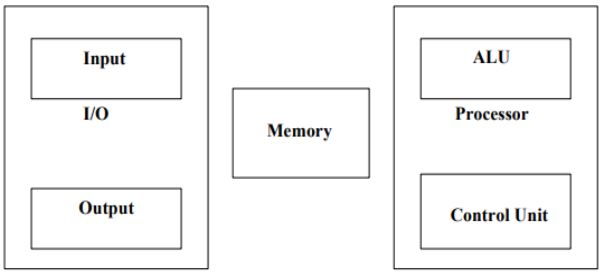
UNIT I BASIC STRUCTURE OF A COMPUTER SYSTEM

Functional Units – Basic Operational Concepts – Performance – Instructions: Language of the Computer – Operations, Operands – Instruction representation – Logical operations – Decision making – MIPS Addressing.

FUNCTIONAL UNITS OF COMPUTER SYSTEM

A computer consists of five functionally independent main parts Input, Memory, Arithmetic and Logic unit (ALU), Output and Control unit.



Input device accepts the coded information as source program i.e. high level language. This is either stored in the memory or immediately used by the processor to perform the desired operations. The program stored in the memory determines the processing steps. Basically the computer converts one source program to an object program. i.e. into machine language.

Finally the results are sent to the outside world through output device. All of these actions are coordinated by the control unit.

Input unit: - The source program/high level language program/coded information/simply data is fed to a computer through input devices keyboard is a most common type. Whenever a key is pressed, one corresponding word or number is translated into its equivalent binary code over a cable & fed either to memory or processor.

Example:- Joysticks, trackballs, mouse, scanners etc are other input devices.

Memory unit: - Its function into store programs and data. It is basically to two types

1. Primary memory
2. Secondary memory

1. Primary memory: - It Is the one exclusively associated with the processor and operates at the electronics speeds programs must be stored in this memory while they are being executed. The memory contains a large number of semiconductors storage cells. Each capable of storing one bit of information. These are processed in a group of fixed site called word.

To provide easy access to a word in memory, a distinct address is associated with each word location. Addresses are numbers that identify memory location.

Number of bits in each word is called word length of the computer. Programs must reside in the memory during execution. Instructions and data can be written into the memory or read out under the control of processor. Memory in which any location can be reached in a short and fixed amount of time after specifying its address is called random-access memory (RAM).

The time required to access one word in called memory access time. Memory which is only readable by the user and contents of which can't be altered is called read only memory (ROM) it contains operating system. Caches are the small fast RAM units, which are coupled with the processor and are often contained on the same IC chip to achieve high performance. Although primary storage is essential it tends to be expensive.

2 Secondary memory: - Is used where large amounts of data & programs have to be stored, particularly information that is accessed infrequently.

Examples: - Magnetic disks & tapes, optical disks (ie CD-ROM's), floppies etc.,

Arithmetic logic unit (ALU):-

Most of the computer operators are executed in ALU of the processor like addition, subtraction, division, multiplication, etc. the operands are brought into the ALU from memory and stored in high speed storage elements called register. Then according to the instructions the operation is performed in the required sequence.

The control and the ALU are may times faster than other devices connected to a computer system. This enables a single processor to control a number of external devices such as key boards, displays,

magnetic and optical disks, sensors and other mechanical controllers.

Output unit:-

These actually are the counterparts of input unit. Its basic function is to send the processed results to the outside world.

Examples:- Printer, speakers, monitor etc.

Control unit:-

It effectively is the nerve center that sends signals to other units and senses their states. The actual timing signals that govern the transfer of data between input unit, processor, memory and output unit are generated by the control unit.

BASIC OPERATIONAL CONCEPTS

To perform a given task an appropriate program consisting of a list of instructions is stored in the memory. Individual instructions are brought from the memory into the processor, which executes the specified operations. Data to be stored are also stored in the memory.

Examples: - Add LOCA, R0

This instruction adds the operand at memory location LOCA, to operand in register R0 & places the sum into register. This instruction requires the performance of several steps,

1. First the instruction is fetched from the memory into the processor
2. The operand at LOCA is fetched and added to the contents of R0
3. Finally the resulting sum is stored in the register R0

The preceding add instruction combines a memory access operation with an ALU Operations. In some other type of computers, these two types of operations are performed by separate instructions for performance reasons.

Load LOCA, R1

Add R1, R0

Transfers between the memory and the processor are started by sending the address of the memory location to be accessed to the memory unit and issuing the appropriate control signals. The data are then transferred to or from the memory. CONTROL ALU … n- GPRs

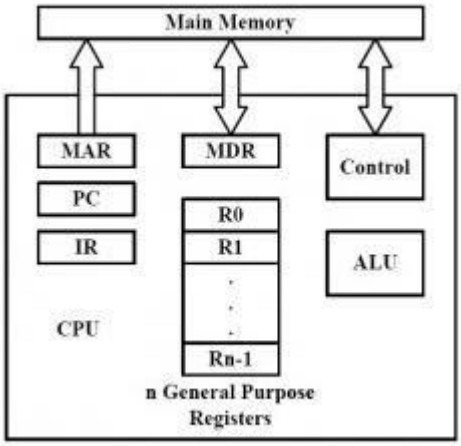


Fig : Connections between the processor and the memory

The fig shows how memory & the processor can be connected. In addition to the ALU & the control circuitry, the processor contains a number of registers used for several different purposes.

The instruction register (IR):- Holds the instructions that is currently being executed. Its output is available for the control circuits which generates the timing signals that control the various processing elements in one execution of instruction.

The program counter PC:- This is another specialized register that keeps track of execution of a program. It contains the memory address of the next instruction to be fetched and executed. Besides IR and PC, there are n-general purpose registers R0 through Rn-1. MAR PC IR MEMORY MDR R0 R1 …

The other two registers which facilitate communication with memory are: -

1. MAR – (Memory Address Register):- It holds the address of the location to be accessed.
2. MDR – (Memory Data Register):- It contains the data to be written into or read out of the address location.

Operating steps are

1. Programs reside in the memory & usually get these through the I/P unit.
2. Execution of the program starts when the PC is set to point at the first instruction of the program.
3. Contents of PC are transferred to MAR and a Read Control Signal is sent to the memory.
4. After the time required to access the memory elapses, the address word is read out of the memory and loaded into the MDR.
5. Now contents of MDR are transferred to the IR & now the instruction is ready to be decoded and executed.
6. If the instruction involves an operation by the ALU, it is necessary to obtain the required operands.
7. An operand in the memory is fetched by sending its address to MAR & Initiating a read cycle.
8. When the operand has been read from the memory to the MDR, it is transferred from MDR to the ALU.
9. After one or two such repeated cycles, the ALU can perform the desired operation.
10. If the result of this operation is to be stored in the memory, the result is sent to MDR.
11. Address of location where the result is stored is sent to MAR & a write cycle is initiated.
12. The contents of PC are incremented so that PC points to the next instruction that is to be executed.

Normal execution of a program may be preempted (temporarily interrupted) if some devices require urgent servicing, to do this one device raises an Interrupt signal.

An interrupt is a request signal from an I/O device for service by the processor. The processor provides the requested service by executing an appropriate interrupt service routine.

The Diversion may change the internal stage of the processor its state must be saved in the memory location before interruption. When the interrupt-routine service is completed the state of the processor is restored so that the interrupted program may continue.

CPU PERFORMANCE

The machine (or CPU) is said to be faster or has better performance running this program if the total execution time is shorter.

- Thus the in**verse of the total measure**d program execution time is a possible performance measure or metric: $\text{Performance}_A = 1 / \text{Execution Time}_A$
- **CPU performance**

$$\frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}$$

- **Instruction performance**

  CPU clock cycles = Instructions for a program × Average clock cycles per instruction
- **CPU Time**
  - ✓ doesn't count I/O or time spent running other programs
  - ✓ can be broken up into system time, and user time

  CPU time = Instruction Count × CPI × Clock Cycle Time

PERFORMANCE

If you were running a program on two different desktop computers, you'd say that the faster one is the desktop computer that gets the job done first. If you were running a datacenter that had several servers running jobs submitted by many users, you'd say that the faster computer was the one that completed the most jobs during a day.

As an individual computer user, you are interested in reducing **response time**—the time between the start and completion of a task—also referred to as execution time.

Datacenter managers are often interested in increasing **throughput or bandwidth**— the total amount of work done in a given time

To maximize performance, we want to minimize response time or execution time for some task. Thus, we can relate performance and execution time for a computer X:

$$\text{Performance}_X = \frac{1}{\text{Execution time}_X}$$

This means that for two computers X and Y, if the performance of X is greater than the performance of Y, we have

$$\text{Performance}_X > \text{Performance}_Y$$

$$\frac{1}{\text{Execution time}_X} > \frac{1}{\text{Execution time}_Y}$$

$$\text{Execution time}_Y > \text{Execution time}_X$$

That is, the execution time on Y is longer than that on X, if X is faster than Y.

In discussing a computer design, we often want to relate the performance of two different computers quantitatively. We will use the phrase "X is *n* times faster than Y"—or equivalently "X is *n* times as fast as Y"—to mean

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = n$$

If X is *n* times faster than Y, then the execution time on Y is *n* times longer than it is on X:

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution time}_Y}{\text{Execution time}_X} = n$$

Measuring Performance

CPU execution time Also called CPU time. The actual time the CPU spends computing for a specific task.

User CPU time The CPU time spent in a program itself.

System CPU time The CPU time spent in the operating system performing tasks on behalf of the program.

CPU Performance and Its Factors

Since we are confining ourselves to CPU performance at this point, the bottom-line performance measure is CPU execution time. A simple formula relates the most basic metrics (clock cycles and clock cycle time) to CPU time:

$$\frac{\text{CPU execution time}}{\text{for a program}} = \frac{\text{CPU clock cycles}}{\text{for a program}} \times \text{Clock cycle time}$$

Alternatively, because clock rate and clock cycle time are inverses,

$$\frac{\text{CPU execution time}}{\text{for a program}} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

Instruction Performance

The performance equations above did not include any reference to the number of instructions needed for the program. (We'll see what the instructions that make up a program look like in the next chapter.) However, since the compiler clearly generated instructions to execute, and the computer had to execute the instructions to run the program, the execution time must depend on the number of instructions in a program. One way to think about execution time is that it equals the number
of instructions executed multiplied by the average time per instruction. Therefore, the number of clock cycles required for a program can be written as

$$\text{CPU clock cycles} = \text{Instructions for a program} \times \frac{\text{Average clock cycles}}{\text{per instruction}}$$

The term clock cycles per instruction, which is the average number of clock cycles each instruction takes to execute, is often abbreviated as CPI. Since different instructions may take different amounts of time depending on what they do, CPI is an average of all the instructions executed in the program. CPI provides one way of comparing two different implementations of the same instruction set architecture, since the number of instructions executed for a program will, of course, be the same.

PERFORMANCE EQUATION

We can now write this basic performance equation in terms of instruction count (the number of instructions executed by the program), CPI, and clock cycle time:

$$\text{CPU time} = \text{Instruction count} \times \text{CPI} \times \text{Clock cycle time}$$

or, since the clock rate is the inverse of clock cycle time:

$$\text{CPU time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}}$$

These formulas are particularly useful because they separate the three key factors that affect performance.

The performance of a program depends on the algorithm, the language, the compiler, the architecture, and the actual hardware. The following table summarizes how these components affect the factors in the CPU performance equation.

