Computer Organization and Architecture

Module 1

Evolution of Computer Systems, MIPS Architecture and Instruction Set

Prof. Indranil Sengupta

Dr. Sarani Bhattacharya

Department of Computer Science and Engineering

IIT Kharagpur

1

Evolution of Computer Systems

2

Introduction

- Computers have become part and parcel of our daily lives.
  - They are everywhere (embedded systems?)
  - Laptops, tablets, mobile phones, intelligent appliances.
- It is required to understand how a computer works.
  - What are there inside a computer?
  - How does it work?
- We distinguish between two terms: *Computer Architecture* and *Computer Organization.*

3

3

- **Computer Organization:**
  - Design of the components and functional blocks using which computer systems are built.
  - **Analogy**: civil engineer's task during building construction (cement, bricks, iron rods, and other building materials).
- **Computer Architecture:**
  - How to integrate the components to build a computer system to achieve a desired level of performance.
  - **Analogy**: architect's task during the planning of a building (overall layout, floorplan, etc.).

4

4

Historical Perspective

- Constant quest of building automatic computing machines have driven the development of computers.
  - Initial efforts: mechanical devices like pulleys, levers and gears.
  - During World War II: mechanical relays to carry out computations.
  - Vacuum tubes developed: first electronic computer called ENIAC.
  - Semiconductor transistors developed and journey of miniaturization began.
    - SSI → MSI → LSI → VLSI → ULSI → …. Billions of transistors per chip

5

5

PASCALINE (1642)

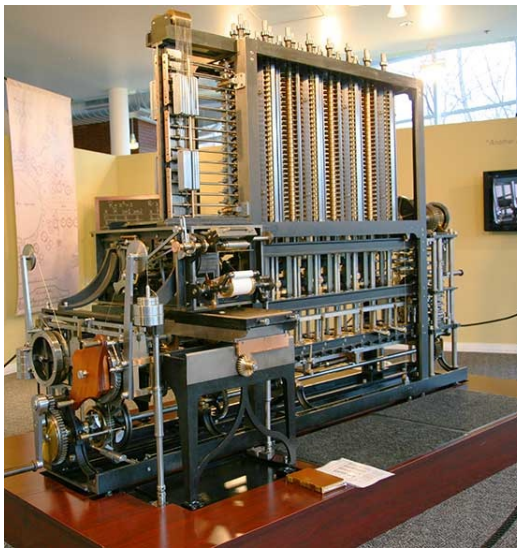
- Mechanical calculator invented by B. Pascal.
- Could add and subtract two numbers directly, and multiply and divide by repetition.

6

6

Babbage Engine

- First automatic computing engine was designed by Charles Babbage in the 19th century, but he could not build it.
- The first complete Babbage engine was built in 2002, 153 years after it was designed.
  - 8000 parts.
  - Weighed 5 tons.
  - 11 feet in length.

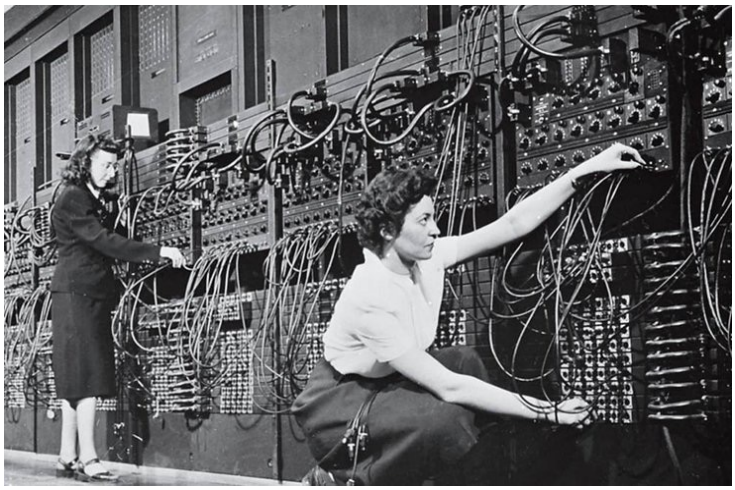


7

7

ENIAC
(Electrical Numerical Integrator and Calculator)

- Developed at the University of Pennsylvania in 1945.
- Used 18,000 vacuum tubes, weighed 30 tons, and occupied a 30ft x 50ft space.



8

8

Harvard Mark 1

- Built at the University of Harvard in 1944, with support from IBM.
- Used mechanical relays (switches) to represent data.
- It weighed 35 tons, and required 500 miles of wiring.

