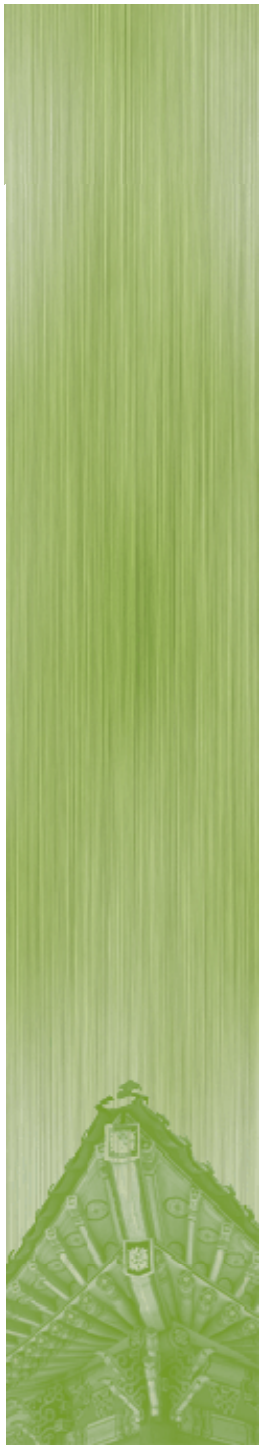


MIPS Instruction Set Architecture I

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu

Architecture (1)

"the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, the logical design, and the physical implementation"

-- Amdahl, Blaauw, and Brooks, Architecture of the IBM System/360, IBM Journal of Research and Development, April 1964.

- The visible interface between software and hardware
- What the user (OS, compiler, …) needs to know to reason about how the machine behaves
- Abstracted from the details of how it may accomplish its task

Architecture (2)

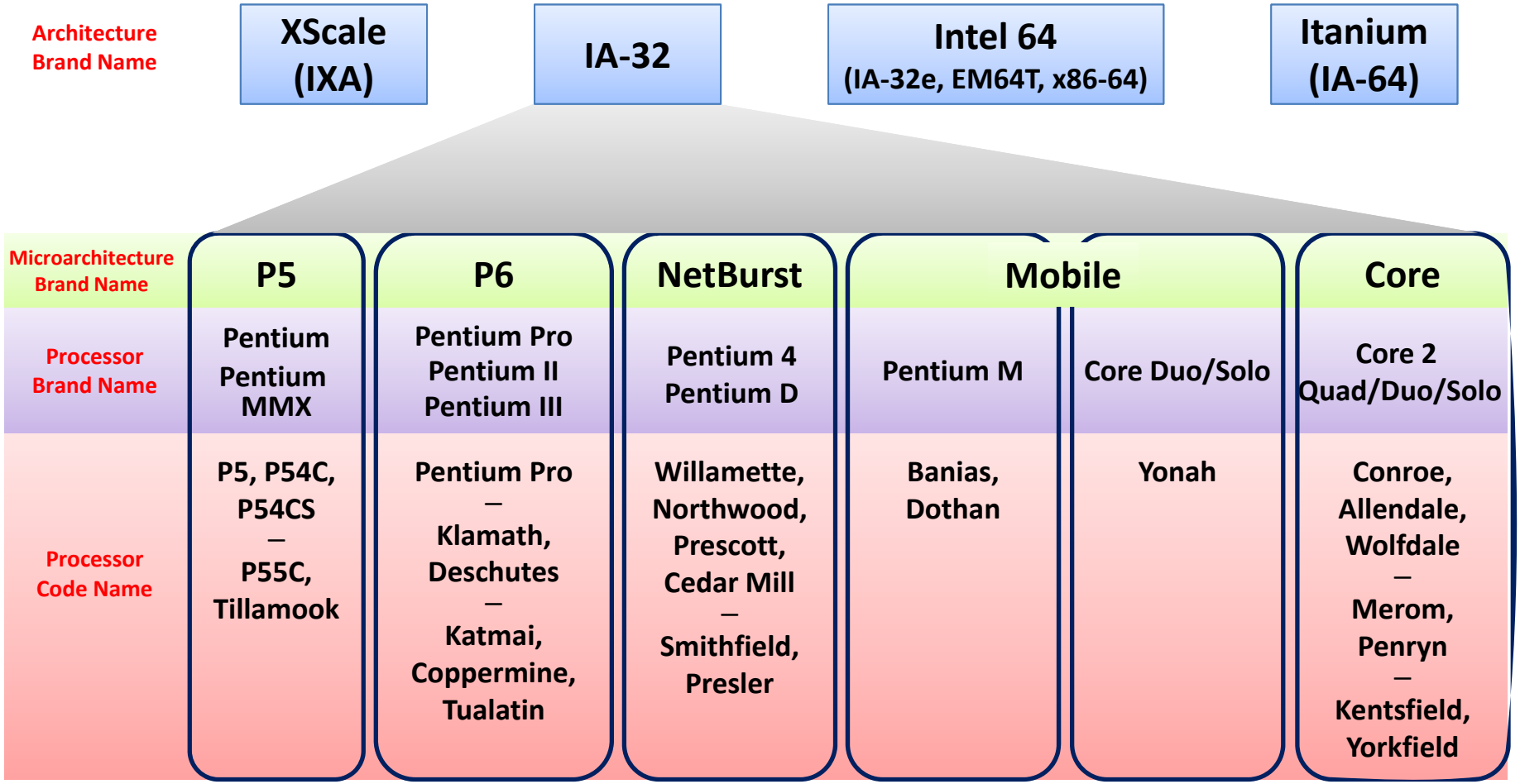
- **Computer "Architecture" defines**
  - Instruction set architecture (ISA)
    - Instruction set
    - Operand types
    - Data types (integers, FPs, …)
    - Memory addressing modes, …
  - Registers and other state
  - The interrupt/exception model
  - Memory management and protection
  - Virtual and physical address layout
  - I/O model
  - …

