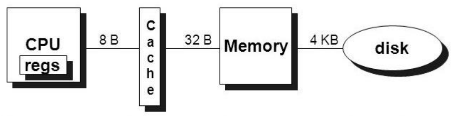
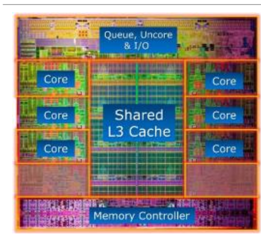
Registers vs. Memory

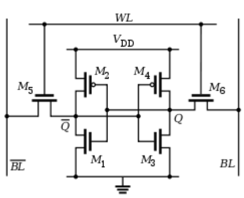
- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible

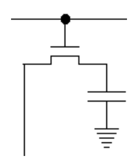


	Register	Cache	Memory	Disk Memory
size:	32 B	32 KB-4MB	1024 MB	100 GB
speed:	1 ns	2 ns	30 ns	8 ms
$/Mbyte:		$125/MB	$0.20/MB	$0.001/MB
line size:	8 B	32 B	4 KB	

Register vs. Memory







# of Cores	4
Processor Base Frequency	1.80 GHz
Cache	8 MB SmartCache
TDP	15 W
Configurable TDP-up	25 W
Configurable TDP-down	10 W

Density	8 Gb
Voltage(V)	1.2V
Power	C
Package	78FBGA

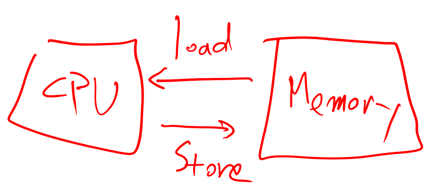
Arithmetic Example

- C code:
  - f = (g + h) - (i + j);
- Compiled MIPS code:
  - add $t0 $s1 $s2 # temp t0 = g + h
  - add $t1 $s3 $s4 # temp t1 = i + j
  - sub $s0 $t0 $t1 # f = t0 - t1

s1 s2 s3 s4 s0

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
  - c.f. Little Endian: least-significant byte at least address



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:
  - Index 8 requires offset of 32
    - 4 bytes per word
  - lw $t0, 32 ($s3)
  - add $s1, $s2, $t0

A[0] → 0($s3)
A[1] → 4($s3)
⋮
A[8] → 32($s3)

int A[ ];
*A ⇐
A[1]
⇒ *(A+1)
A[0]
⇒ *(A+0)

CSULB

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
  - $t0 ← h + A[8] ← Not MIPS
  - sw $t0, 48 ($s3)
  -

A[0] = h + A[8]
sw $t0, 0 ($s3)

CSULB

Immediate Operands

- Constant data specified in an instruction
  - addi $s3, $s3, 4
    D S number
- No subtract immediate instruction subi
  - 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

CSULB

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   # = D + ∅

CSULB

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
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$

$= 1*2^0 + 1*2^1 + 1*2^3 = 11_{10}$

- Using 32 bits
- 0 to +4,294,967,295

CSULB

2s-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 $+2^{n-1} - 1$

not 2^{n-1}

- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$

$= -1*2^{31} + \cdots + 0*2^0 = -4_{10}$

0000 ~ 0011

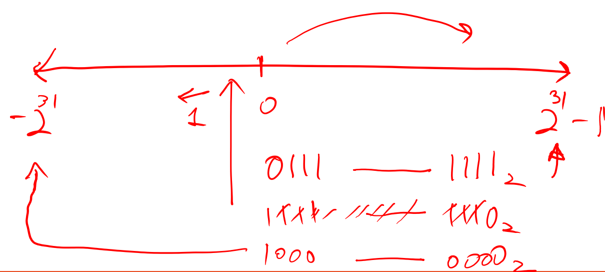
0100

- Using 32 bits
- –2,147,483,648 to +2,147,483,647

CSULB

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. ~~Why? Because of 0.~~
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
  - 0: $0000 \cdots 0000_2$
  - –1: $1111 \cdots 1111_2$
  - Most-negative: $1000 \sim 0000_2$
  - Most-positive: $0111 \sim 1111_2$



Signed Negation

- Complement and add 1
  - Complement means 1 → 0, 0 → 1

$1101_2 \leftrightarrow 0010_2$

$x \qquad \bar{x}$

$$x + \bar{x} = 1111 \sim 1111_2 = -1_{10}$$

$$\Rightarrow \bar{x} + 1 = -x \;*$$

$$\begin{array}{r} 1101 \\ +)\ 0010 \\ \hline 1111 \end{array}$$

- Example: negate +2
  - +2 = $0000 \sim 0010$
  - – 2 = $1111 \sim 1101$

$$\begin{array}{r} 1111 \sim 1101 \\ +)\qquad\quad 1 \\ \hline 1111 \sim 1110_2 \end{array}$$

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
  - c.f. 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_2 ?

    0000 0000 1111 1110_2

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!

MIPS 00000 1010 ~

← 32 bits →

- Register numbers
  - \$t0 – \$t7 are reg's 8 ~ 15
  - \$t8 – \$t9 are reg's 24 ~ 25
  - \$s0 – \$s7 are reg's 16 ~ 23

MIPS R-format Instructions

- Instruction fields
  - op: operation code
  - rs: 1st source register
  - rt: 2nd source register
  - rd: destination
  - shamt: shift amount
  - funct: function code

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

R-format example

- add $t0, $s1, $s2 ← Human

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
0	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

← computer

32 bits

$?_2 = \underline{02324020}_{16}$

Hexadecimal

- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit
- Example: eca8 6420
  - 1110 1100 1010 1000

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

//9/5