9

9

IBM System/360

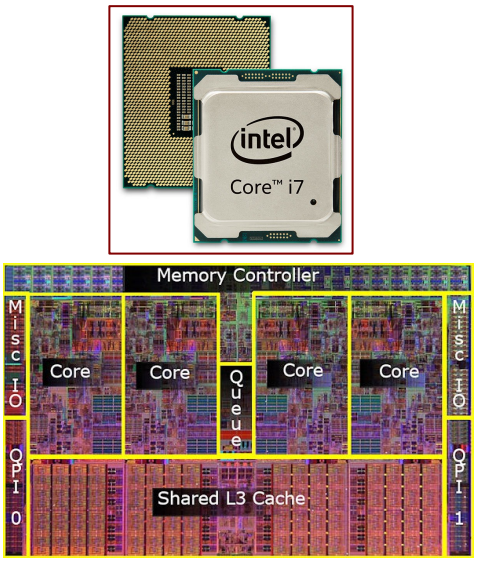
- Very popular mainframe computer of the 60s and 70s.
- Introduced many advanced architectural concepts that appeared in microprocessors several decades later.

10

10

Intel Core i7

- A modern processor chip, that comes in dual-core, quad-core and 6-core variants.
- 64-bit processor that comes with various microarchitectures like Haswell, Nehalem, Sandy Bridge, etc.



11

11

Generation	Main Technology	Representative Systems
First (1945-54)	Vacuum tubes, relays	Machine & assembly language ENIAC, IBM-701
Second (1955-64)	Transistors, memories, I/O processors	Batch processing systems, HLL IBM-7090
Third (1965-74)	SSI and MSI integrated circuits Microprogramming	Multiprogramming / Time sharing IBM 360, Intel 8008
Fourth (1975-84)	LSI and VLSI integrated circuits	Multiprocessors Intel 8086, 8088
Fifth (1984-90)	VLSI, multiprocessor on-chip	Parallel computing, Intel 486
Sixth (1990 onwards)	ULSI, scalable architecture, post-CMOS technologies	Massively parallel processors Pentium, SUN Ultra workstations

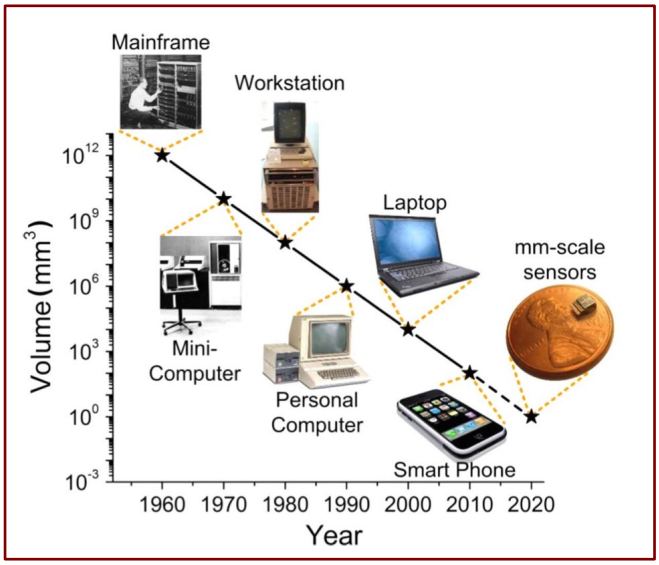
12

12

Evolution of the Types of Computer Systems

The future?

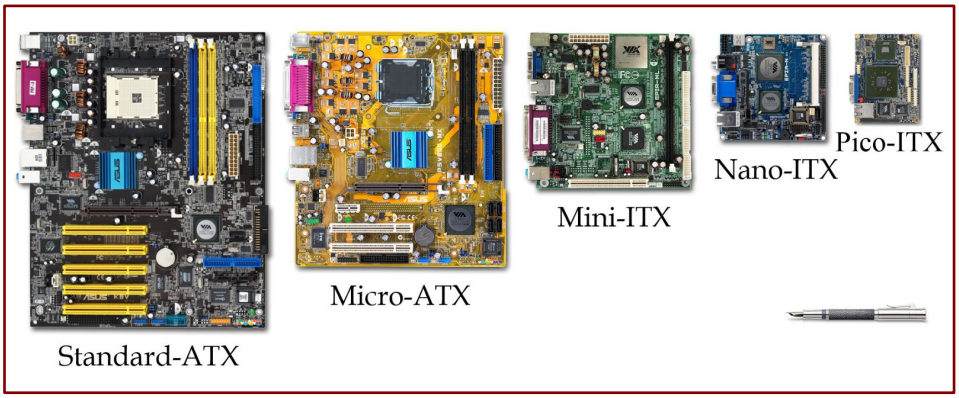
- Large-scale IoT based systems.
- Wearable computing.
- Intelligent objects.