Hardware or software component	Affects what?	How?
Algorithm	Instruction count, possibly CPI	The algorithm determines the number of source program instructions executed and hence the number of processor instructions executed. The algorithm may also affect the CPI, by favoring slower or faster instructions. For example, if the algorithm uses more floating-point operations, it will tend to have a higher CPI.
Programming language	Instruction count, CPI	The programming language certainly affects the instruction count, since statements in the language are translated to processor instructions, which determine instruction count. The language may also affect the CPI because of its features; for example, a language with heavy support for data abstraction (e.g., Java) will require indirect calls, which will use higher CPI instructions.
Compiler	Instruction count, CPI	The efficiency of the compiler affects both the instruction count and average cycles per instruction, since the compiler determines the translation of the source language instructions into computer instructions. The compiler's role can be very complex and affect the CPI in complex ways.
Instruction set architecture	Instruction count, clock rate, CPI	The instruction set architecture affects all three aspects of CPU performance, since it affects the instructions needed for a function, the cost in cycles of each instruction, and the overall clock rate of the processor.

INSTRUCTIONS: LANGUAGE OF THE COMPUTER – OPERATIONS AND OPERANDS

The words of a computer's language are called instructions, and its vocabulary is called an instruction set.

Operands

1. Register Operands

- Arithmetic instructions use register operands
- MIPS has a 32 × 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

Example 1:

C code for the following instruction:

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

2. 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

3. Constant or 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

Operations

1. Arithmetic Operations

Add and Subtract instruction use three Operands - Two Sources and one Destination.

Example add $t0, $s1, $s2

2. Data Transfer Operations

These operations help in moving the data between memory and registers.

Example 1:

C code:

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

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

Compiled MIPS code:

- Index 8 requires offset of 32
  - 4 bytes per word

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

offset — base register

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
```

3. Logical Operations

Instructions for bitwise manipulation

Useful for extracting and inserting groups of bits in a word

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

4. Conditional Operations

1. Branch to a labeled instruction if a condition is true. Otherwise, continue sequentially.
   beq rs, rt, L1
   if (rs == rt) branch to instruction labeled L1;
   bne rs, rt, L1
   if (rs != rt) branch to instruction labeled L1;
   j L1
2. Unconditional jump to instruction labeled L1. Set result to 1 if a condition is true. Otherwise, set to 0.
   slt rd, rs, rt
   if (rs < rt) rd = 1; else rd = 0;
   slti rt, rs, constant
   if (rs < constant) rt = 1; else rt = 0;
3. Use in combination with beq, bne
   slt $t0, $s1, $s2 # if ($s1 < $s2)

bne $t0, $zero, L # branch to L

Jump Instructions

4. Procedure call: jump and link.

   jal ProcedureLabel

   Address of following instruction put in $ra.

   Jumps to target address.

5. Procedure return: jump register

   jr $ra

   Copies $ra to program counter.

   Can also be used for computed jumps.

   e.g., for case/switch statements.

REPRESENT INSTRUCTIONS IN A COMPUTER SYSTEM

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.

- Register numbers

  $t0 – $t7 are reg's 8 – 15

  $t8 – $t9 are reg's 24 – 25

  $s0 – $s7 are reg's 16 – 23

1. R-Format:

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

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)

Example:

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

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$0000001000110010010000000010000_2 = 02324020_{16}$

2. I-Format:

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

Immediate arithmetic and load/store instructions.

Instruction Fields:

rt: destination or source register number

Constant: –215 to +215 – 1

Address: offset added to base address in rs

Example:

lw $t0,32($s3) # Temporary reg $t0 gets A[8]

Here, 19 (for $s3) is placed in the rs field, 8 (for $t0) is placed in the rt field, and 32 is placed in the address field. Note that the meaning of the rt field has changed for this instruction: in a load word instruction, the rt field specifies the destination register, which receives the result of the load.

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
add immediate	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

In the table above, "reg" means a register number between 0 and 31, "address" means a 16-bit address, and "n.a."(not applicable) means this field does not appear in this format. Note that add and sub instructions have the same value in the op field; the hardware uses the function field to decide the variant of the operation: add (32) or subtract (34).

3. **J-Format:**

j target- J-type is short for "jump type". The format of an J-type instruction looks like:

opcode	target
B31-26	B25-0

The semantics of the j instruction (j means jump) are:

PC <- PC31-28 IR25-0 00

where PC is the program counter, which stores the current address of the instruction being executed. You update the PC by using the upper 4 bits of the program counter, followed by the 26 bits of the target (which is the lower 26 bits of the instruction register), followed by two 0's, which creates a 32 bit address.

LOGICAL OPERATIONS

1. **Logical Operations:**

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit NOT	~	~	nor

2. **SHIFT:**

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

shamt: how many positions to shift

Shift left logical

Shift left and fill with 0 bits

sll by i bits multiplies by 2i

Shift right logical

Shift right and fill with 0 bits

srl by i bits divides by 2i (unsigned only)

3. **AND:**

Useful to mask bits in a word

Select some bits, clear others to 0

Eg: and $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 0000 1100 0000 0000

4. **OR:**

Used to include bit in a word

Set some bits to 1, leave others unchanged

Eg: 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 1101 1100 0000

5. NOT:

Useful to invert bits in a word

Change 0 to 1, and 1 to 0

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

DECISION MAKING

Based on the input data and the values created during computation, different instructions execute. Decision making is commonly represented in programming languages using if statement, sometimes combined with go to statements and labels. MIPS assembly language includes two decision-making instructions, similar to an if statement with a go to. The first instruction is

beq register1, register2, L1

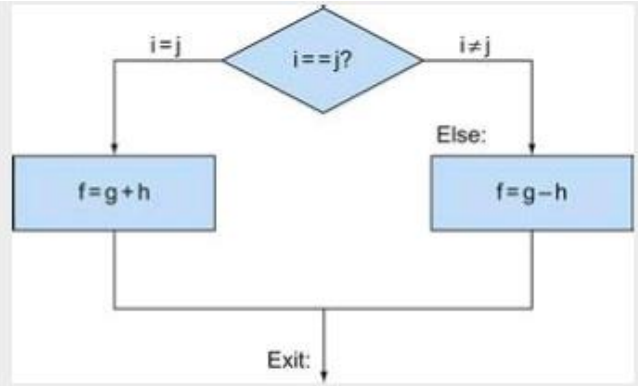
This instruction means go to the statement labeled L1 if the value in register1 equals the value in register2. The mnemonic beq stands for branch if equal. The second instruction is

bne register1, register2, L1

It means go to the statement labeled L1 if the value in register1 does not equal the value in register2. The mnemonic bne stands for bra nch if not equal. These two instructions are traditionally called conditional branches.

Example:

Compiling IF statements:



C code:

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

- f, g, … in $s0, $s1, …

Compiled MIPS code:

```
      bne $s3, $s4, Else
      add $s0, $s1, $s2
      j   Exit
Else: sub $s0, $s1, $s2
Exit: …
```

Assembler calculates addresses

Compiling Loop Statements:

C code:

```
while (save[i] == k) i += 1;
```

- i in $s3, k in $s5, address of save in $s6

Compiled MIPS code:

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

More Conditional operations:

Set result to 1 if a condition is true

Otherwise, set to 0

slt rd, rs, rt

if (rs < rt) rd = 1; else rd = 0;

slti rt, rs, constant

if (rs < constant) rt = 1; else rt = 0;

Use in combination with beq, bne

slt $t0, $s1, $s2 # if ($s1 < $s2)

bne $t0, $zero, L # branch to L

Signed comparison: slt, slti

Unsigned comparison: sltu, sltui

Example

$s0 = 1111 1111 1111 1111 1111 1111 1111 1111

$s1 = 0000 0000 0000 0000 0000 0000 0000 0001

slt $t0, $s0, $s1 # signed

–1 < +1 ⇒ $t0 = 1

sltu $t0, $s0, $s1 # unsigned

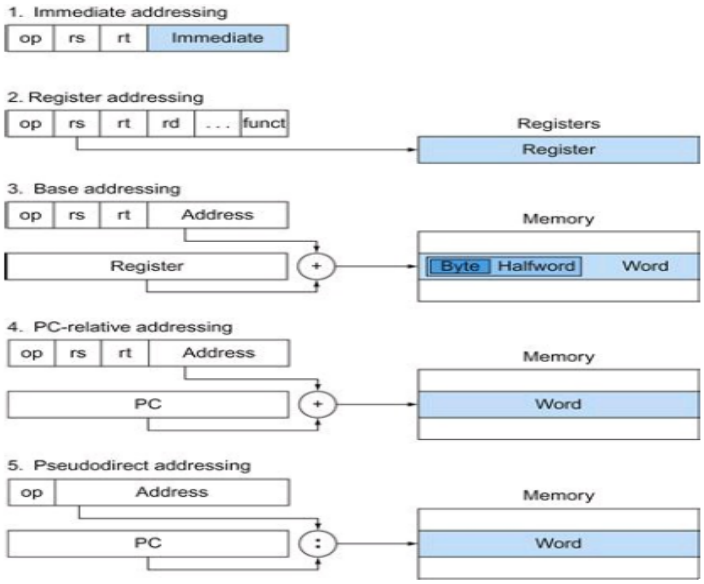
+4,294,967,295 > +1 ⇒ $t0 = 0

MIPS ADDRESSING

Multiple forms of addressing are generically called as addressing modes. The method used to identify the location of an operand. The following are the various types of MIPS addressing modes.

1. **Immediate addressing**, where the operand is a constant within the instruction itself.
2. **Register addressing**, where the operand is a register.
3. **Base or displacement addressing**, where the operand is at the memory location whose address is the sum of a register and a constant in the instruction.
4. **PC-relative addressing**, where the branch address is the sum of the PC and a constant in the Instruction.
5. **Pseudo direct addressing**, where the j ump address is the 26 bits of the instruction concatenated with the upper bits of the PC.

The operands are shaded in color. The operand of mode 3 is in memory, whereas the operand for mode 2 is a register. Note that versions of load and store access bytes, half words, or words. For mode 1, the operand is 16 bits of the instruction itself. Modes 4 and 5 address instructions in memory, with mode 4 adding a 16-bit address shifted left 2 bits to the PC and mode 5 concatenating a 26-bit address shifted left 2 bits with the 4 upper bits of the PC.



UNIT II ARITHMETIC FOR COMPUTERS

Addition and Subtraction – Multiplication – Division – Floating Point Representation – Floating Point Operations – Subword Parallelism

ADDITION OPERATION :FAST ADDER

Fast adder circuit must speed up the generation of carry signals. Carry lookahead logic uses the concepts of generating and propagating carries.

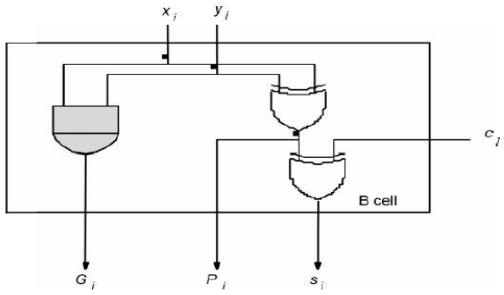
$$s_i = x_i \oplus y_i \oplus c_i$$
$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

Where s_i is the sum and C_{i+1} is the carry out

$$c_{i+1} = G_i + P_i c_i$$
$$where\ G_i = x_i y_i\ and\ P_i = x_i + y_i$$

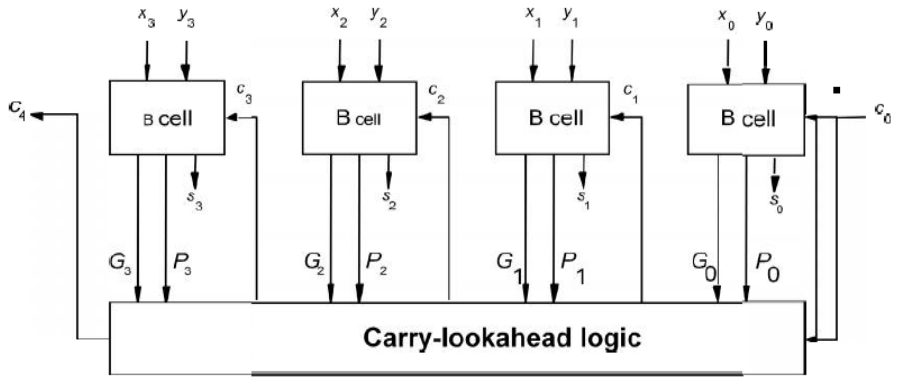
Giand Piare computed only from x_i and y_i and not c_i, thus they can be computed in one gate delay after X and Y are applied to the inputs of an n-bit adder.

Bit-stage cell



4-bit Adder

The complete 4-bit adder is shown in below figure where the B cell indicates Gi, Pi & Si generator. The carries are implemented in the block labeled carry look-ahead logic. An adder implemented in this form is called a carry lookahead adder. Delay through the adder is 3 gate delays for all carry bits and 4 gate delays for all sum bits. In comparison, note that a 4-bit ripple-carry adder requires 7 gate delays for S3(2n-1)and 8 gate delays(2n) for c4.



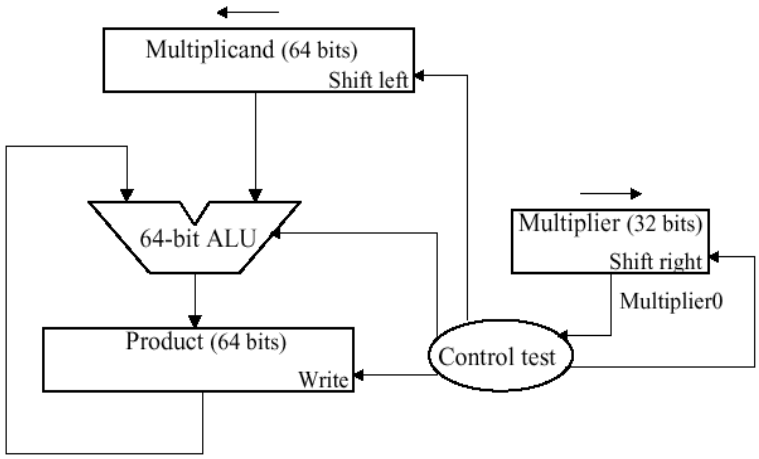
Now, consider the design of a 4-bit parallel adder. The carries can be implemented as

$G_i = A_i.B_i \qquad Pi = (A_i \oplus B_i)$

$C_1 = G_0 + P_0.C_0$
$C_2 = G_1 + P_1.C_1 = G_1 + P1.G_0 + P_1.P_0.C_0$
$C_3 = G_2 + P_2.G_1 + P_2.P_1.G_0 + P_2.P_1.P_0.C_0$
$C_4 = G_3 + P_3.G_2 + P_3.P_2.G_1 + P_3P_2P_1.G_0 + P_3P_2P_1P_0.C_0$

$S_i = A_i \oplus B_i \oplus C_i = P_i \oplus C_i.$

MULTIPLICATION OPERATION WITH ALGORITHM

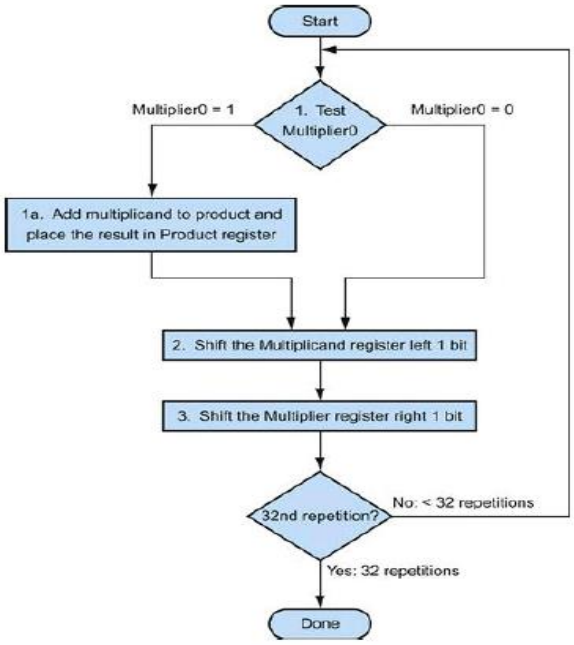


Hardware resemble the paper- and-pencil method

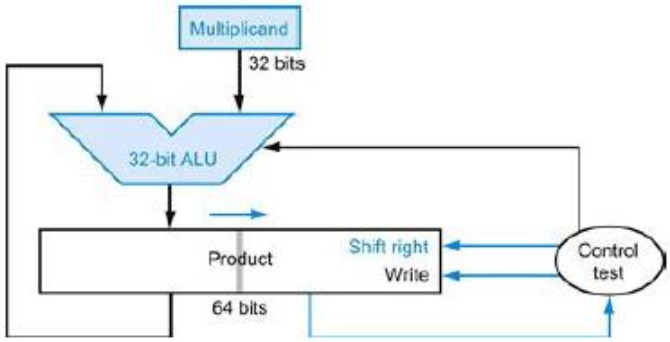
- Move the multiplicand left one digit each step to add with intermediate products
- Over 32 steps, a 32-bit multiplicand move 32-bits to left
- Hence 64-bit Multiplicand register initialized with 32-bit multiplicand in right half and zero in the left half

First Multiplication Algorithm using the previous hardware
3 basic steps for each bit

1. LSB bit of multiplier (Multiplier0) determines whether multiplicand is added to the product register
2. Left shift move the intermediate operands to left.
3. Shift right of multiplier register give us the next bit of the multiplier to examine for the next iteration. Above 3 steps repeated 32 times to obtain the Product.



Refined Version of Multiplication Hardware



- Previous algorithm easily refined to take 1 clock cycle per step
- Speed up by performing operations in parallel
- if the multiplier bit is 1, Multiplier and Multiplicand are shifted while the Multiplicand is added to the product
- An ALU addition operation can be very time consuming when done repeatedly.

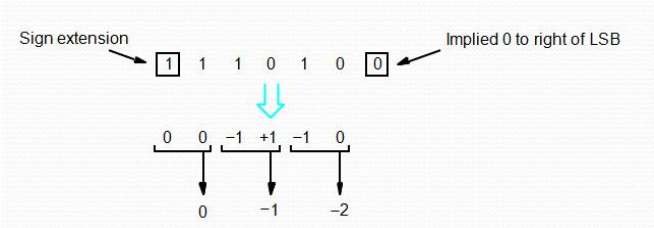
Computers can shift bits faster than adding bits.

BOOTH'S BIT-PAIR RECODING OF THE MULTIPLIER.

A=+13 (Multiplicand) AND B= -6 (Multiplier)

Bit-pair recoding halves the maximum number of summands (versions of the multiplicand).

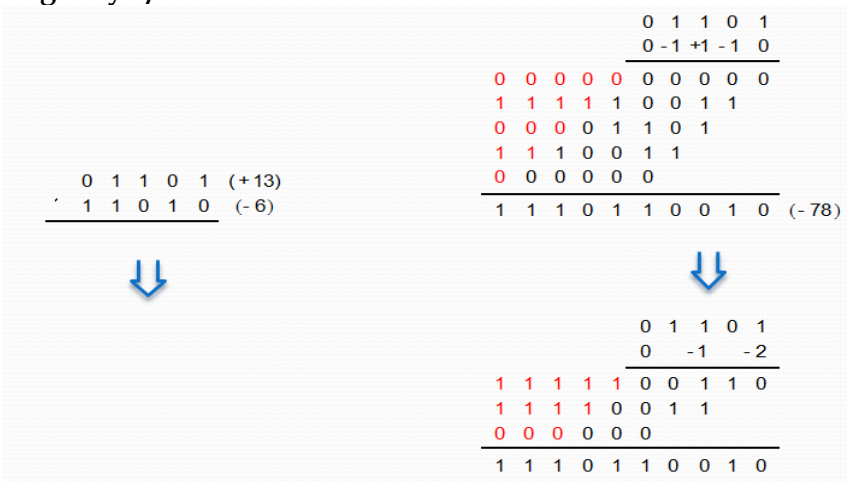
Example



Multiplicand Selection Decisions

Multiplier bit-pair		Multiplier bit on the right	Multiplicand selected at position *i*
i + 1	*i*	*i* − 1	
0	0	0	0 X M
0	0	1	+ 1 X M
0	1	0	+ 1 X M
0	1	1	+ 2 X M
1	0	0	− 2 X M
1	0	1	− 1 X M
1	1	0	− 1 X M
1	1	1	0 X M

Multiplication requiring only n/2 summands



BOOTH'S MULTIPLICATION ALGORITHM WITH SUITABLE EXAMPLE

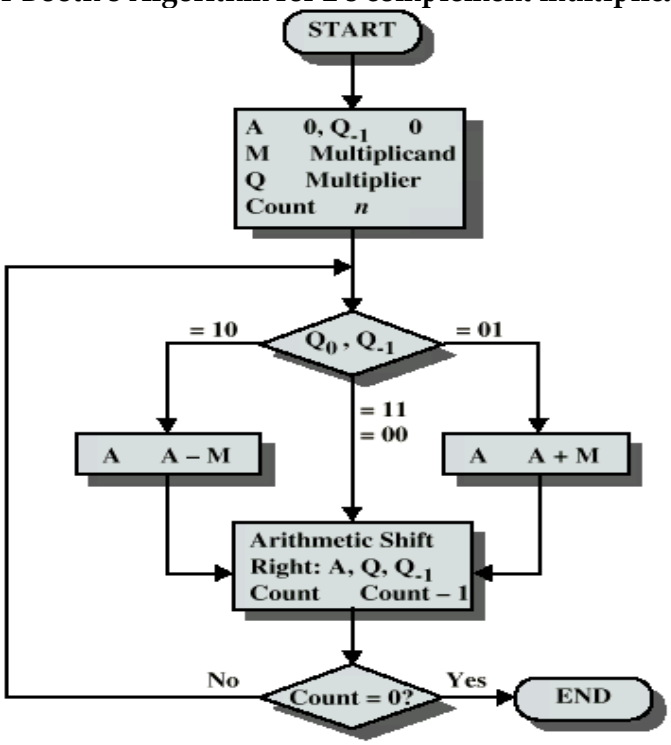
Booth's Algorithm Principle:

- Performs additions and subtractions of the Multiplicand, based on the value of the multiplier bits.
- The algorithm looks at two adjacent bits in the Multiplier in order to decide the operation to be performed.
- The Multiplier bits are considered from the least significant bit (right-most) to the most significant bit; by default a 0 will be considered at the right of the least significant bit of the multiplier.
- If Multiplicand has Md bits and Multiplier has Mp bits, the result will be stored in a Md+Mp bit register and will be initialised with 0s
- As repeated operations and shifts are performed on partial results, the result register is the accumulator (A).
- Booth's algorithm gives a procedure for multiplying signed binary integer. It is based on the fact that strings of 0's in the multiplier require no addition but only shifting and a string of 1's in the multiplier require both operations.
- Algorithm

The Q_0 bit of the register Q and Q_{-1} is examined:

- If two bits are the same (1-1 or 0-0), then all of the bits of the A, Q and Q-1 registers are shifted to the right 1 bit. This shift is called arithmetic shift right.
- If two bits differ i.e., whether 0-1, then the multiplicand is adder or 1-0, then the multiplicand is subtracted from the register A. after that, right shift occurs in the register A, Q and Q-1.

Flowchart of Booth's Algorithm for 2's complement multiplication



A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial Values	
1001	0011	0	0111	A A - M	First Cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	Second Cycle
0101	0100	1	0111	A A + M	Third Cycle
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	Fourth Cycle

VARIOUS METHODS OF PERFORMING MULTIPLICATION OF N-BIT NUMBERS

In multiplication the two input arguments are the multiplier Q given by Q=qn-1 qn-2…q1q0 and the multiplicand M given by M=mn-1 mn-2… m1mo.

1. The Paper and Pencil Method(for unsigned numbers)

This is the simplest method for performing multiplication of two unsigned numbers.

Example

Consider the multiplication of the two unsigned numbers 14 and 10.The process is shown below using the binary representation of the two numbers

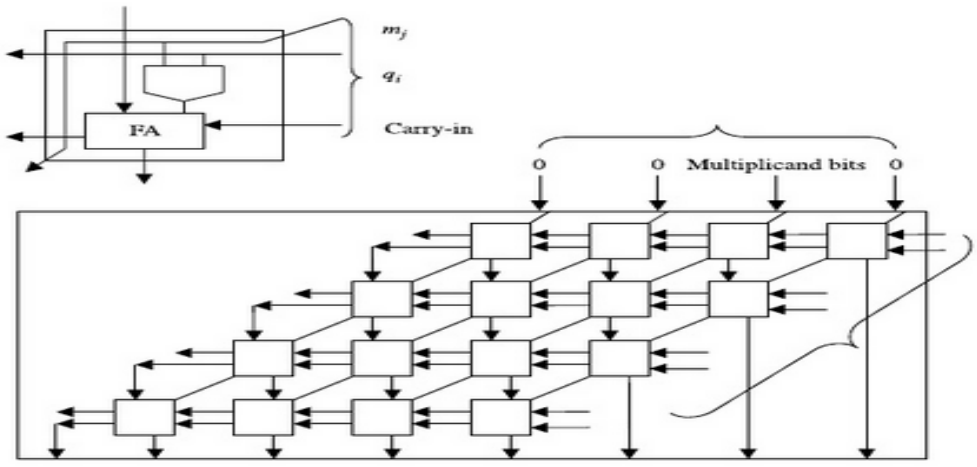
```
    1110 (14) Multiplicand( M)
    1010 (10) Multiplier(Q)