Architecture (3)

- **Microarchitecture**
  - Organization of the machine below the visible level
    - Number/location of functional units
    - Pipeline/cache configurations
    - Programmer transparent techniques: prefetching, …
  - Must provide same meaning (semantics) as the visible architecture model

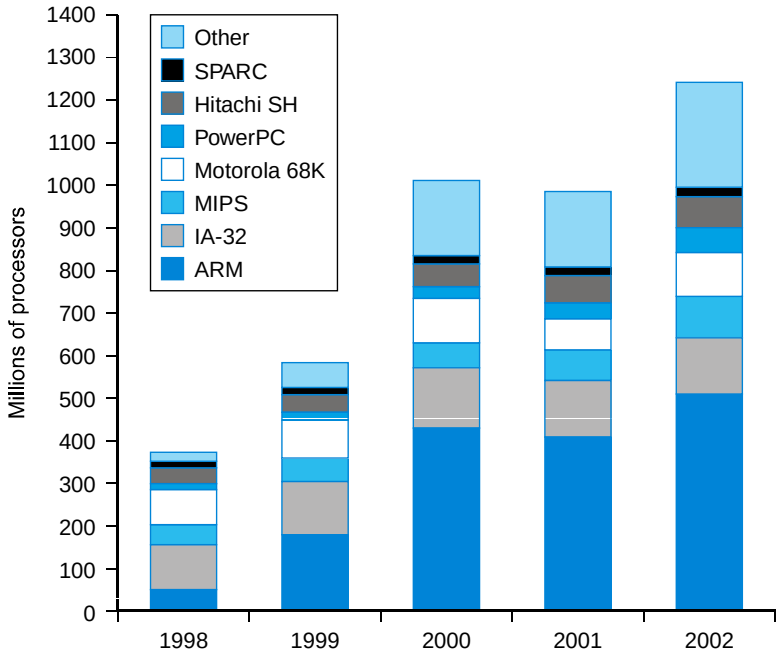
- **Implementation**
  - Hardware realization
  - Logic circuits, VLSI technology, process, …