13

13

Evolution of PC form factors over the years

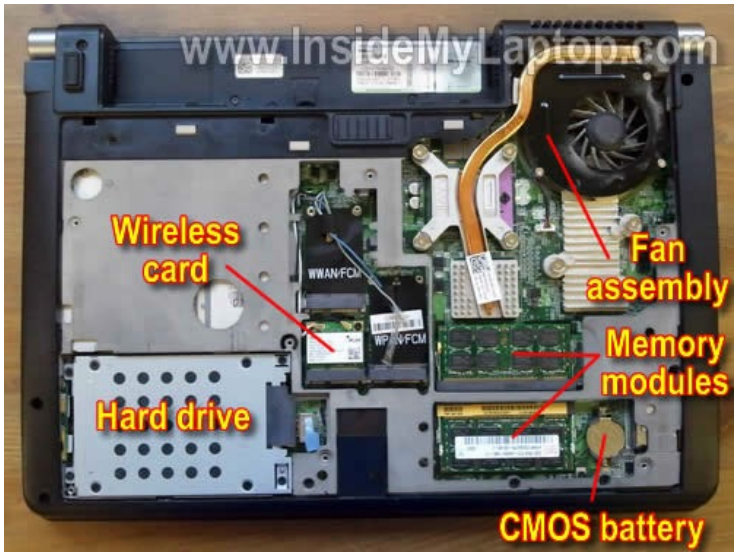


14

14

Inside a laptop

- Miniaturization in feature sizes of all parts.
- Hard drive getting replaced by flash-based memory devices.
- Cooling is a major issue.



15

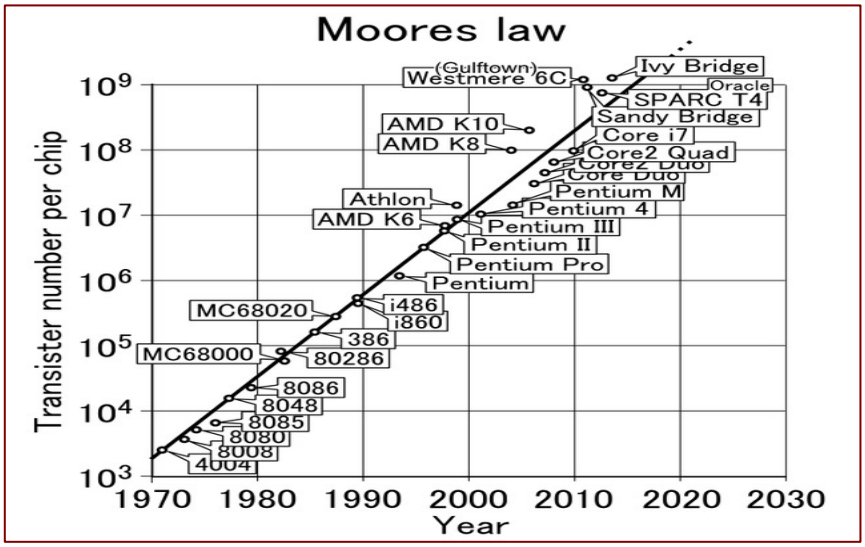
15

Moore’s Law

- Refers to an observation made by Intel co-founder Gordon Moore in 1965. He noticed that the number of transistors per square inch on integrated circuits had doubled every year since their invention.
- Moore's law predicts that this trend will continue into the foreseeable future.
- Although the pace has slowed, the number of transistors per square inch has since doubled approximately every 18 months. This is used as the current definition of Moore's law.

16

16

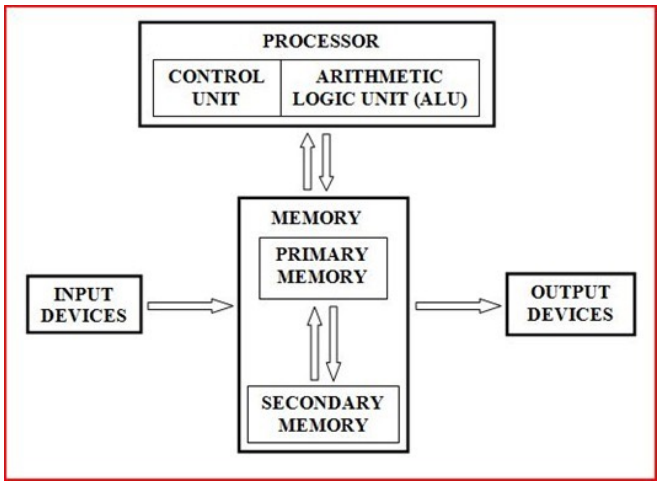


17

17

Simplified Block Diagram of a Computer System

- All instructions and data are stored in memory.
- An instruction and the required data are brought into the processor for execution.
- Input and Output devices interface with the outside world.
- Referred to as *von-Neumann architecture*.



18

18

- **Inside the Processor**
  - Also called *Central Processing Unit* (CPU).
  - Consists of a *Control Unit* and an *Arithmetic Logic Unit* (ALU).
    - All calculations happen inside the ALU.
    - The Control Unit generates sequence of control signals to carry out all operations.
  - The processor fetches an instruction from memory for execution.
    - An instruction specifies the exact operation to be carried out.
    - It also specifies the data that are to be operated on.
    - A program refers to a set of instructions that are required to carry out some specific task (e.g. sorting a set of numbers).

19

19

- **What is the role of ALU?**
  - It contains several registers, some general-purpose and some special-purpose, for temporary storage of data.
  - It contains circuitry to carry out logic operations, like AND, OR, NOT, shift, compare, etc.
  - It contains circuitry to carry out arithmetic operations like addition, subtraction, multiplication, division, etc.
  - During instruction execution, the data (operands) are brought in and stored in some registers, the desired operation carried out, and the result stored back in some register or memory.