    ------
     0000     (Partial Product)
    1110      (Partial Product)
   0000       (Partial Product)
  1110        (Partial Product)
  ======
 10001100 (140) Final Product(P)
```

The above multiplication can be performed using an array of cells each consisting of an FA and an AND. Each cell computes a given partial product. Figure below shows the basic cell and an example array for a 4 x 4 multiplier array. If a given bit of the multiplier is 0 then there should be no need for computing the corresponding partial product.

2. ADD-SHIFT Method

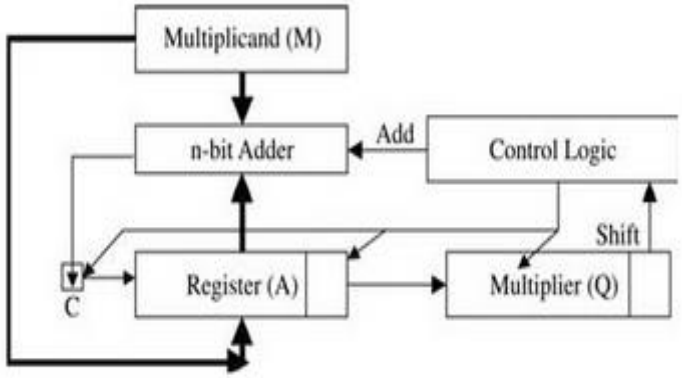
Multiplication is performed as a series of (n) conditional addition and shift operations such that if the given bit of the multiplier is 0 then only a shift operation is performed, while if the given bit of the multiplier is 1 then addition of the partial product and a shift operation is performed.

Array multiplier of Unsigned binary numbers



Example

Multiplication of two unsigned numbers 11 and 13.The process is shown below in a table. A is a 4-bit register and is initialized to 0s and C is the carry bit from the most significant bit position. The process is repeated n=4 times (the number of bits in the multiplier Q).If the bit of the multiplier is "1" then A=A+M and the concatenation of AQ is shifted one bit position to the right. If on the other hand the bit is "0" then only a shift operation is performed on AQ. The structure of the operation is given in the figure below.

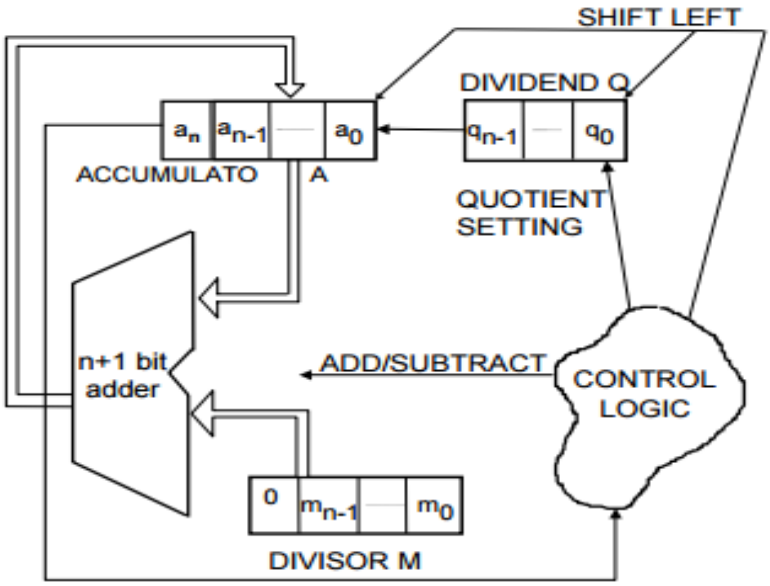


The control logic is used to determine the operation to be performed depending on the LSB in Q. An n-bit adder is used to add the contents of register A and M

M	*C*	*A*	*Q*		
1011	0	0000	1101	Initial values	
1011	0	1011	1101	Add	First cycle
1011	0	0101	1110	Shift	
1011	0	0010	1111	Shift	Second cycle
1011	0	1101	1111	Add	Third cycle
1011	0	0110	1111	Shift	
1011	1	0001	1111	Add	Fourth cycle
1011	0	1000	1111	Shift	

3. Booth's Algorithm: Answer is in previous question

DIVISION ALGORITHM IN DETAIL WITH DIAGRAM



The Figure shows a logic circuit arrangement that implements restoring division. An n-bit positive divisor is loaded into register M and an n-bit positive dividend is loaded into register Q at the start of the operation. Register A is set to 0. After the division is complete, the n-bit quotient is in register Q and the remainder is in register A. The required subtractions are facilitated by using 2's-complement arithmetic. The extra bit position at the left end of both A and M accommodates the sign bit during subtractions.

Algorithm for Restoring Division:

Do n times:

{ left shift A and Q by 1 bit

A← A - M;

if A < 0 (a_{n-1} = 1) , then $q_0 \leftarrow 0$, A ← A+M , (restore)
else
$q_0 \leftarrow 1$;}

Note: When A and Q are left shifted, the MSB of Q becomes the LSB of A, and the MSB of A is lost. The LSB of Q is made available for the next quotient bit.

Example: 8 divide by 3=2 (2/3)

	[M]	0011		
	[A]	0000	[Q]	1000
left shift A/Q		0001		000.
$A = [A] - [M]$	+	1101		
$A < 0$		1110		0000
$A = [A] + [M]$	+	0011		
		0001		0000
left shift A/Q		0010		000.
$A = [A] - [M]$	+	1101		
$A < 0$		1111		0000
$A = [A] + [M]$	+	0011		
		0010		0000
left shift A/Q		0100		000.
$A = [A] - [M]$	+	1101		
$A > 0$		0001		0001
left shift A/Q		0010		001.
$A = [A] - [M]$	+	1101		
$A < 0$		1111		0010
$A = [A] + [M]$	+	0011		
		0010		0010

The quotient $(0010)_2 = 2$ is in register Q, and the reminder $(0010)_2 = 2$ is in register A.

Algorithm for hardware division (non-restoring)

In the algorithm above, if the subtraction produces a non-positive result (A>0), registers A and Q are left shifted and the next subtraction is carried out. But if the subtraction produces a negative result (A<0), the dividend need be first restored by adding the divisor back before left shift A and Q and the next subtraction:

If A >= 0, then 2A - M (left shift and subtract);
If A < 0, then 2(A+M) – M = 2A + M (restore, left shift and subtract).
Note that when A < 0, the restoration is avoided by combining the two steps. This leads to a faster non-restoring division algorithm:

Algorithm for hardware division (non-restoring)

Do n times:
{ left shift A and Q by 1 bit
if (previous A >=0) then A← A - M
else A← A + M ;
if (current A >=0) then $q_0 \leftarrow 1$;
else $q_0 \leftarrow 0$;
}
if (A<0) then A← A + M (remainder must be positive)

	[M]	0011		
	[A]	0000	[Q]	1000
left shift A/Q		0001		000.
A=[A]-[M]	+	1101		
$A < 0$		1110		0000
left shift A/Q		1100		000.
$A = [A] + [M]$	+	0011		
$A < 0$		1111		0000
left shift A/Q		1110		000.
$A = [A] + [M]$	+	0011		
$A > 0$		0001		0001
left shift A/Q		0010		001.
$A = [A] - [M]$	+	1101		
$A < 0$		1111		0010
$A = [A] + [M]$	+	0011		
		0010		0010

The quotient $(0010)_2$ is in register Q, and the reminder $(0010)_2$ is in register A. The restoring division requires two operations (subtraction followed by an addition to restore) for each zero in the quotient. But non-restoring division only requires one operation (either addition or subtraction) for each bit in quotient.

FLOATING POINT ADDITION OPERATION

First, the exponent of one operand is subtracted from the other using the small ALU to determine which is larger and by how much. This difference controls the three multiplexors; from left to right, they select the larger exponent, the significand of the smaller number, and the significand of the larger number. The smaller significand is shifted right, and then the significands are added together using the big ALU. The normalization step then shifts the sum left or right and increments or decrements the exponent. Rounding then creates the final result, which may require normalizing again to produce the final result.

Example: Add numbers in scientific Notation

$9.999_{10} \times 10^{1} + 1.610_{10} \times 10^{-1}$

1. Align decimal points

   Shift number with smaller exponent

   $9.999_{10} \times 10^{1} + .01610_{10} \times 10^{1}$

   Now Equalize exponent

   $0.01610_{10} \times 10^{1} = 0.016 \times 10^{1}$

2. Add significands

   $9.999_{10} \times 10^{1} + 0.016_{10} \times 10^{1} = 10.015_{10} \times 10^{1}$

3. Normalize result & check for over/underflow

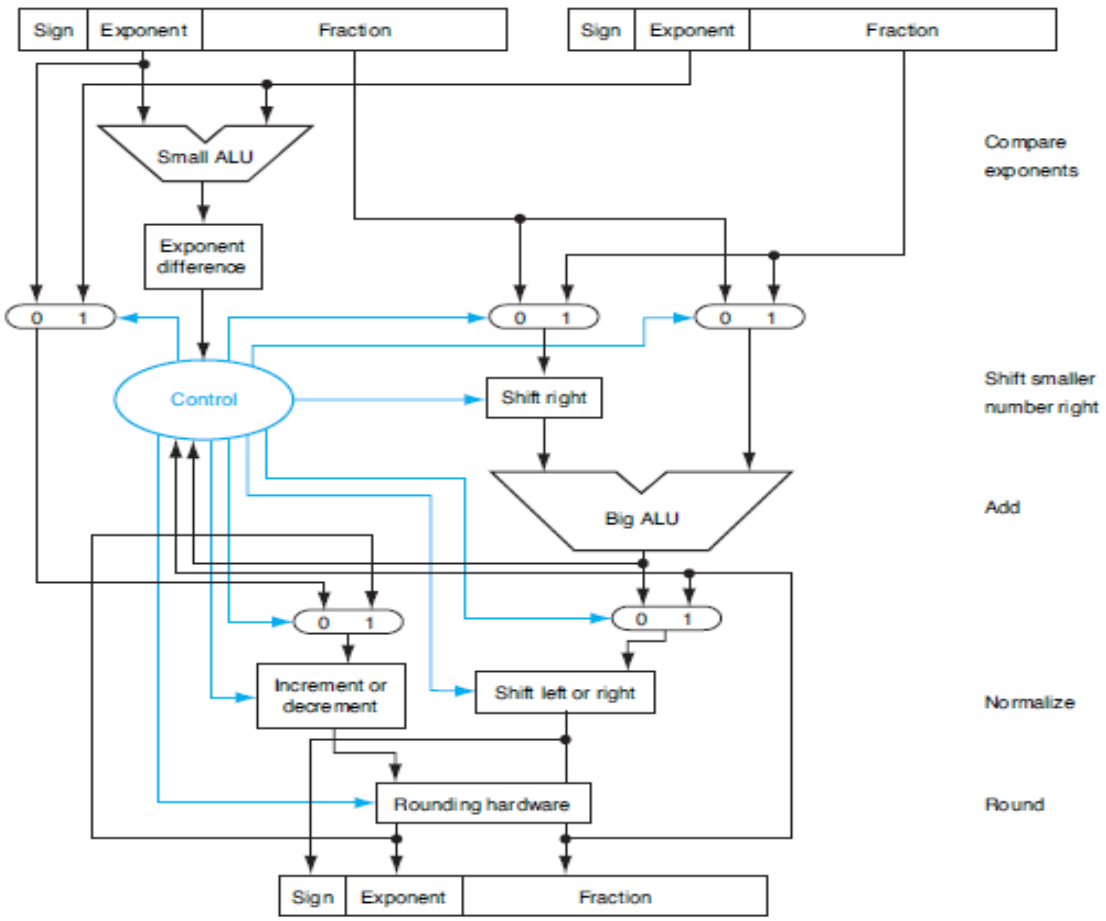
   $1.0015_{10} \times 10^{2}$

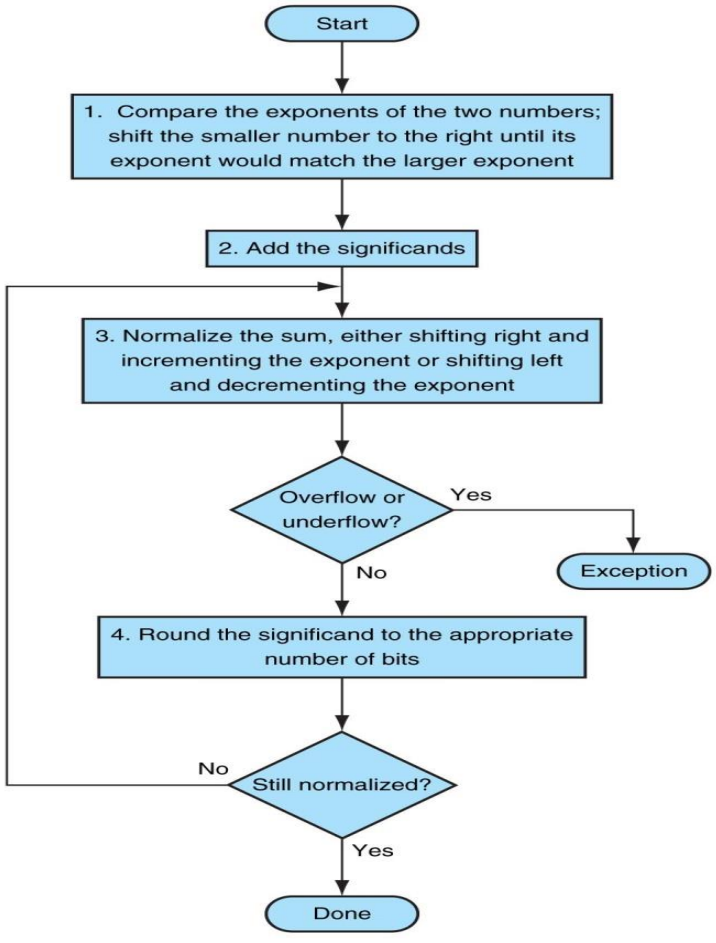
4. Round and renormalize if necessary

   1.002×10^{2}

The first step shifts the significand of the smaller number to the right until its corrected exponent matches that of the larger number

- If sum is not normalized adjust it
- Adjust the exponent
- Whenever the exponent is increased or decreased check for overflow or underflow(Exponent should fits in its field)


Sign
Exponent
Fraction
Sign
Exponent
Fraction
Small ALU
Compare exponents
Exponent difference
0 1
0 1
0 1
Control
Shift right
Shift smaller number right
Big ALU
Add
0 1
0 1
Increment or decrement
Shift left or right
Normalize
Rounding hardware
Round
Sign
Exponent
Fraction


Start
