NPTEL ONLINE CERTIFICATION COURSES

NPTEL

IIT KHARAGPUR | NIT MEGHALAYA

Lecture 6: NUMBER REPRESENTATION

DR. KAMALIKA DATTA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING, NIT MEGHALAYA

Number System: The Basics

- We are accustomed to the so-called *decimal number system*.
  - Ten digits :: 0,1,2,3,4,5,6,7,8,9
  - Every digit position has a weight which is a power of 10.
  - *Base* or *radix* is 10.
- Examples:

  234 $= 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$

  250.67 $= 2 \times 10^2 + 5 \times 10^1 + 0 \times 10^0 + 6 \times 10^{-1} + 7 \times 10^{-2}$

Binary Number System

- Two digits: 0 and 1.
  - Every digit position has a weight that is a power of 2.
  - *Base* or *radix* is 2.
- Examples:

  110 $= 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$

  101.01 $= 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2}$

Binary to Decimal Conversion

- Each digit position of a binary number has a weight.
  - Some power of 2.
- A binary number:

  $B = b_{n-1}\, b_{n-2} \ldots b_1\, b_0 \,.\, b_{-1}\, b_{-2} \ldots b_{-m}$

  where b_i are the binary digits.

Corresponding value in decimal:

$$D = \sum_{i=-m}^{n-1} b_i\, 2^i$$

Some Examples

1. 101011 ➔ $1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 43$

   $(101011)_2 = (43)_{10}$

2. .0101 ➔ $0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} = .3125$

   $(.0101)_2 = (.3125)_{10}$

3. 101.11 ➔ $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 5.75$

   $(101.11)_2 = (5.75)_{10}$

Decimal to Binary Conversion

- Consider the integer and fractional parts separately.
- For the integer part:
  - Repeatedly divide the given number by 2, and go on accumulating the remainders, until the number becomes zero.
  - Arrange the remainders *in reverse order*.
- For the fractional part:
  - Repeatedly multiply the given fraction by 2.
    - Accumulate the integer part (0 or 1).
    - If the integer part is 1, chop it off.
  - Arrange the integer parts *in the order* they are obtained.

Examples

```
2 | 239
2 | 119  --- 1
2 |  59  --- 1
2 |  29  --- 1
2 |  14  --- 1
2 |   7  --- 0
2 |   3  --- 1
2 |   1  --- 1
2 |   0  --- 1
```

$(239)_{10} = (11101111)_2$

```
2 | 64
2 | 32  --- 0
2 | 16  --- 0
2 |  8  --- 0
2 |  4  --- 0
2 |  2  --- 0
2 |  1  --- 0
2 |  0  --- 1
```

$(64)_{10} = (1000000)_2$

.634 x 2 = 1.268
.268 x 2 = 0.536
.536 x 2 = 1.072
.072 x 2 = 0.144
.144 x 2 = 0.288
:

$(.634)_{10} = (.10100......)_2$

37.0625

$(37)_{10} = (100101)_2$

$(.0625)_{10} = (.0001)_2$

$\therefore (37.0625)_{10} = (100101\ .\ 0001)_2$

Hexadecimal Number System

- A compact way to represent binary numbers.
  - Group of four binary digits are represented by a hexadecimal digit.
  - Hexadecimal digits are 0 to 9, A to F.

Hex	Binary	Hex	Binary
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Binary to Hexadecimal Conversion

- For the integer part:
  - Scan the binary number from *right to left*.
  - Translate each group of four bits into the corresponding hexadecimal digit.
    - Add *leading* zeros if necessary.
- For the fractional part:
  - Scan the binary number from *left to right*.
  - Translate each group of four bits into the corresponding hexadecimal digit.
    - Add *trailing* zeros if necessary.

Examples

1. $(\underline{1011}\ \underline{0100}\ \underline{0011})_2 = (B43)_{16}$
2. $(\underline{10}\ \underline{1010}\ \underline{0001})_2 = (2A1)_{16}$ — *Two leading 0s are added*
3. $(.\underline{1000}\ \underline{010})_2 = (.84)_{16}$ — *A trailing 0 is added*
4. $(\underline{101}\ .\ \underline{0101}\ \underline{111})_2 = (5.5E)_{16}$ — *A leading 0 and trailing 0 are added*

Hexadecimal to Binary Conversion

- Translate every hexadecimal digit into its 4-bit binary equivalent.
- Examples:

$(3A5)_{16} = (0011\ 1010\ 0101)_2$

$(12.3D)_{16} = (0001\ 0010\ .\ 0011\ 1101)_2$

$(1.8)_{16} = (0001\ .\ 1000)_2$

How are Hexadecimal Numbers Written?