20

20

- **What is the role of control unit?**
  - Acts as the nerve center that senses the states of various functional units and sends control signals to control their states.
  - To carry out a specific operation (say, R1 ← R2 + R3), the control unit must generate control signals in a specific sequence.
    - Enable the outputs of registers R2 and R3.
    - Select the addition operation.
    - Store the output of the adder circuit into register R1.
  - When an instruction is fetched from memory, the operation (called *opcode*) is decoded by the control unit, and the control signals issued.

21

21

- **Inside the Memory Unit**
  - Two main types of memory subsystems.
    - *Primary or Main memory*, which stores the active instructions and data for the program being executed on the processor.
    - *Secondary memory*, which is used as a backup and stores all active and inactive programs and data, typically as files.
  - The processor only has direct access to the primary memory.
  - In reality, the memory system is implemented as a hierarchy of several levels.
    - L1 cache, L2 cache, L3 cache, primary memory, secondary memory.
    - Objective is to provide faster memory access at affordable cost.

22

22

- Various different types of memory are possible.
  a) Random Access Memory (RAM), which is used for the cache and primary memory sub-systems. Read and Write access times are independent of the location being accessed.
  b) Read Only Memory (ROM), which is used as part of the primary memory to store some fixed data that cannot be changed.
  c) Magnetic Disk, which uses direction of magnetization of tiny magnetic particles on a metallic surface to store data. Access times vary depending on the location being accessed, and is used as secondary memory.
  d) Flash Memory, which is replacing magnetic disks as secondary memory devices. They are faster, but smaller in size as compared to disk.

23

23

24

24

Input Unit

- Used to feed data to the computer system from the external environment.
  - Data are transferred to the processor/memory after appropriate encoding.
- Common input devices:
  - Keyboard
  - Mouse
  - Joystick
  - Camera

25

25







26

26

Output Unit

- Used to send the result of some computation to the outside world.
- Common output devices:
  - LCD/LED screen
  - Printer and Plotter
  - Speaker / Buzzer
  - Projection system

27

27






28

28

MIPS32 Architecture: A Case Study

- As a case study, we shall be considering the MIPS32 architecture.
  - It belongs to the family of *Reduced Instruction Set Computer* (RISC).
  - Look into the *Instruction Set Architecture* (ISA) of MIPS32.
    - What CPU registers are available to the assembly language programmer?
    - What assembly language instructions are supported?
  - Required for the first set of experiments in the COA Laboratory.

29

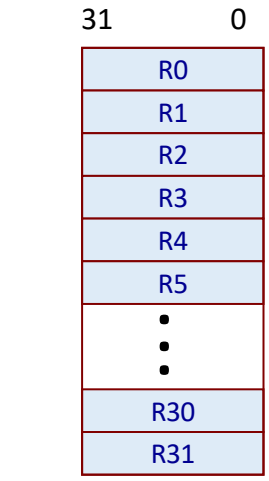
29

MIPS32 CPU Registers

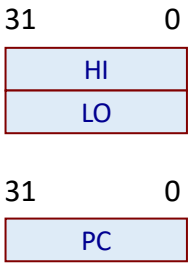
- The MIPS32 ISA has the following CPU registers that are visible to the machine/assembly language programmer.
  a) 32, 32-bit general purpose registers (GPRs), *R0* to *R31*.
  b) A special-purpose 32-bit program counter (*PC*).
     - Points to the next instruction in memory to be fetched and executed.
     - Not directly visible to the programmer.
     - Affected only indirectly by certain instructions (like branch, call, etc.)
  c) A pair of 32-bit special-purpose registers *HI* and *LO*, which are used to hold the results of multiply, divide, and multiply-accumulate instructions.

30

30



General Purpose Registers



Special Purpose Registers

Two of the GPRs have assigned functions:

a) R0 is hard-wired to a value of zero.
   - Can be used as a source when a zero value is needed.

b) R31 is used to store the return address when a function call is made.
   - Used by the jump-and-link and branch-and-link instructions like JAL, BLTZAL, BGEZAL, etc.
   - Can also be used as a normal register.

31

31

Some Examples

```
LD      R4, 50(R3)      // R4 = Mem[50+R3]
ADD     R2, R1, R4      // R2 = R1 + R4
SD      54(R3), R2      // Mem[54+R3] = R2
```

```
ADD   R2, R5, R0      // R2 = R5
```

```
MAIN:  ADDI   R1, R0, 35    // R1 = 35
       ADDI   R2, R0, 56    // R2 = 56
       JAL    GCD
       .....

GCD:   .....      // Find GCD of R1 & R2

       JR     R31
```

32

32

How are the HI and LO registers used?

- During a multiply operation, the *HI* and *LO* registers store the product of an integer multiply.
  - HI denotes the high-order 32 bits, and LO denotes the low-order 32 bits.
- During a multiply-add or multiply-subtract operation, the HI and LO registers store the result of the integer multiply-add or multiply-subtract.
- During a division, the *HI* and *LO* registers store the quotient (in LO) and remainder (in HI) of integer divide.