1. Compare the exponents of the two numbers; shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
Done

SUBWORD PARALLELISM

A subword is a lower precision unit of data contained within a word. In subword parallelism, multiple subwords are packed into a word and then process whole words. With the appropriate subword boundaries this technique results in parallel processing of subwords. Since the same instruction is applied to all subwords within the word, This is a form of SIMD(Single Instruction Multiple Data) processing. It is possible to apply subword parallelism to noncontiguous subwords of different sizes within a word. In practical implementation is simple if subwords are same size and they are contiguous within a word. The data parallel programs that benefit from subword parallelism tend to process data that are of the same size.

For example,

- If word size is 64bits and subwords sizes are 8,16 and 32 bits. Hence an instruction operates on eight 8bit subwords, four 16bit subwords, two 32bit subwords or one 64bit subword in parallel.
- Subword parallelism is an efficient and flexible solution for media processing because algorithm exhibit a great deal of data parallelism on lower precision data.
- It is also useful for computations unrelated to multimedia that exhibit data parallelism on lower precision data.
- Graphics and audio applications can take advantage of performing simultaneous operations on short vectors

Example: 128-bit adder
Sixteen 8-bit adder
Eight 16-bit adder
Four 32-bit adder

Also called data-level parallelism, vector parallelism, or Single Instruction, Multiple Data (SIMD)

UNIT III PROCESSOR AND CONTROL UNIT

Basic MIPS implementation – Building datapath – Control Implementation scheme – Pipelining – Pipelined datapath and control – Handling Data hazards & Control hazards – Exceptions.

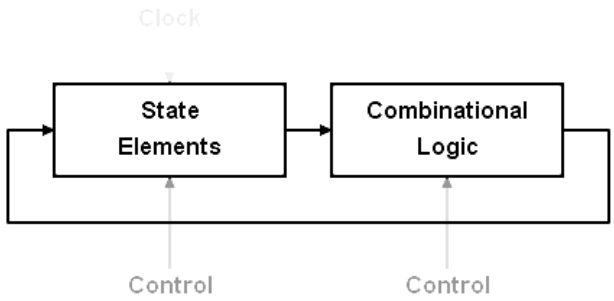
BASIC MIPS IMPLEMENTATION

Logic circuits use two different values of a physical quantity, usually voltage, to represent the boolean values true (or 1) and false (or 0). Logic circuits can have inputs and they have one or more outputs that are, at least partially, dependent on their inputs. In logic circuit diagrams, connections from one circuit's output to another circuit's input are often shown with an arrowhead at the input end. In terms of their behavior, logic circuits are much like programming language functions or methods. Their inputs are analogous to function parameters and their outputs are analogous to function returned values. However, a logic circuit can have multiple outputs.

There are two basic types of logic circuitry: combinational circuitry and state circuitry. Combinational circuitry behaves like a simple function. The output of combinational circuitry depends only on the current values of its input. State circuitry behaves more like an object method. The output of state circuitry does not just depend on its inputs — it also depends on the past history of its inputs. In other words, the circuitry has memory.

This is much like an object method whose value is dependent on the object's state: its instance variables. These two types of circuitry work together to make up a processor datapath.

Processor Datapath



Processor datapath control signals can be classified according to which part of the processor they control:

- State controls
- Combinational controls

A processor's datapath is conceptually organized into two parts:

State elements hold information about the state of the processor during the current clock cycle. All registers are state elements.

Combinational logic determines the state of the processor for the next clock cycle. The ALU is combinational logic.

Single-cycle organization

It is characterized by the fact that each instruction is executed in a single clock cycle. It is not a realistic implementation — it requires two separate memories: one for data and one for instructions. Also, the clock cycle has to be made quite long in order for all of the signals generated in a cycle to reach stable values.

Multi-cycle organization

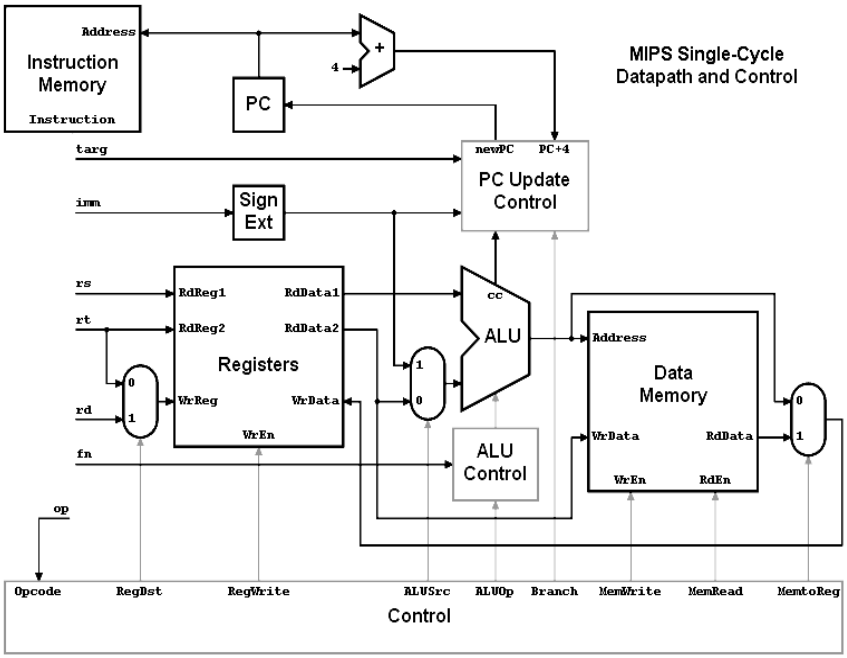
This organization uses multiple clock cycles for executing each instruction. Each cycle does only a small part of the work required so the cycles are much shorter. Much of the circuitry is the same as the single-cycle implementation. However, more state components must be added to hold data that is generated in an early cycle but used in a later cycle.

Pipelined organization

Like the multicycle organization, the pipelined organization uses multiple clock cycles for executing each instruction. By adding more state components for passing data and control signals between cycles, the pipelined organization turns the circuitry into an assembly line. After the first cycle of one instruction has completed you can start the execution of another instruction, while the first moves to its next cycle. Several instructions can be in different phases of execution at the same time.

Register renaming organization

Register renaming is an extension of the pipelining idea. It deals with the data dependence problem for a pipeline — the fact that instructions in the pipeline produce data needed by other instructions in the pipeline.



BASIC MIPS IMPLEMENTATION WITH NECESSARY MULTIPLEXERS AND CONTROL LINES

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the *rt* field (bits 20-16).	The register destination number for the Write register comes from the *rd* field (bits 15-11).
RegWrite	None	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC+4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None	Data memory contents disgnated by the address input are put on the Read data output.
MemWrite	None	Data memory contents disgnated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

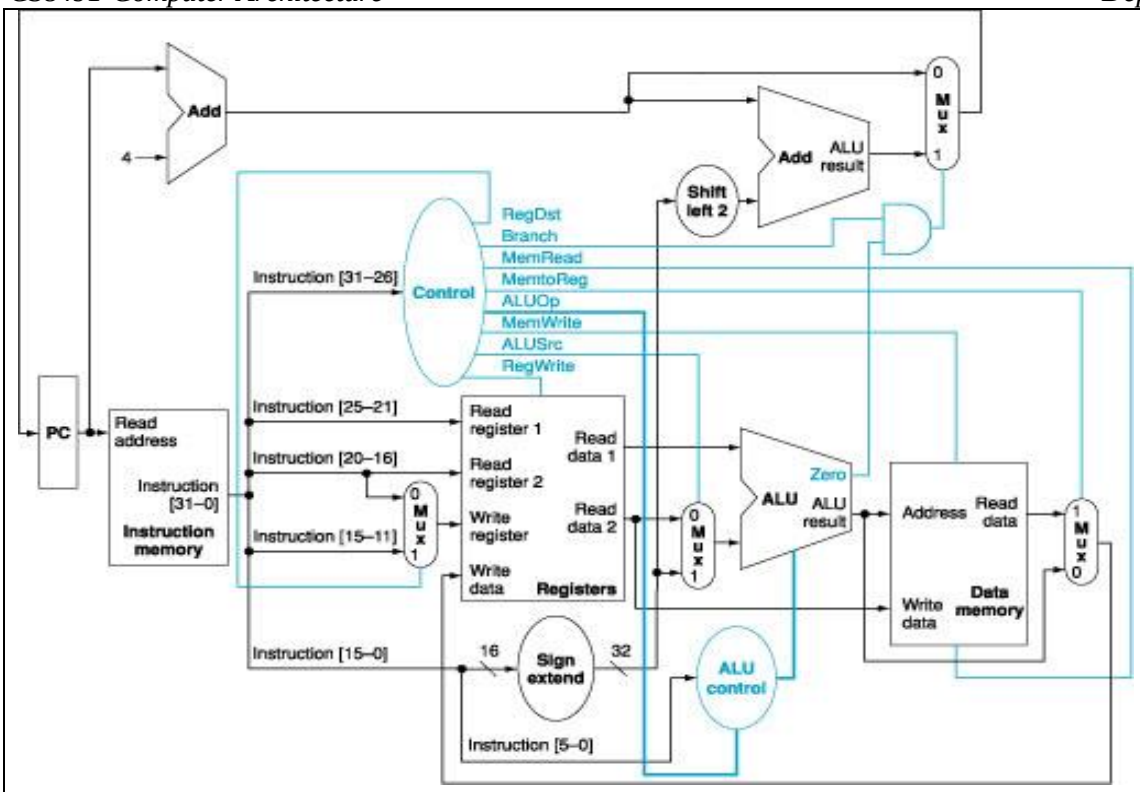


Fig: The effect of each of the seven control signals

When the 1bit control to a two way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the 0 input. Remember that the state elements all have the clock as an implicit input and that the clock is used in controlling writes. Gating the clock externally to a state element can create timing problems

The setting of the control lines is completely determined by the opcode fields of the instruction.

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

- The first row of the table corresponds to the **R-format instructions** (add, sub, AND, OR, and slt). For all these instructions, the source register fields are rs and rt, and the destination register field is rd; this defines how the signals ALUSrc and RegDst are set.
- Furthermore, an R-type instruction writes a register (RegWrite = 1), but neither reads nor writes data memory. When the Branch control signal is 0, the PC is unconditionally replaced with PC + 4; otherwise, the PC is replaced by the branch target if the Zero output of the ALU is also high. The ALUOp field for R-type instructions is set to 10 to indicate that the ALU control should be generated from the funct field.