Architecture (4)



Instruction Set (1)

- **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
    - "RISC (Reduced Instruction Set Computer)"



Instruction Set (2)

- **MIPS instruction set**
  - Similar to other architectures developed since 1980's
  - Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
  - Large share of embedded core market
    - Applications in consumer electronics, network/storage equipment, cameras, printers, …
    - Almost 100 million MIPS processors manufactured in 2002
    - Used by NEC, Nintendo, Cisco, Silicon Graphics, Sony, …
  - Typical of many modern ISAs
    - See MIPS Reference Data tear-out card ("green card"), and Appendixes B and E

Arithmetic Operations (1)

- **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 (2)

- **Arithmetic example**
  - C code:

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

  - Compiled MIPS code:

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

Operands (1)

- **Register operands**
  - Arithmetic instructions use register operands
  - MIPS has a 32 x 32-bit register file
    - Use for frequently accessed data
    - Numbered 0 to 31
    - 32-bit data called a "word"
  - Assembler names
    - `$t0`, `$t1`, …, `$t9` for temporary values
    - `$s0`, `$s1`, …, `$s7` for saved variables
  - **Design Principle 2**: Smaller is faster
    - (cf.) Main memory: millions of locations

Operands (2)

- **Register operand example**
  - C code:

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

    – f, g, h, i, j in `$s0`, `$s1`, `$s2`, `$s3`, `$s4`

  - Compiled MIPS code:

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

Operands (3)

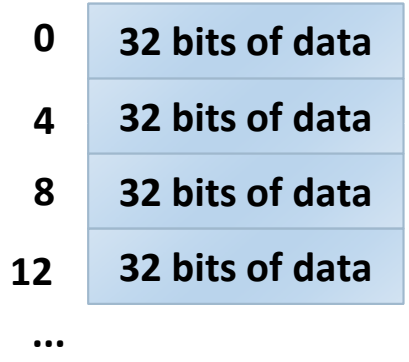
- MIPS registers

#	Name	Usage
0	$zero	The constant value 0
1	$at	Assembler temporary
2	$v0	Values for results and expression evaluation
3	$v1	
4	$a0	Arguments
5	$a1	
6	$a2	
7	$a3	
8	$t0	Temporaries (Caller-save registers)
9	$t1	
10	$t2	
11	$t3	
12	$t4	
13	$t5	
14	$t6	
15	$t7	

#	Name	Usage
16	$s0	Saved temporaries (Callee-save registers)
17	$s1	
18	$s2	
19	$s3	
20	$s4	
21	$s5	
22	$s6	
23	$s7	
24	$t8	More temporaries (Caller-save registers)
25	$t9	
26	$k0	Reserved for OS kernel
27	$k1	
28	$gp	Global pointer
29	$sp	Stack pointer
30	$fp	Frame pointer
31	$ra	Return address

Operands (4)

- **Memory operands**
  - Main memory used for composite data
    - Arrays, structures, dynamic data
  - To apply 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
    - Most-significant byte at least address of a word



Operands (5)

- **Memory operand example 1**
  - C code:

```
g = h + A[8];
```

    - g in $s1, h in $s2, base address of A in $s3

  - Compiled MIPS code:
    - 4 bytes per word
    - Index 8 requires offset of 32

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

offset base register

Operands (6)

- **Memory operand example 2**
  - C code:

```
A[12] = h + A[8];
```

    - h in $s2, base address of A in $s3

  - Compiled MIPS code:
    - Index 8 requires offset of 32

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

Operands (7)

- **Register 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

Operands (8)

- **Immediate operands**
  - Constant data specified in an instruction

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

  - No subtract immediate instruction
    - Just use a negative constant

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

  - **Design Principle 3**: Make the common case fast
    - Small constants are common
    - Immediate operand avoids a load instruction

Operands (9)