33

33

Some MIPS32 Assembly Language Conventions

- The integer registers of MIPS32 can be accessed as R0..R31 or r0..r31 in an assembly language program.
- Several assemblers and simulators are available in the public domain (like *QtSPIM*) that follow some specific conventions.
  - These conventions have become like a *de facto* standard when we write assembly language programs for MIPS32.
  - Basically some alternate names are used for the registers to indicate their intended usage.

34

34

Register name	Register number	Usage
$zero	R0	Constant zero

Used to represent the constant zero value, wherever required in a program.

35

35

Register name	Register number	Usage
$at	R1	Reserved for assembler

May be used as temporary register during macro expansion by assembler.

- Assembler provides an extension to the MIPS32 instruction set that are converted to standard MIPS32 instructions.

Example: Load Address instruction used to initialize pointers

```
la    R5,addr
      ⇩
lui   $at,Upper-16-bits-of-addr
ori   R5,$at,Lower-16-bits-of-addr
```

36

36

Register name	Register number	Usage
$v0	R2	Result of function, or for expression evaluation
$v1	R3	Result of function, or for expression evaluation

May be used for up to two function return values, and also as temporary registers during expression evaluation.

37

37

Register name	Register number	Usage
$a0	R4	Argument 1
$a1	R5	Argument 2
$a2	R6	Argument 3
$a3	R7	Argument 3

May be used to pass up to four arguments to functions.

38

38

Register name	Register number	Usage
$t0	R8	Temporary (not preserved across call)
$t1	R9	Temporary (not preserved across call)
$t2	R10	Temporary (not preserved across call)
$t3	R11	Temporary (not preserved across call)
$t4	R12	
$t5	R13	
$t6	R14	
$t7	R15	
$t8	R24	Temporary (not preserved across call)
$t9	R25	Temporary (not preserved across call)

May be used as temporary variables in programs.

These registers might get modified when some functions are called (other than user-written functions).

39

39

Register name	Register number	Usage
$s0	R16	Temporary (preserved across call)
$s1	R17	Temporary (preserved across call)
$s2	R18	Temporary (preserved across call)
$s3	R19	Temporary (preserved across call)
$s4	R20	Temporary (preserved across call)
$s5	R21	Temporary (preserved across call)
$s6	R22	
$s7	R23	

May be used as temporary variables in programs.

These registers do not get modified across function calls.

40

40

Register name	Register number	Usage
$gp	R28	Pointer to global area
$sp	R29	Stack pointer
$fp	R30	Frame pointer
$ra	R31	Return address (used by function call)

These registers are used for a variety of pointers:

- Global area: points to the memory address from where the global variables are allocated space.
- Stack pointer: points to the top of the stack in memory.
- Frame pointer: points to the activation record in stack.
- Return address: used while returning from a function.

41

41

Register name	Register number	Usage
$k0	R26	Reserved for OS kernel
$k1	R27	Reserved for OS kernel

These registers are supposed to be used by the OS kernel in a real computer system.

It is highly recommended not to use these registers.

42

42

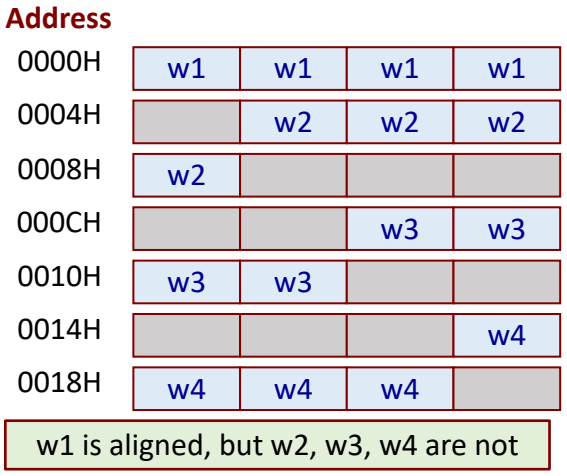
MIPS32 INSTRUCTION SET

Instruction Set Classification

- MIPS32 instruction can be classified into the following functional groups:
  a) Load and Store
  b) Arithmetic and Logical
  c) Jump and Branch
  d) Miscellaneous
  e) Coprocessor instruction (to activate an auxiliary processor).
- All instructions are encoded in 32 bits.

Alignment of Words in Memory

- MIPS requires that all words must be aligned in memory to word boundaries.
  - Must start from an address that is some power of 4.
  - Last two bits of the address must be 00.
- Allows a word to be fetched in a single cycle.
  - Misaligned words may require two cycles.