- The second and third rows of this table give the control signal settings **for lw and sw**. These ALUSrc and ALUOp fields are set to perform the address calculation. The MemRead and MemWrite are set to perform the memory access. Finally, RegDst and RegWrite are set for a load to cause the result to be stored into the rt register.

The **branch instruction** is similar to an R-format operation, since it sends the rs and rt registers to the ALU. The ALUOp field for branch is set for a subtract (ALU control = 01), which is used to test for equality. Notice that the MemtoReg field is irrelevant when the RegWrite signal is 0: since the register is not being written, the value of the data on the register data write port is not used. Thus, the entry MemtoReg in the last two rows of the table is replaced with X for don't care. Don't cares can also be added to RegDst when RegWrite is 0. This type of don't care must be added by the designer, since it depends on knowledge of how the datapath works.

BUILDING DATAPATH

There are two basic types of logic components: combinational components and state components.

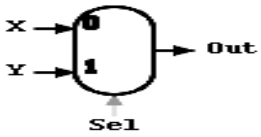
- A combinational component behaves like a simple function. Its outputs depend only on the current values of its inputs.
- A state component behaves more like an object method. Its output also depends on the past history of its inputs.

Combinational Components

The following are the combinational components in the diagram.

- Multiplexers
- Sign Ext
- Adder
- ALU
- Control
- ALU Control
- PC Update Control

1.1 Multiplexers



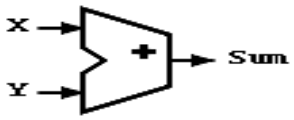
A multiplexer is a logic component with multiple data inputs (X and Y), a single control input (Sel), and a single output (Out). At any time its output is the same as one of its inputs which is determined by the control input. The number of bits in the output signal can in principle be any number. Each data input has the same number of bits. A multiplexer can in principle have any number of data inputs. The control signal has as many bits as needed for the selection. One bit suffices for just 2 data inputs, 2 bits suffices for 3 or 4 data inputs, and 3 bits suffices for 5 to 8 data inputs. In MIPS Single-Cycle Diagram, the three multiplexers in the MIPS diagram all have just two 32-bit data inputs and their outputs are the same size. Since they have only two data inputs, their control inputs are just one bit.

1.2 Sign Extension

The Sign Ext component performs sign extension, converting a 16-bit 2's complement number to a 32-bit 2's complement number. The low-order 16 bits of the output are the same as the input. The high-order 16 bits of the output are all copies of the sign (high-order) bit of the input.

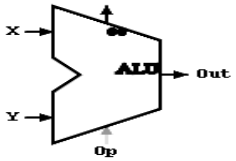
1.3 Adder

An adder just performs binary addition of its two inputs to produce a sum output.



ALU

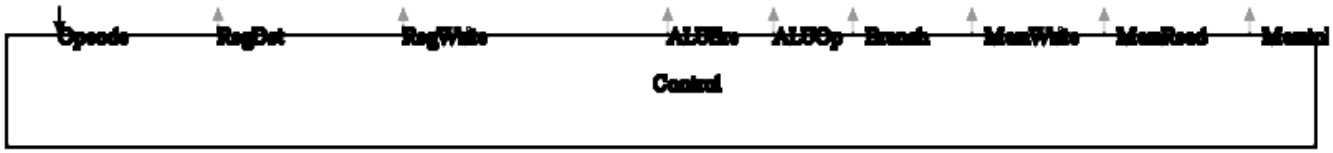
The ALU (Arithmetic-Logic Unit) can perform a number of different operations, combining its two data inputs (X and Y) into a single primary output (Out). Typical operations include additions, subtractions, and bitwise logical operations. The ALU also has a secondary output (cc) that encodes a comparison of its inputs. It is a 2-bit signal. One bit is true when X = Y, the other when X > Y. The cc output is only valid when the ALU is directed to do a subtraction.



The operation performed by the ALU is determined by a control signal (Op). For comparisons the control signal usually directs the ALU to do a subtraction.

1.5 Control

The (main) control component uses the opcode field of the instruction to generate most of the control signals.

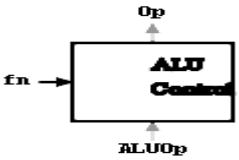


1.6 ALU Control

All op bits for an R-type instruction are 0. The operation is encoded into the fn bits. For R-type instructions, the main Control component delegates the determination of the ALU operation to the ALU Control.

1.7 Operation Control

For non-R-type instructions, the ALU Control just passes the ALUOp control from the main Control to the ALU. For R-type instructions the main Control sends a special code to the ALU Control. This code directs the ALU Control to use the fn bits to determine the operation.

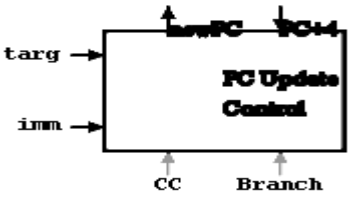


PC Update Control

The PC Update Control component handles program counter update as directed by a Branch control signal and a comparison code (CC) signal from the ALU. It selects among addresses constructed from the incremented PC (PC+4), the instruction imm field, and the instruction targ field.

MIPS Single-Cycle Diagram

The comparison code is also a single bit: the zero output of the ALU, which indicates equality when the ALU is directed to subtract. The implementation in following Figure only supports a branch on equality and no jumps. In Figure, jumps are supported by adding a Jump control signal. We can view this as converting the Branch signal into a 2-bit signal, though perhaps it should be renamed. Supporting a variety of branch conditions requires additional information from the ALU about the comparison and additional Branch control bits to indicate when branching should occur.



2. State Components

2.1 The Program Counter (PC)

The PC is just a simple 32-bit register. Its output (pc) is used as the address for Instruction Memory. It also feeds into an adder to generate the default pc+4 value address of the next instruction. This is one of the several inputs to the PC Update Control that it uses to generate its newpc output.

Program Counter Update

When the clock (not shown) starts a new cycle pc changes to match the newpc input. The newpc input is generated by the PC Update Control circuitry.

2.2 Registers

The Registers component is a register bank — a component that contains multiple registers and provides read and write access to them.

In the MIPS processor there are 32 registers in the Registers component. Each register consists of 32 flip-flops, each storing 1 bit of data. In addition the component contains two read ports and one write port.

Read Port

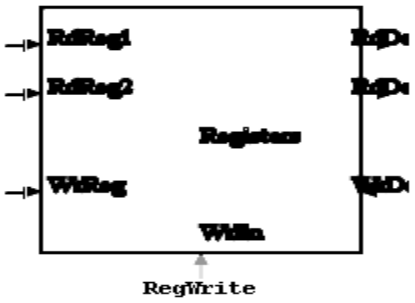
The input signal RdReg1 and the output signal RdData1 make up the first read port. The input signal RdReg2 and the output signal RdData2 make up the second read port.

- The input signals, RdReg1 and RdReg2, are 5-bit signals that specify a register number.
- The output signals, RdData1 and RdData2, are 32-bit signals. Their values are the contents of the register specified by the port input signal.

Write Port

The input signals, WrReg and WrData, and the control input signal WrEn make up the write port.

- WrReg specifies the register number for the register to be written.
- WrData specifies the data to be written to the register.
- WrEn enables the write when its value is 1. The write does not happen if WrEn is 0. This signal is necessary because not all instructions write to a register.