- **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
```

Representing Data (1)

- **Unsigned binary integers**
  - Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

  - 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 bits
    - 0 to +4,294,967,295

Representing Data (2)

- **2's-complement signed integers**
  - Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

  - Range: -2^{n-1} to $+2^{n-1} - 1$
  - Example
    - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
      $= -1 \times 2^{31} + 1 \times 2^{30} + \ldots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
      $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
  - Using 32 bits
    - $-2{,}147{,}483{,}648$ to $+2{,}147{,}483{,}647$

Representing Data (3)

- **2's-complement signed integers (cont'd)**
  - Bit 31 is sign bit
    - 1 for negative numbers
    - 0 for non-negative numbers
  - $-(-2^{n-1})$ can't be represented
  - Non-negative numbers have the same unsigned and 2's-complement representation
  - Some specific numbers:

```
                0:  0000 0000 ... 0000
               -1:  1111 1111 ... 1111
Most-negative:  1000 0000 ... 0000
Most-positive:  0111 1111 ... 1111
```

Representing Data (4)

- **Signed negation**
  - Complement and add 1
    - Complement means 1 → 0, 0 → 1

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

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

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

Representing Data (5)

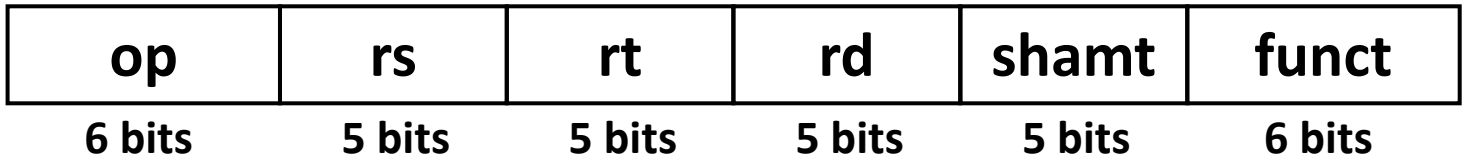
- **Sign extension**
  - Representing a number using more bits
    - Preserve the numeric value
  - In MIPS instruction set
    - `addi`: extend immediate value
    - `lb`, `lh`: extend loaded byte/halfword
    - `beq`, `bne`: extend the displacement
  - Replicate the sign bit to the left
    - (cf.) 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 (1)

- **Representing instructions in MIPS**
  - 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
    - `$t0` – `$t7` are registers 8 – 15