45

45

(a) Load and Store Instructions

- MIPS32 is a *load-store architecture*.
  - All operations are performed on operands held in processor registers.
  - Main memory is accessed only through *LOAD* and *STORE* instructions.
- There are various types of LOAD and STORE instructions, each used for a particular purpose.
  a) By specifying the size of the operand (W: word, H: half-word, B: byte)
     - Examples: LW, LH, LB, SW, SW, SB
  b) By specifying whether the operand is signed (by default) or unsigned.
     - Examples: LHU, LBU

46

46

c) Accessing fields that are not word aligned.
   - Examples: LWL, LWR, SWL, SWR

d) Atomic memory update for read-modify-write instructions
   - Examples: LL, SC

47

47

Data sizes that can be accessed through LOAD and STORE

Data Size	Load Signed	Load Unsigned	Store
Byte	YES	YES	YES
Half-word	YES	YES	YES
Word	YES	Only for MIPS64	YES
Unaligned word	YES		YES
Linked word (atomic modify)	YES		YES

48

48

Type	Mnemonic	Function
Aligned	LB	Load Byte
	LBU	Load Byte Unsigned
	LH	Load Half-word
	LHU	Load Half-word Unsigned
	LW	Load Word
	SB	Store Byte
	SH	Store Half-word
	SW	Store Word

Type	Mnemonic	Function
Unaligned	LWL	Load Word Left
	LWR	Load Word Right
	SWL	Store Word Left
	SWR	Store Word Right
Atomic Update	LL	Load Linked Word
	SB	Store Conditional Word

49

49

(b) Arithmetic and Logic Instructions

- All arithmetic and logic instructions operate on registers.
- Can be broadly classified into the following categories:
  - ALU immediate
  - ALU 3-operand
  - ALU 2-operand
  - Shift
  - Multiply and Divide

50

50

Type	Mnemonic	Function
16-bit Immediate Operand	ADDI	Add Immediate Word
	ADDIU	Add Immediate Unsigned Word
	ANDI	AND Immediate
	LUI	Load Upper Immediate
	ORI	OR Immediate
	SLTI	Set on Less Than Immediate
	SLTIU	Set on Less Than Immediate Unsigned
	XORI	Exclusive-OR Immediate

51

51

Type	Mnemonic	Function
3-Operand	ADD	Add Word
	ADDU	Add Unsigned Word
	AND	Logical AND
	NOR	Logical NOR
	SLT	Set on Less Than
	SLTU	Set on Less Than Unsigned
	SUB	Subtract Word
	SUBU	Subtract Unsigned Word
	XOR	Logical XOR

52

52

Type	Mnemonic	Function
Shift	ROTR	Rotate Word Right
	ROTRV	Rotate Word Right Value (Register)
	SLL	Shift Word Left Logical
	SLLV	Shift Word Left Logical Value (Register)
	SRA	Shift Word Right Arithmetic
	SRAV	Shift Word Right Arithmetic Value (Register)
	SRL	Shift Word Right Logical
	SRLV	Shift Word Right Logical Value (Register)

53

53

(c) Multiply and Divide Instructions

- The multiply and divide instructions produce twice as many result bits.
  - When two 32-bit numbers are multiplied, we get a 64-bit product.
  - After division, we get a 32-bit quotient and a 32-bit remainder.
- Results are produced in the HI and LO register pair.
  - For multiplication, the low half of the product is loaded into LO, while the higher half in HI.
  - Multiply-Add and Multiply-Subtract produce a 64-bit product, and adds or subtracts the product from the concatenated value of HI and LO.
  - Divide produces a quotient that is loaded into LO and a remainder that is loaded into HI.

54

54

- Only exception is the MUL instruction, which delivers the lower half of the result directly to a GPR.
  - Useful is situations where the product is expected to fit in 32 bits.

55

55

Type	Mnemonic	Function
Multiply and Divide	DIV	Divide Word
	DIVU	Divide Unsigned Word
	MADD	Multiply and Add Word
	MADDU	Multiply and Add Word Unsigned
	MFHI	Move from HI
	MFLO	Move from LO
	MSUB	Multiply and Subtract Word
	MSUBU	Multiply and Subtract Word Unsigned
	MTHI	Move to HI
	MTLO	Move to LO
	MUL	Multiply Word to Register
	MULT	Multiply Word
	MULTU	Multiply Unsigned Word

56

56

(d) Jump and Branch Instructions

- The following types of Jump and Branch instructions are supported by MIPS32.
  - PC relative conditional branch
    - A 16-bit offset is added to PC.
  - PC-relative unconditional jump
    - A 28-bit offset if added to PC.
  - Absolute (register) unconditional jump
  - Special Jump instructions that link the return address in R31.