- Using the suffix “H” or using the prefix “0x”.
- Examples:
  - ADDI R1,2AH — // Add the hex number 2A to register R1
  - 0x2AB4 — // The 16-bit number 0010 1010 1011 0100
  - 0xFFFFFFFF — // The 32-bit number for the all-1 string

Unsigned Binary Numbers

- An *n*-bit binary number can have 2^n distinct combinations.
  - For example, for n=3, the 8 distinct combinations are:
    000, 001, 010, 011, 100, 101, 110, 111 (0 to 2^3-1 = 7 in decimal).

Number of bits (n)	Range of Numbers
8	0 to 2^8-1 (255)
16	0 to 2^{16}-1 (65535)
32	0 to 2^{32}-1 (4294967295)
64	0 to 2^{64}-1

- An *n*-bit binary integer:

  $b_{n-1}b_{n-2} \dots b_2b_1b_0$

- Equivalent unsigned decimal value:

  $D = b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \dots + b_2 2^2 + b_1 2^1 + b_0 2^0$

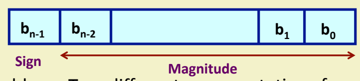
- Each digit position has a weight that is some power of 2.

Signed Integer Representation

- Many of the numerical data items that are used in a program are signed (positive or negative).
  - Question:: *How to represent sign?*
- Three possible approaches:
  a) Sign-magnitude representation
  b) One's complement representation
  c) Two's complement representation

(a) Sign-magnitude Representation

- For an n-bit number representation:
  - The most significant bit (MSB) indicates sign (0: positive, 1: negative).
  - The remaining (n-1) bits represent the magnitude of the number.
- Range of numbers: $-(2^{n-1} - 1)$ to $+(2^{n-1} - 1)$



- A problem: Two different representations for zero.
  +0: 0 00..000 and -0: 1 00..000

(b) Ones Complement Representation

- Basic idea:
  - Positive numbers are represented exactly as in sign-magnitude form.
  - Negative numbers are represented in 1's complement form.
- How to compute the 1's complement of a number?
  - Complement every bit of the number (1→0 and 0→1).
  - MSB will indicate the sign of the number (0: positive, 1: negative).

Example for n=4

Decimal	1's complement	Decimal	1's complement
+0	0000	-7	1000
+1	0001	-6	1001
+2	0010	-5	1010
+3	0011	-4	1011
+4	0100	-3	1100
+5	0101	-2	1101
+6	0110	-1	1110
+7	0111	-0	1111

To find the representation of, say, -4, first note that

+4 = 0100

-4 = 1's complement of 0100 = 1011

- Range of numbers that can be represented in 1's complement:
  - Maximum :: *+ $(2^{n-1} - 1)$*
  - Minimum :: *- $(2^{n-1} - 1)$*
- A problem:
  - Two different representations of zero.
    - +0 → 0 000....0
    - -0 → 1 111....1
- Advantage of 1's complement representation:
  - Subtraction can be done using addition.
  - Leads to substantial saving in circuitry.

(c) Twos Complement Representation

- Basic idea:
  - Positive numbers are represented exactly as in sign-magnitude form.
  - Negative numbers are represented in 2's complement form.
- How to compute the 2's complement of a number?
  - Complement every bit of the number (1→0 and 0→1), and then *add one* to the resulting number.
  - MSB will indicate the sign of the number (0: positive, 1: negative).

Example for n=4

Decimal	2's complement	Decimal	2's complement
+0	0000	-8	1000
+1	0001	-7	1001
+2	0010	-6	1010
+3	0011	-5	1011
+4	0100	-4	1100
+5	0101	-3	1101
+6	0110	-2	1110
+7	0111	-1	1111

To find the representation of, say, -4, first note that

+4 = 0100

-4 = 2's complement of 0100 = 1011 + 1 = 1100

- Range of numbers that can be represented in 2's complement:
  - Maximum :: *+ $(2^{n-1} - 1)$*
  - Minimum :: *- 2^{n-1}*
- Advantage of 2's complement representation:
  - Unique representation of zero.
  - Subtraction can be done using addition.
  - Leads to substantial saving in circuitry.
- Almost all computers today use 2's complement representation for storing negative numbers.

- Some other features of 2's complement representation
  a) Weighted number representation, with the MSB having weight -2^{n-1}.

-2^{n-1}	2^{n-2}		2^1	2^0
b_{n-1}	b_{n-2}		b_1	b_0

$$D = -b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \ldots + b_2 2^2 + b_1 2^1 + b_0 2^0$$

  b) Shift left by *k* positions with zero padding multiplies the number by 2^k.

