
- In MIPS instruction set
  - `addi`: extend immediate value
  - `lb, lh`: extend loaded byte/halfword
  - `beq, bne`: extend the displacement
- Replicate the sign bit to the left
  - compare unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110

Representing Instructions

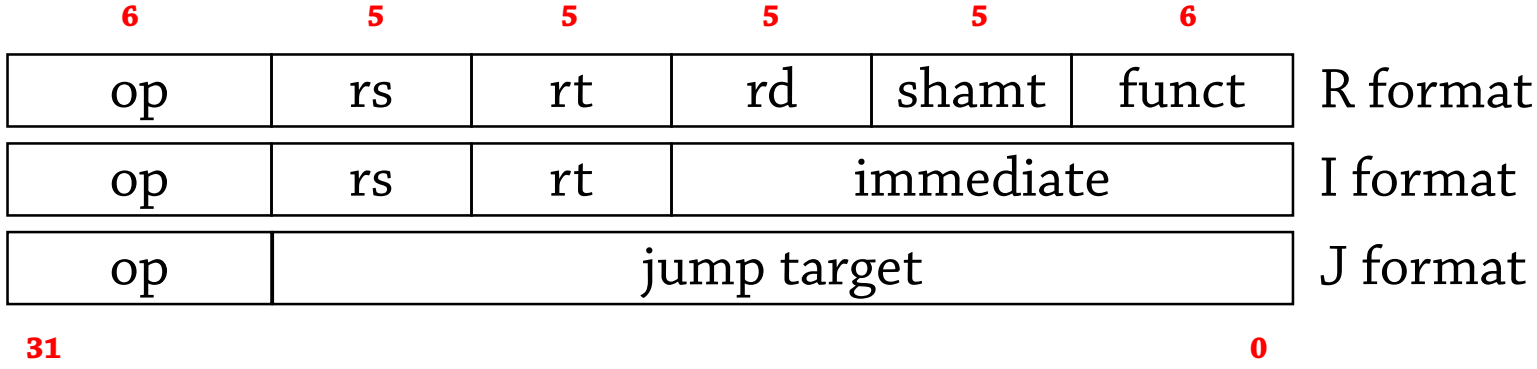
- Instructions are encoded in binary
  - Called machine code
- MIPS instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding
  - operation code (opcode), register numbers, …
  - Regularity!
- Register numbers (convention)
  - $t0 – $t7 are reg's 8 – 15
  - $t8 – $t9 are reg's 24 – 25
  - $s0 – $s7 are reg's 16 – 23

MIPS Register Convention

Name	Reg. No.	Usage	Preserve on call?
$zero	0	Const. 0	n. a.
$at	1	Reserved for assembler	n. a.
$v0 - $v1	2 - 3	Return values	No
$a0 - $a3	4 – 7	Arguments	Yes
$t0 - $t7	8 – 15	Temporary values	No
$s0 - $s7	16 - 23	Saved values	Yes
$t8 - $t9	24 – 25	Temporary values	No
$k0 - $k1	26 – 27	Reserved for OS kernel	
$gp	28	Global pointer	Yes
$sp	29	Stack pointer	Yes
$fp	30	Frame pointer	Yes
$ra	31	Return address	Yes

MIPS-32 ISA

- 3 Instruction Formats: all 32 bits wide



MIPS-32 R-Format

- Fields are given names to make them easier to refer to

- op 6-bits, opcode specifying operation
- rs 5-bits, reg. file address first source operand
- rt 5-bits, reg. file address second source operand
- rd 5-bits, reg. file address result destination
- shamt 5-bits, shift amount (for shift instructions)
- funct 6-bits, function code augmenting the opcode

MIPS-32 R-Format

Instruction	Opcode	Function
add	0	20_h
addu	0	21_h
and	0	24_h
nor	0	27_h
or	0	25_h
sub	0	22_h
subu	0	23_h

R-format (Example)

op	rs	rt	rd	shamt	funct

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

$t0 → R8, $s1 → R17, $s2 → R18

special	$s1	$s2	$t0	0	add
0	17	18	8	0	20_h
000000	10001	10010	01000	00000	100000

$000000 | 10001 | 10010 | 01000 | 00000 | 100000_2 = 02324020_h$

R-format (Example)

op	rs	rt	rd	shamt	funct

```
subu $s0, $t1, $t2
```

$s0 → R16, $t1 → R9, $t2 → R10

special	$t1	$t2	$s0	0	subu
0	9	10	16	0	23_h
000000	01001	01010	10000	00000	100011