57

57

Type	Mnemonic	Function
Unconditional Jump within a 256 MB Region	J	Jump
	JAL	Jump and Link
	JALX	Jump and Link Exchange

Type	Mnemonic	Function
Unconditional Jump using Absolute Address	JALR	Jump and Link Register
	JR	Jump Register

58

58

Type	Mnemonic	Function
PC-Relative Conditional Branch Comparing Two Registers	BEQ	Branch on Equal
	BNE	Branch on Not Equal

Type	Mnemonic	Function
PC-Relative Conditional Branch Comparing With Zero	BGEZ	Branch on Greater Than or Equal to Zero
	BGEZAL	Branch on Greater Than or Equal to Zero and Link
	BGTZ	Branch on Greater than Zero
	BLEZ	Branch on Less Than or Equal to Zero

59

59

(e) Miscellaneous Instructions

- These instructions are used for various specific machine control purposes.
- They include:
  - Exception instructions
  - Conditional MOVE instructions
  - Prefetch instructions
  - NOP instructions

60

60

Type	Mnemonic	Function
System Call and Breakpoint	BREAK	
	SYSCALL	
Trap-on-Condition Comparing Two Registers	TEQ	
	TGE	
	TGEU	
	TLT	
	TLTU	Trap if Less Than Unsigned
	TNE	Trap if Not Equal

Type	Mnemonic	Function
Trap-on-Condition Comparing an Immediate Value	TEQI	Trap if Equal Immediate
	TGEI	Trap if Greater Than or Equal Immediate
	TGEIU	Trap if Greater Than or Equal Immediate Unsigned
	TLTI	Trap if Less Than Immediate
	TLTIU	Trap if Less Than Immediate Unsigned
	TNEI	Trap if Not Equal Immediate

61

61

Type	Mnemonic	Function
Conditional Move	MOVF	Move Conditional on Floating Point False
	MOVN	Move Conditional on Not Zero
	MOVT	Move Conditional on Floating Point True
	MOVZ	Move Conditional on Zero

Type	Mnemonic	Function
Prefetch	PREF	Prefetch Register+Offset
NOP	NOP	No Operation

62

62

(e) Coprocessor Instructions

- The MIPS architecture defines four coprocessors (designated CP0, CP1, CP2, and CP3).
  - Coprocessor 0 (CP0) is incorporated on the CPU chip and supports the virtual memory system and exception handling. CP0 is also referred to as the System Control Coprocessor.
  - Coprocessor 1 (CP1) is reserved for the floating point coprocessor.
  - Coprocessor 2 (CP2) is available for specific implementations.
  - Coprocessor 3 (CP3) is available for future extensions.
- These instructions are not discussed here.

63

63

- MIPS32 architecture also supports a set of floating-point registers and floating-point instructions.
  - Shall be discussed later.

64

64

MIPS PROGRAMMING EXAMPLES

65

65

Some Examples of MIPS32 Arithmetic

C Code

```
A = B + C;
```

MIPS32 Code

```
add     $s1, $s2, $s3
```

B loaded in $s2
C loaded in $s3
A ← $s1

C Code

```
A = B + C – D;
E = F + A;
```

MIPS32 Code

```
add     $t0, $s1, $s2
sub     $s0, $t0, $s3
add     $s4, $s5, $s0;
```

B loaded in $s1
C loaded in $s2
D loaded in $s3
F loaded in $s5
$t0 is a temporary
A ← $s0; E ← $s4

66

66

Example on LOAD and STORE

C Code

```
A[10] = X – A[12];
```

MIPS32 Code

```
lw      $t0, 48($s3)
sub     $t0, $s2, $t0
sw      $t0, 40($s3);
```

$s3 contains the starting address of the array A

$s2 loaded with X

$t0 is a temporary

Address of A[10] will be $s3+40 (4 bytes per element)

Address of A[12] will be $s3+48

67

67

Examples on Control Constructs

C Code

```
if (x==y)  z = x – y;
```

MIPS32 Code

```
        bne     $s0, $s1, Label
        sub     $s3, $s0, $s1
Label:  ……
```

$s0 loaded with x

$s1 loaded with y

z ← $s3

68

68

C Code

```
if (x != y)  z = x – y;
else         z = x + y;
```

MIPS32 Code

```
        beq     $s0, $s1, Lab1
        sub     $s3, $s0, $s1
        j       Lab2
Lab1:   add     $s3, $s0, $s1
Lab2:   ….
```

$s0 loaded with x

$s1 loaded with y

z ← $s3

69

69

- MIPS32 supports a limited set of conditional branch instructions:

```
beq   $s2,Label      // Branch to Label of $s2 = 0
bne   $s2,Label      // Branch to Label of $s2 != 0
```

- Suppose we need to implement a conditional branch after comparing two registers for less-than or greater than.

C Code

```
if (x < y)   z = x – y;
else         z = x + y;
```

MIPS32 Code

```
        slt     $t0,$s0,$s1
        beq     $t0, $zero, Lab1
        sub     $s3, $s0, $s1
        j       Lab2
Lab1:   add     $s3, $s0, $s1
Lab2:   ….
```

Set if less than. If $s0 < $s1, then set $t0=1; else $t0=0.

70

70

- MIPS32 assemblers supports several pseudo-instructions that are meant for user convenience.
  - Internally the assembler converts them to valid MIPS32 instructions.
- Example: The pseudo-instruction branch if less than

  blt $s1, $s2, Label

MIPS32 Code

```
        slt     $at, $s1, $s2
        bne     $t0, $zero, Label
        ....
Label:  ....
```

The assembler requires an extra register to do this.

The register $at (= R1) is reserved for this purpose.