  00010011 = +19 :: Shift left by 2 :: 01001100 = +76

  11100011 = -29 :: Shift left by 2 :: 10001100 = -116

c) Shift right by *k* positions with sign bit padding divides the number by 2^k.

00010110 = +22 :: Shift right by 2 :: 00000101 = +5

11100100 = -28 :: Shift right by 2 :: 11111001 = -7

d) The sign bit can be copied as many times as required in the beginning to extend the size of the number (called *sign extension*).

X = 00101111 (8-bit number, value = +47)
Sign extend to 32 bits:
00000000 00000000 00000000 00101111

X = 10100011 (8-bit number, value = -93)
Sign extend to 32 bits:
11111111 11111111 11111111 10100011

END OF LECTURE 6

Lecture 7: INSTRUCTION FORMAT AND ADDRESSING MODES

DR. KAMALIKA DATTA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING, NIT MEGHALAYA

Instruction Format

- An instruction consists of two parts:-
  a) Operation Code or Opcode
    - Specifies the operation to be performed by the instruction.
    - Various categories of instructions: data transfer, arithmetic and logical, control, I/O and special machine control.
  b) Operand(s)
    - Specifies the source(s) and destination of the operation.
    - Source operand can be specified by an immediate data, by naming a register, or specifying the address of memory.
    - Destination can be specified by a register or memory address.

- Number of operands varies from instruction to instruction.
- Also for specifying an operand, various *addressing modes* are possible:
  – Immediate addressing
  – Direct addressing
  – Indirect addressing
  – Relative addressing
  – Indexed addressing, and many more.

Instruction Format Examples

Format	Example
opcode |	Implied addressing: NOP, HALT
opcode | memory address	1-address: ADD X, LOAD M
opcode | memory address | memory address	2-address: ADD X,Y
opcode | register | memory address	Register-memory: ADD R1,X
opcode | register | register | register	Register-register: ADD R1,R2,R3

A 32-bit Instruction Example

- Suppose we have an ISA with 32-bit instructions only.
  – Fixed size instructions make the decoding easier.
- Some instruction encoding examples are shown.
  – Assume that there are 32 registers R0 to R31, all of 32-bits.
  – 5-bits are required to specify a register.

31–26	25–21	20–16	15–0
opcode	dest	source	16-bit immediate data
LOAD	01011	00010	0000000001100100

LOAD R11,100(R2)
:: R11 = Mem(R2+100)

31–26	25–21	20–16	15–11	10–0
opcode	dest	source	source	ALU function
ALU op	00010	00101	01000	ADD

ADD R2,R5,R8
:: R2 = R5 + R8

ADDRESSING MODES

What are Addressing Modes?

- They specify the mechanism by which the operand data can be located.
- Some ISA's are quite complex and supports many addressing modes.
- ISA's based on load-store architecture are usually simple and support very limited number of addressing modes.
- Various addressing modes exist:
  - Immediate, Direct, Indirect, Register, Register Indirect, Indexed, Stack, Relative, Autoincrement, Autodecrement, Based, etc.
  - Not all processors support all addressing modes.
  - We shall briefly look at the common addressing modes and how they work.

Immediate Addressing

- The operand is part of the instruction itself.
  - No memory reference is required to access the operand.
  - Fast but limited range (because a limited number of bits are provided to specify the immediate data).
- Examples:
  - ADD #25 // ACC = ACC + 25
  - ADDI R1,R2,42 // R1 = R2 + 42

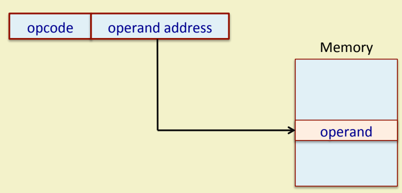
opcode	immediate data

Direct Addressing

- The instruction contains a field that holds the memory address of the operand.

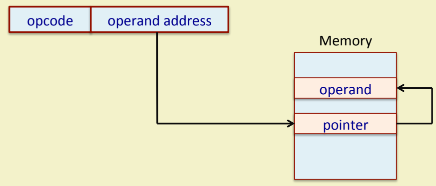
opcode	operand address

- Examples:
  - ADD R1,20A6H // R1 = R1 + Mem[20A6]
- Single memory access is required to access the operand.
  - No additional calculations required to determine the operand address.
  - Limited address space (as number of bits is limited, say, 16 bits).