2.3Data Memory

The Data Memory component is actually just an interface to the bus that interconnects the processor, main memory, and I/O devices. Since other devices use this bus, an enable signal is required for both memory reads and memory writes.

Note that the Address input is shared by both the read port and the write port.

Read Port

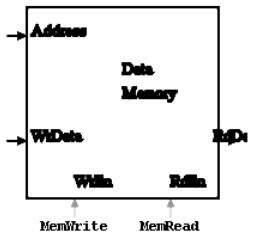
The input signal, Address, the output signal, RdData, and the control input signal RdEn make up the read port.

- The input signal Address is a 32-bit signal that specifies a memory address.
- The output signal RdData is a 32-bit signal. Its value is the data at the address specified by the Address input signal.
- RdEn enables the read when its value is 1. The write does not happen if RdEn is 0. This signal is necessary to ensure the processor does not access the bus except when it needs to.

Write Port

The input signals, Address and WrData, and the control input signal WrEn make up the write port.

- The input signal Address is a 32-bit signal that specifies a memory address.
- WrData specifies the data to be written at that address.
- WrEn enables the write when its value is 1. The write does not happen if WrEn is 0. This signal is necessary to ensure the processor does not access the bus except when it needs to.



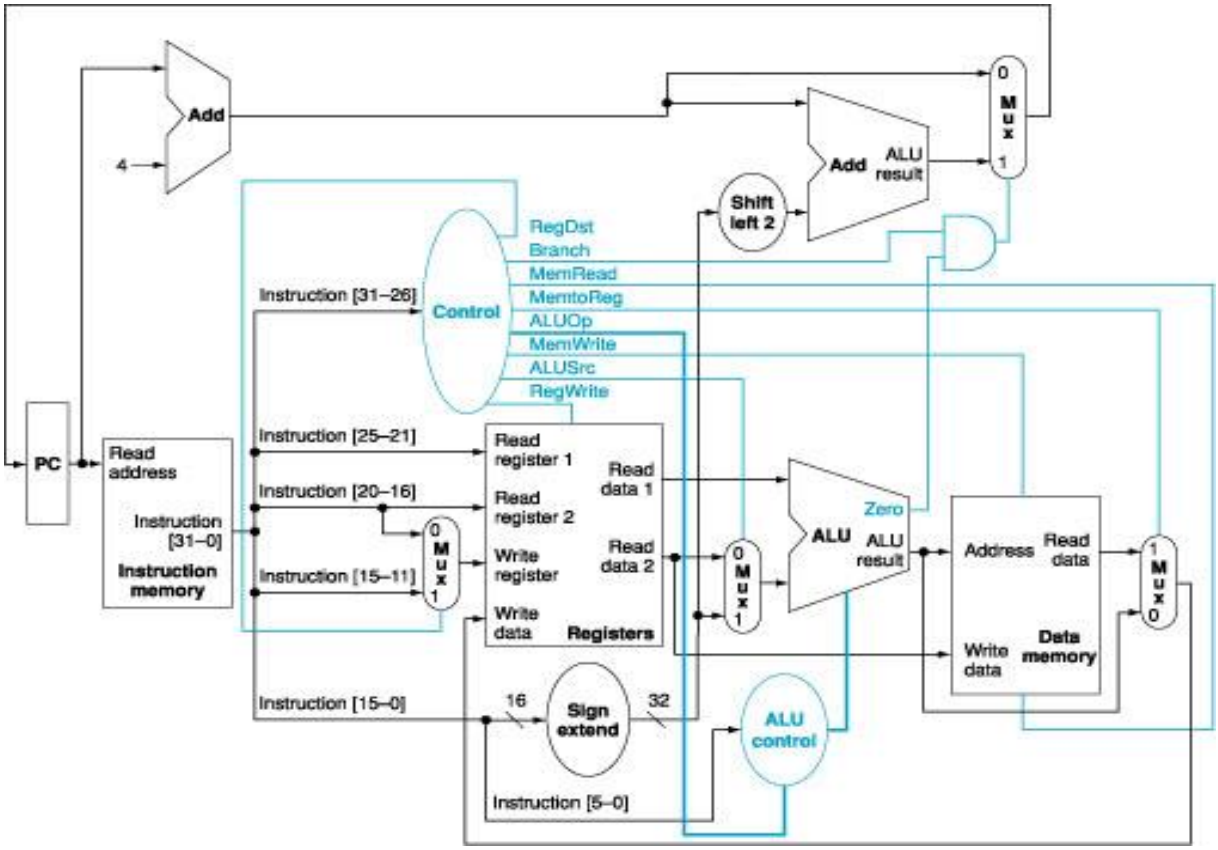
2.4 Instruction Memory

Instruction Memory has a simple task: given a memory address input, fetch the instruction at that address. Consequently, the interface to Instruction Memory is just a single read port. The input signal Address is a 32-bit signal that specifies an instruction address. The output signal Instruction is a 32-bit signal. Its value is the instruction at the address specified by the Address input signal. For convenience, the instruction is then broken up into its fields. Instruction Memory does not show a read enable because it is automatic — in a single-cycle implementation an instruction is read every cycle.

DATA PATH AND ITS CONTROL

We can design the control unit logic, but before we do that, let's look at how each instruction uses the datapath. In datapath of an **R-type instruction**, such as **add $t1,$t2,$t3**, everything occurs in one clock cycle, we can think of four steps to execute the instruction; these steps are ordered by the flow of information:

1. The instruction is fetched, and the PC is incremented.
2. Two registers, $t2 and $t3, are read from the register file; also, the main control unit computes the setting of the control lines during this step.
3. The ALU operates on the data read from the register file, using the function code (bits 5:0, which is the funct field, of the instruction) to generate the ALU function.
4. The result from the ALU is written into the register file using bits 15:11 of the instruction to select the destination register ($t1).



Similarly, we can illustrate the execution of a **load word**, such as **lw $t1, offset($t2)**

1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. A register ($t2) value is read from the register file.
3. The ALU computes the sum of the value read from the register file and the sign-extended, lower 16 bits of the instruction (offset).
4. The sum from the ALU is used as the address for the data memory.
5. The data from the memory unit is written into the register file; the register destination is given by bits 20:16 of the instruction ($t1).

Finally, we can show the operation of the branch-on-equal instruction, such as **beq $t1,$t2,offset**, in the same fashion. It operates much like an R-format instruction, but the ALU output is used to determine

whether the PC is written with PC + 4 or the branch target address.

1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. Two registers, $t1 and $t2, are read from the register file.
3. The ALU performs a subtract on the data values read from the register file. The value of PC + 4 is added to the sign-extended, lower 16 bits of the instruction (offset) shifted left by two; the result is the branch target address.
4. The Zero result from the ALU is used to decide which adder result to store into the PC.

The simple datapath with the control unit :

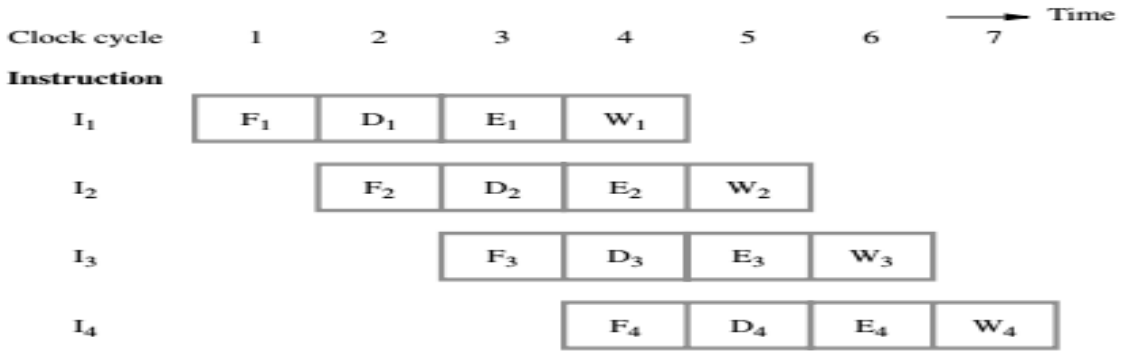
The input to the control unit is the 6-bit opcode field from the instruction. The outputs of the control unit consist of three 1-bit signals that are used to control multiplexors (RegDst, ALUSrc, and MemtoReg), three signals for controlling reads and writes in the register file and data memory (RegWrite, MemRead, and MemWrite), a 1-bit signal used in determining whether to possibly branch (Branch), and a 2-bit control signal for the ALU (ALUOp). An AND gate is used to combine the branch control signal and the Zero output from the ALU; the AND gate output controls the selection of the next PC. Notice that PCSrc is now a derived signal, rather than one coming directly from the control unit.

INSTRUCTION PIPELINE WORKS

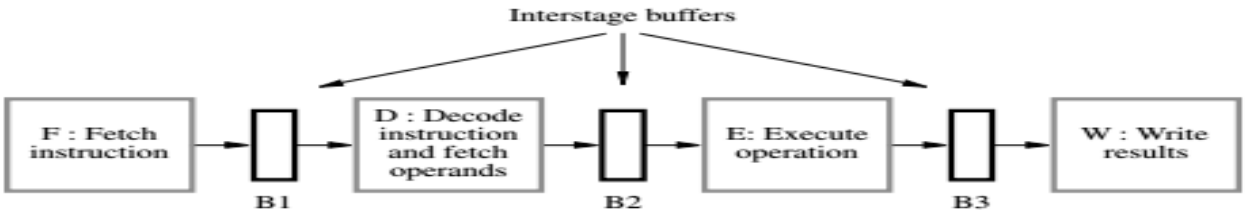
A pipelined computer executes instructions concurrently.

Consider a pipeline processor, which process each instruction in four steps;

F: Fetch, Read the instruction from the memory
D: decode, decode the instruction and fetch the source operand (S)
O: Operate, perform the operation
W: Write, store the result in the destination location



(a) Instruction execution divided into four steps



(b) Hardware organization

Hardware units are organized into stages:

- Execution in each stage takes exactly 1 clock period.
- Stages are separated by pipeline registers that preserve and pass partial results to the next stage.

Performance = complexity + cost.

The pipeline approach brings additional expense plus its own set of problems and complications, called hazards.

Issues affecting pipeline performance

1. Pipelining is an implementation technique in which multiple instructions are overlapped in execution. It is an important technique used to make fast CPUs which takes much less time to complete the execution when compared to sequential execution technique.

2. A substantial improvement in performance can be achieved by overlapping the execution of successive instructions using a technique called pipelining. Pipelining is an effective way of organizing concurrent activity in a computer system.
3. In a computer pipeline, each step in the pipeline completes the execution of a part of the instruction. Different steps are completing different instructions in parallel. Each of these steps is called a **pipeline stage** (or) pipeline segment.
4. Pipelining yields a reduction in the average execution time per instruction. It is an implementation technique that exploits parallelism among the instructions in a sequential instruction stream. Pipelining would not decrease the time required to complete the execution of one instruction, but when we have multiple instructions to execute, the improvement in throughput decreases the total time to complete the execution of the entire program.