71

71

Working with Immediate Values in Registers

- **Case 1**: Small constants, which can be specified in 16 bits.
  - Occurs most frequently (about 90% of the time).
  - Examples:

```
A = A + 16;   →   addi   $s1, $s1, 16       (A in $s1)
X = Y – 1025; →   subi   $s1, $s2, 1025     (X in $s1, Y in $s2)
A = 100;      →   addi   $s1, $zero, 100    (A in $s1)
```

72

72

- **Case 2**: Large constants, that require 32 bits to represent.
  - How to load a large constant in a register?
  - Requires two instructions.
    - A "*Load Upper Immediate*" instruction, that loads a 16-bit number into the upper half of a register (lower bits filled with zeros).
    - An "*OR Immediate*" instruction, to insert the lower 16-bits.
  - Suppose we want to load 0xAAAA3333 into a register $s1.

lui $s1, 0xAAAA	1010101010101010 0000000000000000
ori $s1, $s1, 0x3333	1010101010101010 0011001100110011

73

73

Other MIPS Pseudo-instructions

Pseudo-Instruction	Translates to	Function
blt $1, $2, Label	slt $at, $1, $2 bne $at, $zero, Label	Branch if less than
bgt $1, $2, Label	sgt $at, $1, $2 bne $at, $zero, Label	Branch if greater than
ble $1, $2, Label	sle $at, $1, $2 bne $at, $zero, Label	Branch if less or equal
bge $1, $2, Label	sge $at, $1, $2 bne $at, $zero, Label	Branch if greater or equal
li $1, 0x23ABCD	lui $1, 0x0023 ori $1, $1, 0xABCD	Load immediate value into a register

74

74

Pseudo-Instruction	Translates to	Function
move $1, $2	add $1, $2, $zero	Move content of one register to another
la $a0, 0x2B09D5	lui $a0, 0x002B ori $a0, $a0, l0x09D5	Load address into a register

75

75

A Simple Function

C Function

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

MIPS32 Code

```
swap:  muli   $t0, $s0, 4
       add    $t0, $s1, $t0
       lw     $t1, 0($t0)
       lw     $t2, 4($t0)
       sw     $t2, 0($t0)
       sw     $t1, 4($t0)
       jr     $ra
```

$s0 loaded with index k

$s1 loaded with base address of A

Address of A[k] = $s1 + 4 * $s0

Exchange A[k] and A[k+1]

76

76

SPIM System Calls

```
li       $v0,1            # print an integer in $a0
li       $a0,100
syscall

li       $v0,5            # read an integer into $v0
syscall

li       $v0,11           # print a character in $a0
li       $a0,'a'
syscall

li       $v0,12           # read a character into $v0
syscall

li       $v0,4            # print an ASCIIZ string at $a0
la       $a0,msg_hello
syscall
```

77

77

A complete program (check for perfect number)

```
## Takes in a positive integer - num and prints out if the number is perfect(or not)
## Registers Used :
##                  $t0 for storing num
##                  $t1 for storing k
##                  $t2 for storing sum
##                  $t3 for storing num % k

.text

main:
        la      $a0 , num_msg       # loads address of entering number msg in $a0
        li      $v0 , 4             # loads print string syscall command
        syscall                     # makes a system call

        li      $v0 , 5             # loads read int syscall command
        syscall                     # makes a system call
        move    $t0 , $v0           # store the value in $t0

        li      $t1 , 1             # initialize k with 1
        li      $t2 , 0             # initialize sum with 0

loop:
        bge     $t1 , $t0 , endloop # if k >= num then we are done
        rem     $t3 , $t0 , $t1     # load the remainder , num % k into $t3
        beqz    $t3 , adder         # branch to adder
```

78

78

```
loop1:
        add     $t1 , $t1 , 1           # increment the value of k by 1, i.e k++
        b       loop                    # unconditional branching to loop

adder:
        add     $t2 , $t2 , $t1         # add the value of k to the sum
        b       loop1                   # unconditional branching to lab

endloop:
        beq     $t2 , $t0 , perfect     # after the loop, if the value of sum = number entered, the number is a perfect number
        bne     $t2 , $t0 , notperfect  # if the value of sum is not equal to the number , the number is not a perfect number

perfect:
        la      $a0 , success_msg       # loads address of success msg in $a0
        li      $v0 , 4                 # loads print string syscall command
        syscall                         # makes a system call
        b       exit                    # unconditional branching to exit label

notperfect:
        la      $a0 , failure_msg       # loads address of failure msg in $a0
        li      $v0 , 4                 # loads print string syscall command
        syscall                         # makes a system call
        b       exit                    # unconditional branching to exit label

exit:                                   #### exit the program ####
        li      $v0 , 10                # load exit command into $v0
        syscall                         # makes a system call

.data                                   ##### data for storing strings ######
num_msg:    .asciiz "Enter a positive Integer\n"
success_msg: .asciiz "Entered number is a perfect number\n"
failure_msg: .asciiz "Entered number is not a perfect number\n"
```

79

79