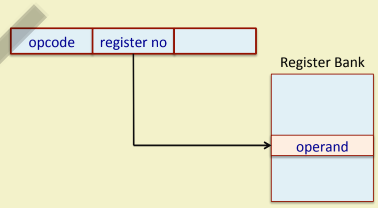
Indirect Addressing

- The instruction contains a field that holds the memory address, which in turn holds the memory address of the operand.
- Two memory accesses are required to get the operand value.
- Slower but can access large address space.
  - Not limited by the number of bits in operand address like direct addressing.
- Examples:
  - ADD R1,(20A6H) // R1 = R1 + (Mem[20A6])



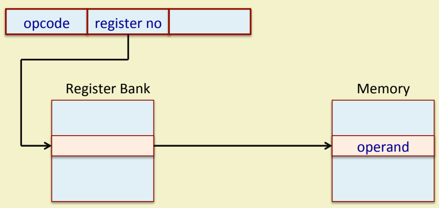
Register Addressing

- The operand is held in a register, and the instruction specifies the register number.
  - Very few number of bits needed, as the number of registers is limited.
  - Faster execution, since no memory access is required for getting the operand.
- Modern load-store architectures support large number of registers.
- Examples:
  - ADD R1,R2,R3 // R1 = R2 + R3
  - MOV R2,R5 // R2 = R5



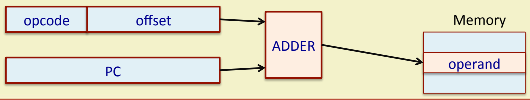
Register Indirect Addressing

- The instruction specifies a register, and the register holds the memory address where the operand is stored.
  - Can access large address space.
  - One fewer memory access as compared to indirect addressing.
- Example:
  - ADD R1,(R5) // PC = R1 + Mem[R5]



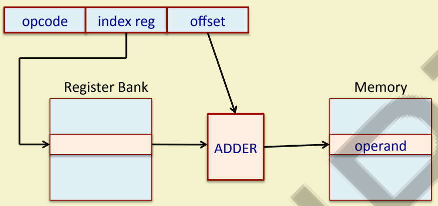
Relative Addressing (PC Relative)

- The instruction specifies an offset of displacement, which is added to the program counter (PC) to get the effective address of the operand.
  - Since the number of bits to specify the offset is limited, the range of relative addressing is also limited.
  - If a 12-bit offset is specified, it can have values ranging from -2048 to +2047.



Indexed Addressing

- Either a special-purpose register, or a general-purpose register, is used as *index register* in this addressing mode.
- The instruction specifies an offset of displacement, which is added to the index register to get the effective address of the operand.
- Example:
  - LOAD R1,1050(R3) // R1 = Mem[1050+R3]
- Can be used to sequentially access the elements of an array.
  - Offset gives the starting address of the array, and the index register value specifies the array element to be used.



Stack Addressing

- Operand is implicitly on top of the stack.
- Used in zero-address machines earlier.
- Examples:
  - ADD
  - PUSH X
  - POP X
- Many processors have a special register called the stack pointer (SP) that keeps track of the stack-top in memory.
  - PUSH, POP, CALL, RET instructions automatically modify SP.

Some Other Addressing Modes

- Base addressing
  - The processor has a special register called the *base register* or *segment register*.
  - All operand addresses generated are added to the base register to get the final memory address.
  - Allows easy movement of code and data in memory.
- Autoincrement and Autodecrement
  - First introduced in the PDP-11 computer system.
  - The register holding the operand address is automatically incremented or decremented after accessing the operand (like a++ and a-- in C).

END OF LECTURE 7

NPTEL ONLINE CERTIFICATION COURSES

IIT KHARAGPUR | NIT MEGHALAYA

Lecture 8: CISC AND RISC ARCHITECTURE

DR. KAMALIKA DATTA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING, NIT MEGHALAYA

Broad Classification

- Computer architectures have evolved over the years.
  - Features that were developed for mainframes and supercomputers in the 1960s and 1970s have started to appear on a regular basis on later generation microprocessors.
- Two broad classifications of ISA:
  a) Complex Instruction Set Computer (CISC)
  b) Reduced Instruction Set Computer (RISC)

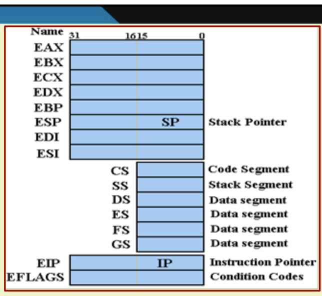
CISC versus RISC Architectures

- Complex Instruction Set Computer (CISC)
  - More traditional approach.
  - Main features:
    - Complex instruction set
    - Large number of addressing modes (R-R, R-M, M-M, indexed, indirect, etc.)
    - Special-purpose registers and Flags (sign, zero, carry, overflow, etc.)
    - Variable-length instructions / Complex instruction encoding
    - Ease of mapping high-level language statements to machine instructions
    - Instruction decoding / control unit design more complex
    - Pipeline implementation quite complex

- CISC Examples:
  - IBM 360/370 (1960-70)
  - VAX-11/780 (1970-80)
  - Intel x86 / Pentium (1985-present)

Only CISC instruction set that survived over generations.
- Desktop PC's / Laptops use these.
- The volume of chips manufactured is so high that there is enough motivation to pay the extra design cost.
- Sufficient hardware resources available today to translate from CISC to RISC internally.