    - `$t8` – `$t9` are registers 24 – 25
    - `$s0` – `$s7` are registers 16 – 23

Representing Instructions (2)

- **MIPS R-format instructions**



  - Instruction fields
    - **op**: operation code (opcode)
    - **rs**: first source register number
    - **rt**: second source register number
    - **rd**: destination register number
    - **shamt**: shift amount (00000 for now)
    - **funct**: function code (extends opcode)

Representing Instructions (3)

- MIPS 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

$00000010001100100100000000100000_2 = 02324020_{16}$

Representing Instructions (4)

- **MIPS I-format instructions**

  - 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

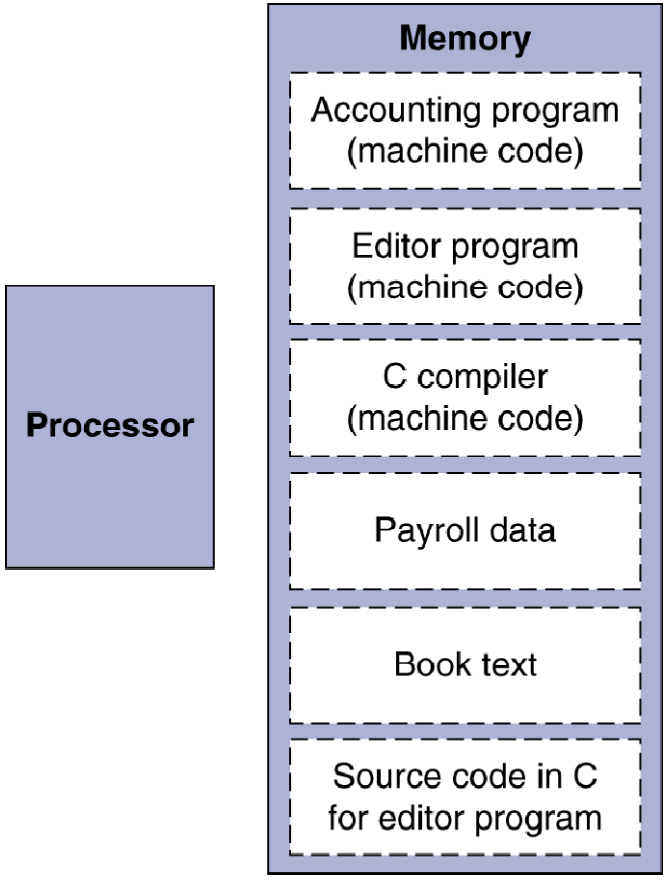
Representing Instructions (5)

- **MIPS J-format instructions**

  - Jump instructions (`j` and `jal`)
    - **Address**: encodes 26-bit target address

Representing Instructions (6)

- **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 (1)

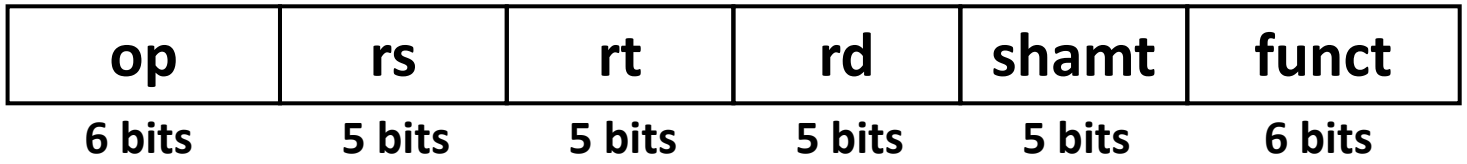
- **Logical operations**
  - Instructions for bitwise manipulation

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

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

Logical Operations (2)

- **Shift operations**

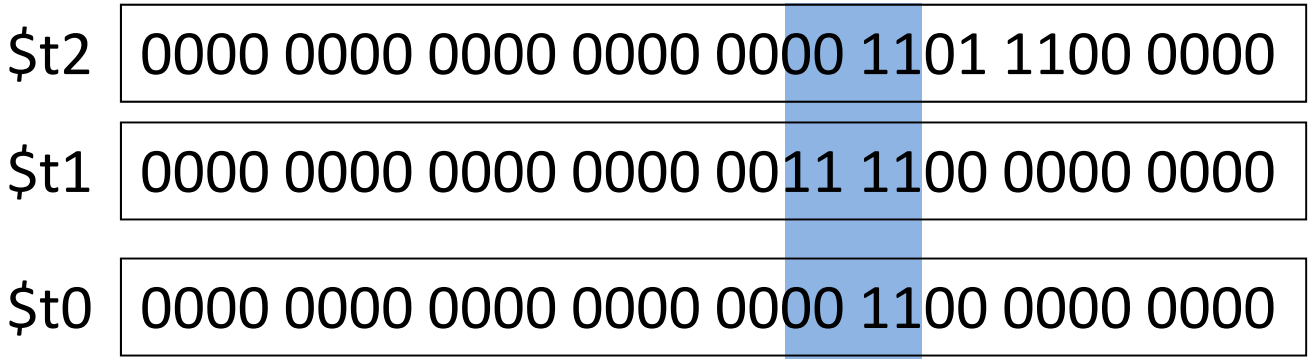


  - **shamt**: how many positions to shift
  - Shift left logical
    - Shift left and fill with 0 bits
    - `sll` by *i* bits multiples by 2^i
  - Shift right logical
    - Shift right and fill with 0 bits
    - `srl` by *i* bits divides by 2^i (unsigned only)

Logical Operations (3)

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

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



Logical Operations (4)

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

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

$t2	0000 0000 0000 0000 0000 1101 1100 0000
$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	0000 0000 0000 0000 0011 1100 0000 0000

Logical Operations (5)

- **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