$000000 | 01001 | 01010 | 10000 | 00000 | 100011_2 = 012a8023_h$

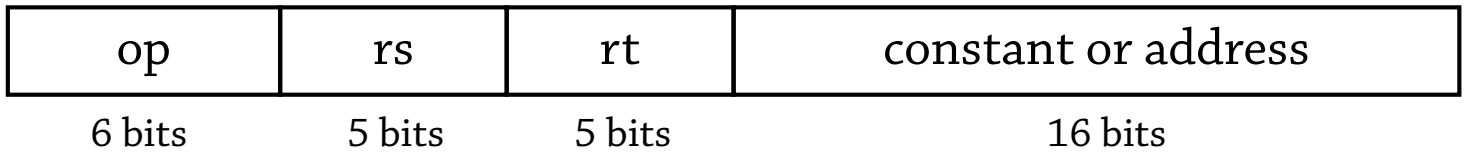
Hexadecimal

- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Example: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

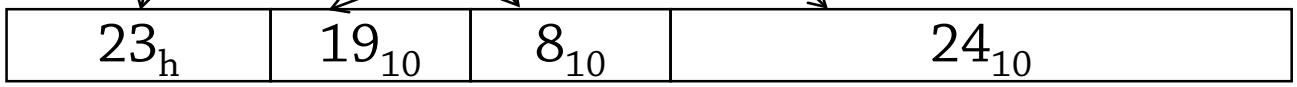
MIPS-32 I-Format



- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: -2^{15} to $+2^{15}-1$
  - Address: offset added to base address in rs
- Design Principle 4:
  Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