Register Set in Pentium



Addressing Modes in VAX

Addressing Mode	Example	Micro-operation
Register direct	ADD R1,R2	R1 = R1 + R2
Immediate	ADD R1,#15	R1 = R1 + 15
Displacement	ADD R1,220(R5)	R1 = R1 + Mem[220+R5]
Register indirect	ADD R1,(R3)	R1 = R1 + Mem[R3]
Indexed	ADD R1,(R2+R3)	R1 = R1 + Mem[R2+R3]
Direct	ADD R1, (1000)	R1 = R1 + Mem[1000]
Memory indirect	ADD R1,@(R4)	R1 = R1 + Mem[Mem[R4]]
Autoincrement	ADD R1,(R2)+	R1 = R1 + Mem[R2]; R2++
Autodecrement	ADD R1,(R2)-	R1 = R1 + Mem[R2]; R2--
Scaled	ADD R1,50(R2)[R3]	R1 = R1 + Mem[50+R2+R3*d]

- Reduced Instruction Set Computer (RISC)
  - Very widely used among many manufacturers today.
  - Also referred to as *Load-Store Architecture*.
    - Only LOAD and STORE instructions access memory.
    - All other instructions operate on processor registers.
  - Main features:
    - Simple architecture for the sake of efficient pipelining.
    - Simple instruction set with very few addressing modes.
    - Large number of general-purpose registers; very few special-purpose.
    - Instruction length and encoding uniform for easy instruction decoding.
    - Compiler assisted scheduling of pipeline for improved performance.

- RISC Examples:
  - CDC 6600 (1964)
  - MIPS family (1980-90)
  - SPARC
  - ARM microcontroller family

- Almost all the computers today use a RISC based pipeline for efficient implementation.
  - RISC based computers use compilers to translate into RISC instructions.
  - CISC based computers (e.g. x86) use hardware to translate into RISC instructions.

Results of a Comparative Study

- A quantitative comparison of VAX 8700 (a CISC machine) and MIPS M2000 (a RISC machine) with comparable organizations was carried out in 1991.
- Some findings:
  - MIPS required execution of about twice the number of instructions as compared to VAX.
  - Cycles Per Instructions (CPI) for VAX was about six times larger than that of MIPS.
  - Hence, MIPS had three times the performance of VAX.
  - Also, much less hardware is required to build MIPS as compared to VAX.

- **Conclusion**:
  - Persisting with CISC architecture is too costly, both in terms of hardware cost and also performance.
  - VAX was replaced by ALPHA (a RISC processor) by Digital Equipment Corporation (DEC).
  - CISC architecture based on x86 is different.
    - Because of huge number of installed base, backward compatibility of machine code is very important from commercial point of view.
    - They have adopted a balanced view: (a) user's view is a CISC instruction set, (b) hardware translates every CISC instruction into an equivalent set of RISC instructions internally, (c) an instruction pipeline executes the RISC instructions efficiently.

MIPS32 Architecture: A Case Study

- As a case study of RISC ISA, we shall be considering the MIPS32 architecture.
  - Look into the instruction set and instruction encoding in detail.
  - Design the data path of the MIPS32 architecture, and also look into the control unit design issues.
  - Extend the basic data path of MIPS32 to a pipeline architecture, and discuss some of the issues therein.

MIPS32 CPU Registers

- The MIPS32 ISA defines 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.

- Some common registers are missing in MIPS32.
  - ***Stack Pointer*** (SP) register, which helps in maintaining a stack in main memory.
    - Any of the GPRs can be used as the stack pointer.
    - No separate PUSH, POP, CALL and RET instructions.
  - ***Index Register*** (IX), which helps in accessing memory words sequentially in memory.
    - Any of the GPRs can be used as an index register.
  - ***Flag registers*** (like ZERO, SIGN, CARRY, OVERFLOW) that keeps track of the results of arithmetic and logical operations.
    - Maintains flags in registers, to avoid problems in pipeline implementation.