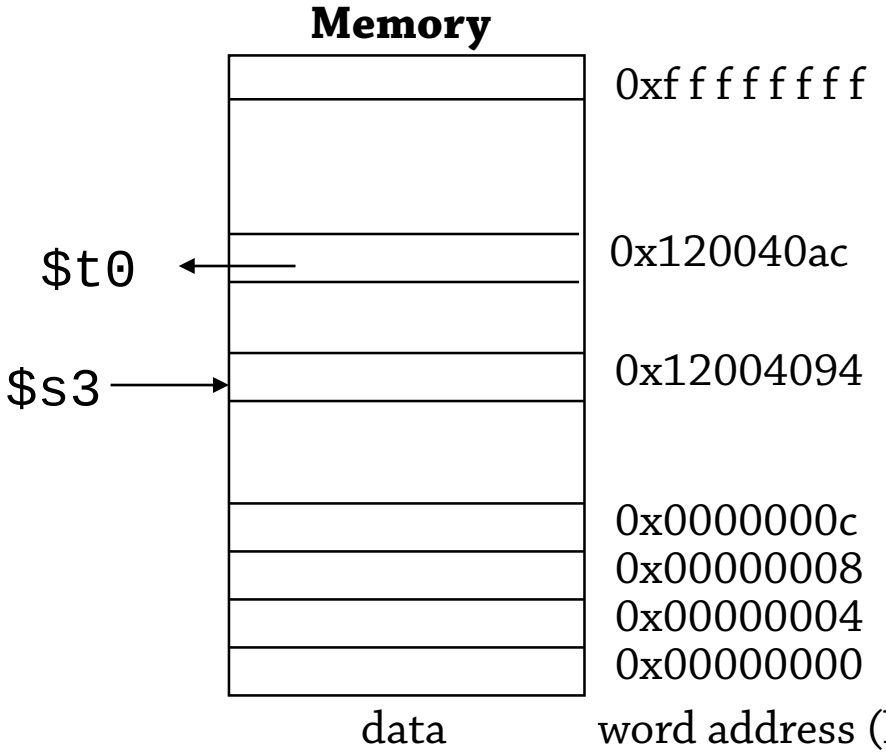
Load (Example)

```
lw $t0, 24($s3)
```



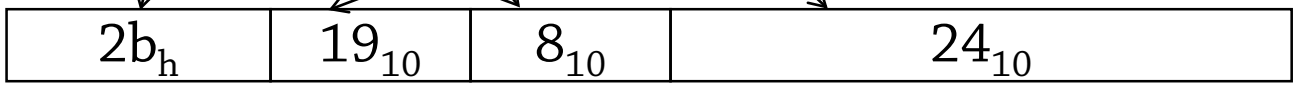
24_{10} + $s3 =

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



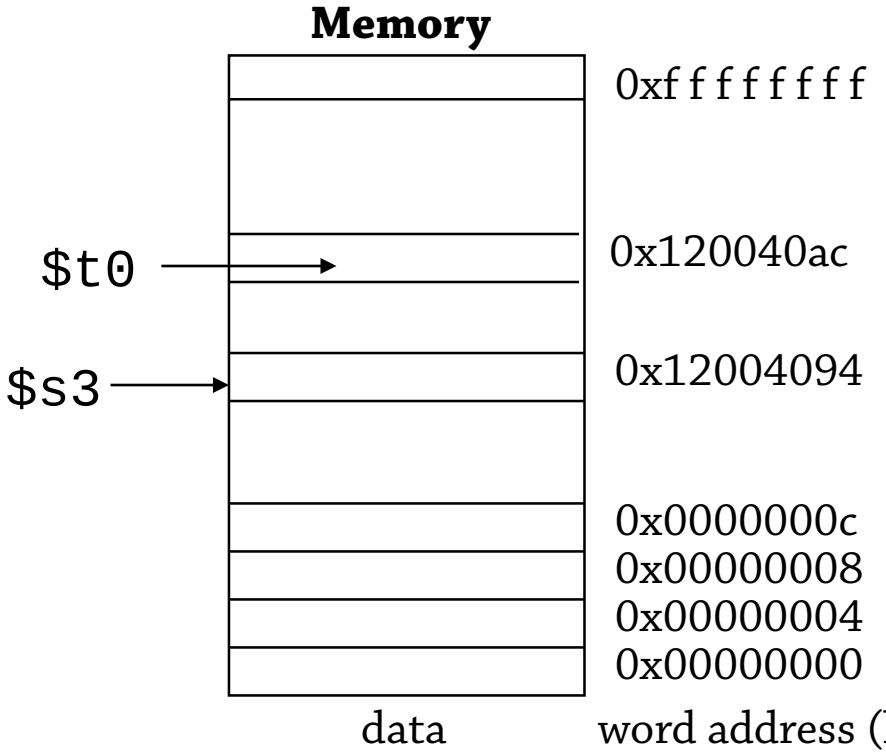
Store (Example)

```
sw $t0, 24($s3)
```



24_{10} + $s3 =

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



Machine Code (Example)

```
A[300] = h + A[300];
```

Base address of A: $t1
h: $s2

```
lw  $t0, 1200($t1)
add $t0, $s2, $t0
sw $t0, 1200($t1)
```

23_h	9_{10}	8_{10}	1200_{10}		
0	18_{10}	8_{10}	8_{10}	0	20_h
$2b_h$	9_{10}	8_{10}	1200_{10}		

$1000\ 11 \mid 01\ 001 \mid 0\ 1000 \mid 0000\ 0100\ 1011\ 0000_2 = 8d28\ 04b0_h$

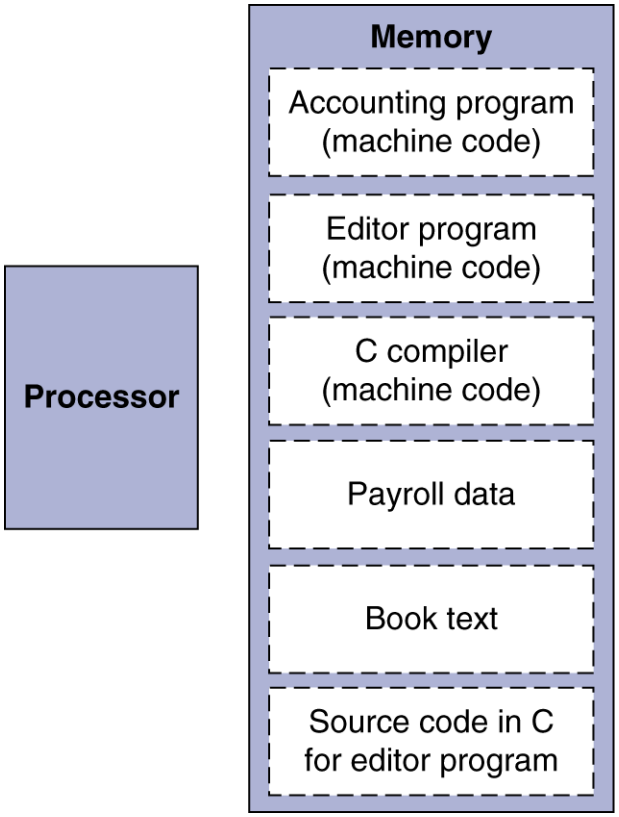
$0000\ 00 \mid 10\ 010 \mid 0\ 1000 \mid 0100\ 0 \mid 000\ 00 \mid 11\ 0000_2 = 0248\ 4030_h$