31 0
R0
R1
R2
R3
R4
R5
⋮
R30
R31

General Purpose Registers

31 0
HI
LO

31 0
PC

Special Purpose Registers

Two of the GPRs have assigned functions:

a) R0 is hard-wired to a value of zero.
- Can be used as the target register for any instruction whose result is to be discarded.
- Can also 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.

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

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
```

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

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.

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

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.

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Register name	Register number	Usage
$zero	R0	Constant zero

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

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

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
```


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.

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.

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).

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.

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.

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.

END OF LECTURE 8

NPTEL ONLINE CERTIFICATION COURSES

IIT KHARAGPUR | NIT MEGHALAYA

Lecture 9: MIPS32 INSTRUCTION SET

DR. KAMALIKA DATTA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING, NIT MEGHALAYA

Instruction Set Classification

- MIPS32 instruction can be classified into the following functional groups:
  - Load and Store
  - Arithmetic and Logical
  - Jump and Branch
  - Miscellaneous
  - 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.

Address				
0000H	w1	w1	w1	w1
0004H		w2	w2	w2
0008H	w2			
000CH			w3	w3
0010H	w3	w3		
0014H				w4
0018H	w4	w4	w4	

w1 is aligned, but w2, w3, w4 are not

(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

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

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

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

(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

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

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

Type	Mnemonic	Function
2-Operand	CLO	Count Leading Ones in Word
	CLZ	Count Leading Zeros in Word

Type	Mnemonic	Function
Shift	ROTR	Rotate Word Right
	ROTRV	Rotate Word Right Variable
	SLL	Shift Word Left Logical
	SLLV	Shift Word Left Logical Variable
	SRA	Shift Word Right Arithmetic
	SRAV	Shift Word Right Arithmetic Variable
	SRL	Shift Word Right Logical
	SRLV	Shift Word Right Logical Variable

(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.
  a) For multiplication, the low half of the product is loaded into LO, while the higher half in HI.
  b) Multiply-Add and Multiply-Subtract produce a 64-bit product, and adds or subtracts the product from the concatenated value of HI and LO.
  c) Divide produces a quotient that is loaded into LO and a remainder that is loaded into HI.

- 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.

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

(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-region unconditional jump
    - A 28-bit offset if added to PC.
  - Absolute (register) unconditional jump
  - Special Jump instructions that link the return address in R31.

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
	JALRHB	Jump and Link Register with Hazard Barrier
	JR	Jump Register
	JRHB	Jump Register with Hazard Barrier

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

(e) Miscellaneous Instructions

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

Type	Mnemonic	Function
System Call and Breakpoint	BREAK	
	SYSCALL	

Type	Mnemonic	Function
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

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

(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.

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

END OF LECTURE 9

NPTEL ONLINE CERTIFICATION COURSES
NPTEL
IIT KHARAGPUR | NIT MEGHALAYA

Lecture 10: MIPS PROGRAMMING EXAMPLES

DR. KAMALIKA DATTA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING, NIT MEGHALAYA

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

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

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

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

- 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.

- 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.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

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)
```

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

- **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

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

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

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

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
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

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

A Simple Function Call

C Function	MIPS32 Code	

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

MIPS Instruction Encoding

- All MIPS32 instructions can be classified into three groups in terms of instruction encoding.
  - R-type (Register), I-type (Immediate), and J-type (Jump).
  - In an instruction encoding, the 32 bits of the instruction are divided into several fields of fixed widths.
  - All instructions may not use all the fields.
- Since the relative positions of some of the fields are same across instructions, instruction decoding becomes very simple.

(a) R-type Instruction Encoding

- Here an instruction can use up to three register operands.
  - Two source and one destination.
- In addition, for shift instructions, the number of bits to shift can also be specified.

31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	rd	shamt	funct
6-bit opcode	*Source register 1*	*Source register 2*	*Destination register*	*Shift amount*	*Opcode extension (additional functions)*

- Examples of R-type instructions:

```
add   $s1, $s2, $s3
sub   $t1, $s3, $s4
sla   $s1, $s2, 5      // shift left $s2 by 5 places, and store in $s1
```

- An example instruction encoding: *add \$t1, \$s1, \$s2*
  - Recall: \$t1 is R9, \$s1 is R17, and \$s2 is R18.
  - For "add", opcode = 000000, and funct = 100000,

31 26	25 21	20 16	15 11	10 6	5 0
000000	10001	10010	01001	00000	100000

(b) I-type Instruction Encoding

- Contains a 16-bit immediate data field.
- Supports one source and one destination register.

31 26	25 21	20 16	15 0
opcode	rs	rt	Immediate Data
6-bit opcode	*Source register 1*	*Destination register*	*16-bit immediate data*

- Examples of I-type instructions:

```
lw     $s1, 50($s5)
sw     $t1, 100($s1)
addi   $t0, $s1, 188
beq    $s1, $s2, Label     // Label is encoded as a 16-bit offset relative to PC
bne    $s3, $zero, Label
```

- An example instruction encoding: *lw \$t1, 48(\$s1)*
  - Recall: \$t1 is R9, \$s1 is R17.
  - For "lw", opcode = 100011, and funct = 100000,

31 26	25 21	20 16	15 0
100011	10001	01001	0000000000110000

(c) J-type Instruction Encoding

- Contains a 26-bit jump address field.
  - Extended to 28 bits by padding two 0's on the right.
- Example: *j Label*

31 26	25 0
opcode	Immediate Data

6-bit opcode — *26-bit jump address*

A Quick View

R-type (31–26, 25–21, 20–16, 15–11, 10–6, 5–0):

opcode	rs	rt	rd	shamt	funct

I-type (31–26, 25–21, 20–16, 15–0):

opcode	rs	rt	Immediate Data

J-type (31–26, 25–0):

opcode	Immediate Data

- Some instructions require two register operands *rs* & *rt* as input, while some require only *rs*.
- Gets known only after instruction is decoded.
- While decoding is going on, we can prefetch the registers in parallel.
  - May or may not be required later.

- Similarly, the 16-bit and 26-bit immediate data are retrieved and sign-extended to 32-bits in case they are required later.

Addressing Modes in MIPS32

- Register addressing — *add $s1, $s2, $s3*
- Immediate addressing — *addi $s1, $s2, 200*
- Base addressing — *lw $s1, 150($s2)*
  - Content of a register is added to a "base" value to get the operand address.
- PC relative addressing — *beq $s1, $s2, Label*
  - 16-bit offset is added to PC to get the target address.
- Pseudo-direct addressing — *j Label*
  - 26-bit offset if shifted left by 2 bits and then added to PC to get the target address.

END OF LECTURE 10

NPTEL ONLINE CERTIFICATION COURSES

NPTEL

IIT KHARAGPUR | NIT MEGHALAYA

Lecture 11: SPIM – A MIPS32 SIMULATOR

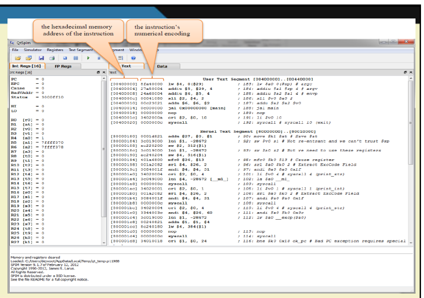
DR. KAMALIKA DATTA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

How to Run MIPS32 Programs?

- Best way to learn MIPS32 assembly language programming is through a simulator.
  - SPIM is a self-contained simulator available in the public domain that runs MIPS32 programs.
  - Available for download in http://spimsimulator.sourceforge.net
  - SPIM implements almost the entire MIPS32 instruction set (along with the extensions, viz. pseudo-instructions).

- SPIM has both terminal-based and window-based interfaces.
  - Terminal versions are available on Linux, Windows, and Mac OS X.
  - Window-based interface is provides by the QTSPIM program, which is also available on Linux, Windows and Mac OS X.
- *SPIM is copyrighted by James Larus and distributed under a BSD license.*
- What can SPIM do?
  - It can read and execute assembly language programs for MIPS32.
  - Provides a simple debugger.
  - Provides minimal set of OS services via system calls.

Screenshot of QTSPIM



MIPS32 Assembly Code Layout

```
        .text               # code section
        .globl main         # starting point, must be global
main:

        # user program code goes here

        .data               # data section

        # user program data goes here
```


Assembler Directives

a) .text
  - Specifies the user text segment, which contains the instructions.

b) .data
  - Specifies the data segment, where all the data items are defined.

c) .globl sym
  - Specifies that the symbol "sym" is global, and can be referred from other files.

d) .word w1, w2, …, wn
  - Stores the specified 32-bit numbers in successive memory words.

e) .half h1, h2, …, hn
  - Stores the specified 16-bit numbers in successive memory half-words.

f) .byte b1, b2, …, bn
  - Stores the specified 8-bit numbers in successive memory bytes.

g) .ascii str
  - Stores the specified string in memory (in ASCII code), but do not null-terminate it.
  - Strings are enclosed in double quotes and follow C-like convention ("\n", etc.).

h) .asciiz str
  - Stores the specified string in memory (in ASCII code), and null-terminate it.

i) .space n
  - Reserve space for n successive bytes in memory.

Register Naming Conventions

- Already discussed earlier :: quick recall :-
  - $zero — constant zero
  - $at — reserved by assembler
  - $v0, $v1 — for parameter passing
  - $a0 to $a3 — for arguments
  - $t0 to $t9 — temporary registers (not saved by callee)
  - $s0 to $s7 — registers (saved by callee)
  - $gp — global pointer
  - $sp — stack pointer
  - $ra — return address

Pseudo-instructions

- The MIPS32 pseudo-instructions, as discussed earlier, are all supported by SPIM.
- SPIM converts these into MIPS32 instructions before executing them.

Operating System Interface: "syscall"

Service	Code (put in $v0)	Arguments	Result
print_int	1	$a0 = integer	
print_string	4	$a0 = address of string	
read_int	5		int in $v0
read_string	8	$a0 = address of buffer, $a1 = length	
exit	10		

Other system calls for floating-point numbers also exist

Example Program 1