$1010\ 11 \mid 01\ 001 \mid 0\ 1000 \mid 0000\ 0100\ 1011\ 0000_2 = ad28\ 04b0_h$

MIPS-32 (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

Stored Program Computers



- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	MIPS-32	Type	Opcode	Function
Shift left	<<	<<	sll	R	0	00_h
Shift right	>>	>>>	srl	R	0	02_h
Bitwise AND	&	&	and andi	R I	0 c_h	24_h ---
Bitwise OR	|	|	or ori	R I	0_h d_h	25_h ---
Bitwise NOT	~	~	nor	R	0_h	27_h

- Useful for extracting and inserting groups of bits in a word

Shift Operations

sll:	0	---	rt	rd	shamt	00
srl:	0	---	rt	rd	shamt	02

- shamt: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - sll by i bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - srl by i bits divides by 2^i (unsigned only)

Shift Operations (Example)

- C Code:

```
x = y << 4;
```

- Compiled MIPS code:

```
sll $t2, $s0, 4
```

- Machine Code:

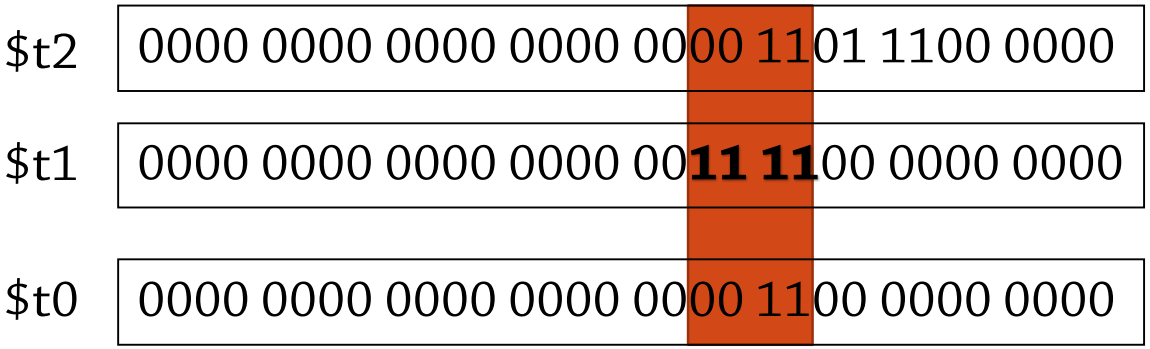
0	0	16	10	4	00_h

$0000\ 00 \mid 00\ 000 \mid 1\ 0000 \mid 0101\ 0 \mid 000\ 10 \mid 00\ 0000_2 =$
$0010\ 5080_h$

AND Operations

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

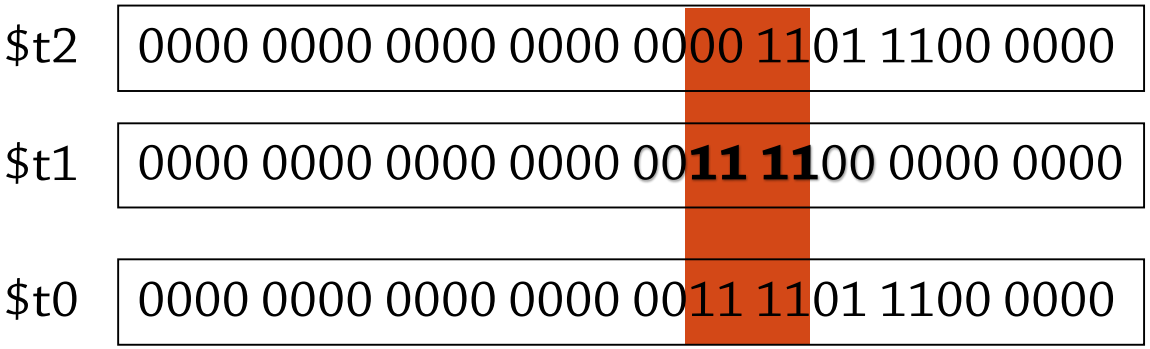
```
and $t0, $t1, $t2
```



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)

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

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

182.690 RECHNERSTRUKTUREN - INSTRUCTIONS

Thomas Polzer
tpolzer@ecs.tuwien.ac.at
Institut für Technische Informatik