```
        .text
        .globl  main

main:   la      $t0, value
        lw      $t1, 0($t0)
        lw      $t2, 4($t0)
        add     $t3, $t1, $t2
        sw      $t3, 8($t0)

        .data
value:  .word   50, 30, 0
```

Add two numbers in memory and store the result in the next location.

Example Program 2

```
        .text
        .globl  main

main:   add     $t1, $zero, 0x2A
        add     $t2, $zero, 0x0D
        add     $s3, $t1, $t2
```

Add two constant numbers specified as immediate data, and store the result in a register.

Example Program 3

```
        .text
        .globl  main

main:   add     $t1, $zero, 0x2A
        add     $t2, $zero, 0x0D
        add     $s3, $t1, $t2

        li      $v0, 10
        syscall
```

The same program but using system call to exit.

Example Program 4

Read two numbers from the keyboard and print the sum.

```
        .data
str1: . asciiz "Enter first number: "
str2: . asciiz "Enter second number: "
str3: . asciiz "The sum is = "

        .text
        .globl   main

main:   li $v0, 4                  # print string
        la $a0, str1
        syscall

        li $v0, 5                  # read integer
        syscall
        move $t0, $v0
```

```
li      $v0, 4
la      $a0, str2
syscall

li      $v0, 5
syscall
move    $t1, $v0

add $t1, $t0, $t1
        # $t1= $t0 + $t1

li      $v0, 4
la      $a0, str3
syscall
```

```
li      $v0, 1
move    $a0, $t1
syscall

li      $v0,10
syscall
```


Example Program 5

Calculate sum of 10 32-bit numbers stored in consecutive memory locations.

```
        .data
num:    .word   1, 2, 3, 4, 5, 6 ,7, 8, 9, 10
        .text
        .globl  main
main:
        la      $t0, num
        li      $t2, 0          # holds the sum
        li      $t3, 0          # counter for loop
loop:   lw      $t1, 0($t0)
        add     $t2, $t2, $t1
        addi    $t3, $t3, 1
        addi    $t0, $t0, 4     # point to next
        bne     $t3, 10, loop

        li      $v0,10
        syscall
```


Example Program 6

Check if a given number is a palindrome.

```
        .data
num:    .word   0
msg:    .asciiz "Enter the Number: "
msg1:   .asciiz "Palindrome"
msg2:   .asciiz "Not Palindrome"

        .text
        .globl main
main:
        li      $v0,4
        la      $a0, msg
        syscall
```



```
        li      $v0,5
        syscall
        move    $t0, $v0
        move    $t3, $t0
        li      $t2, 0
loop:
        mul     $t2, $t2, 10
        rem     $t1,$t0,10
        div     $t0,$t0,10
        add     $t2, $t2, $t1
        bne     $t0,$zero,loop
        bne     $t3, $t2, np
```

```
        li      $v0, 4
        la      $a0, msg1
        syscall

        li      $v0, 10
        syscall
np:
        li      $v0, 4
        la      $a0, msg2
        syscall

        li      $v0, 10
        syscall
```


Function Calls in MIPS32

- MIPS uses the jump and link instructions.
  - Control is transferred to the function using the **jal** instruction.
  - The **jal** instruction jumps to a label and stores the PC value in the **$ra** register.
  - To transfer the control back to the caller program we use: **jr $ra**.

Example Program

Function call and return.

```
        .data
num1:   .word   14
num2:   .word   15
sum:    .word   0

        .text

main:
        lw      $t0, num1
        lw      $t1, num2
        jal     SumFunc
        sw      $t1, sum

        li      $v0,10
        syscall
```

```
SumFunc:
        add $t1,$t1,$t0
        jr  $ra
```

END OF LECTURE 11