Pipeline Performance (or) Speedup

The potential increase in performance resulting from pipelining is proportional to the number of pipeline stages. If all the stages take about the same amount of time and there is enough work to do, then the speed-up due to pipelining is equal to the number of stages in the pipeline.

If the stages are perfectly balanced, then the time between instructions on the pipelined processor – assuming ideal conditions – is equal to

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{non-pipelined}}}{\text{Number of pipeline stages}}$$

Under ideal conditions and with a large number of instructions, the speed-up from pipelining is approximately equal to the number of pipeline stages; a five-stage pipeline is nearly five times faster.

Pipelining improves performance by increasing instruction throughput, as opposed to decreasing the execution time of an individual instruction, but instruction throughput is the important metric because real programs execute billions of instructions.

MIPS PROCESSOR – PIPELINE STAGES

The same principles apply to processors where the pipeline instructions are in execution. MIPS instructions classically take five steps:

1. Fetch instruction from memory
2. Read registers while decoding the instruction. The regular format of MIPS instructions allows reading and decoding to occur simultaneously
3. Execute the operation or calculate an address
4. Access an operand in data memory
5. Write the result into a register

A pipelined processor allows multiple instructions to execute at once, and each instruction uses a different functional unit in the datapath. This increases throughput, so programs can run faster. One instruction can finish executing on every clock cycle, and simpler stages also lead to shorter cycle times.

Example – Single-Cycle versus Pipelined Performance

Consider a simple program segment consists of eight instructions: lw, sw, add, sub, AND, OR, slt and beq. Compare the average time between instructions of a single-cycle implementation, in which all instructions take 1 clock cycle, to a pipelined implementation. The operation times for the major functional units in this example are 200 ps for memory access, 200 ps for ALU operation, and 100 ps for register file read or write.

Figure 3.29 shows the time required for each of the eight instructions. The single-cycle design must allow for the slowest instruction – in Figure 3.29 it is lw – so the time required for every instruction is 800 ps. Thus, the time between the first and fourth instructions in the non-pipelined design is 3 x 800 ns or 2400 ps.

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (`lw`)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (`sw`)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (`add`, `sub`, `AND`, `OR`, `slt`)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (`beq`)	200 ps	100 ps	200 ps			500 ps

Figure 3.29 Total time for each instruction calculated from the time for each component

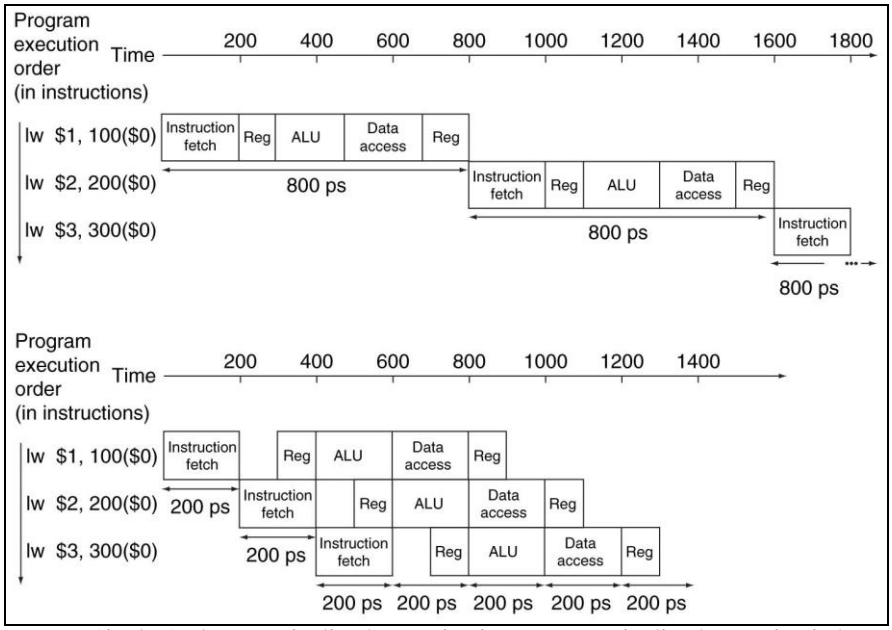


Figure 3.30 Single-Cycle, Non-Pipelined Execution in top versus Pipelined Execution in bottom.

All the pipeline stages take a single clock cycle, so the clock cycle must be long enough to accommodate the slowest operation. Just as the single-cycle design must take the worst-case clock cycle of 800 ps, even though some instructions can be as fast as 500 ps, the pipelined execution clock cycle must have the worst-case clock cycle of 200 ps, even though some stages take only 100 ps. Pipelining still offers a fourfold performance improvement; the time between the first and fourth instructions is 3 x 200 ps or 600 ps.

Pipelining suggests that a five-stage pipeline should offer nearly a fivefold improvement over the 800 ps non-pipelined time, or a 160 ps clock cycle. The above example shows that the stages may be imperfectly balanced.

PIPELINED DATAPATH & its CONTROL

The goal of pipelining is to allow multiple instructions execute at the same time. It may need to perform several operations in a clock cycle such as increment the PC and add registers at the same time, fetch one instruction while another one reads or writes data, etc,.

Pipelined Datapath

The division of an instruction into five stages means a five-stage pipeline, which in turn means that up to five instructions will be in execution during any single clock cycle. Thus, it must separate the datapath into five pieces, with each piece named corresponding to a stage of instruction execution.

1. IF: Instruction Fetch
2. ID: Instruction decode and register file read
3. EX: Execution or Address Calculation
4. MEM: Data Memory Access
5. WB: Write Back

In figure 3.1, these five components correspond roughly to the way the datapath is drawn; instructions and data move generally from left to right through the five stages as they complete execution.

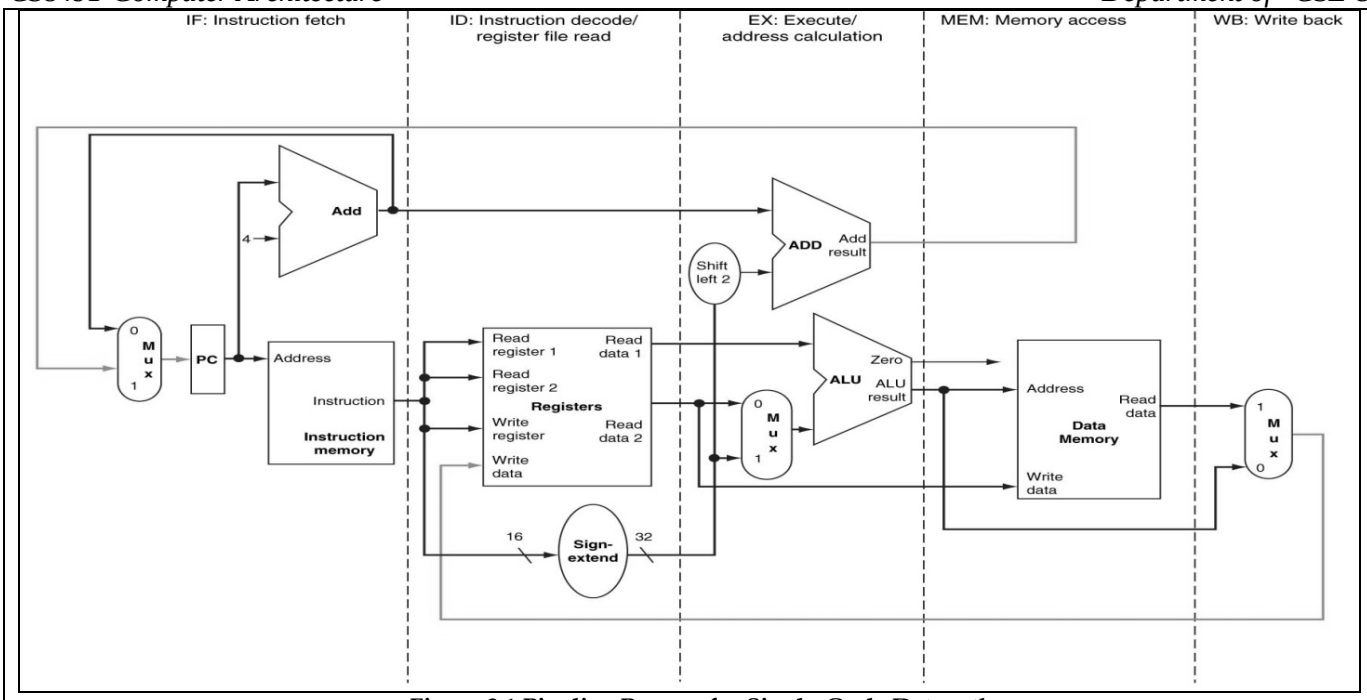


Figure 3.1 Pipeline Process for Single-Cycle Datapath

There are however, two exceptions to this left-to-right flow of instructions:

- The write-back stage, which places the result back into the register file in the middle of the datapath.
- The selection of the next value of the PC, choosing between the incremented PC and the branch address from the MEM stage.

Data flowing from left does not affect the current instruction; only later instructions in the pipeline are influenced by these reverse data movements. Note that the first right-to-left flow of data can lead to data hazards and the second leads to control hazards.

One way to show what happens in pipelined execution is to pretend that each instruction has its own datapath, and then to place these data paths on a timeline to show their relationship. Figure 3.2 shows the execution of the instructions by displaying their private data paths on a common timeline.

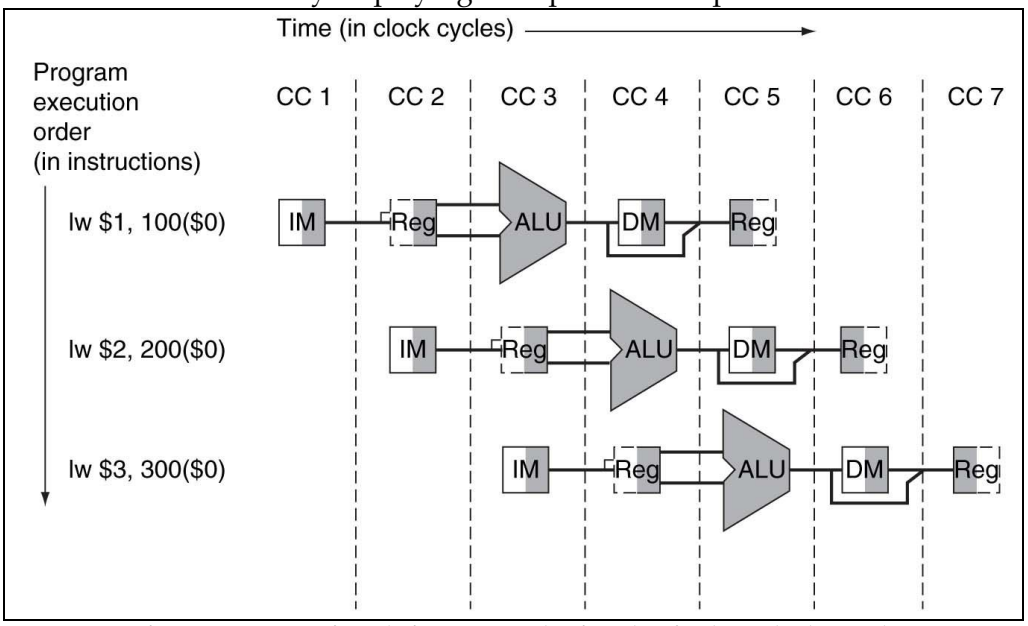


Figure 3.2 Instructions being executed using the single-cycle datapath

This figure seems to suggest that three instructions need three datapaths. Instead, add registers to hold data so that portions of a single datapath can be shared during instruction execution. The instruction memory is used during only one of the five stages of an instruction, allowing it to be shared by following instructions during the other four stages. To retain the value of an individual instruction for its other four stages, the value read from instruction memory must be saved in a register. Similar

arguments apply to every pipeline stage must be placed in registers.

Figure 3.3 shows the pipelined datapath with the pipeline registers highlighted. All instructions advance during each clock cycle from one pipeline register to the next. The registers are named for the two stages separated by that register. For example, the pipeline register between the IF and ID stages is called IF/ID.

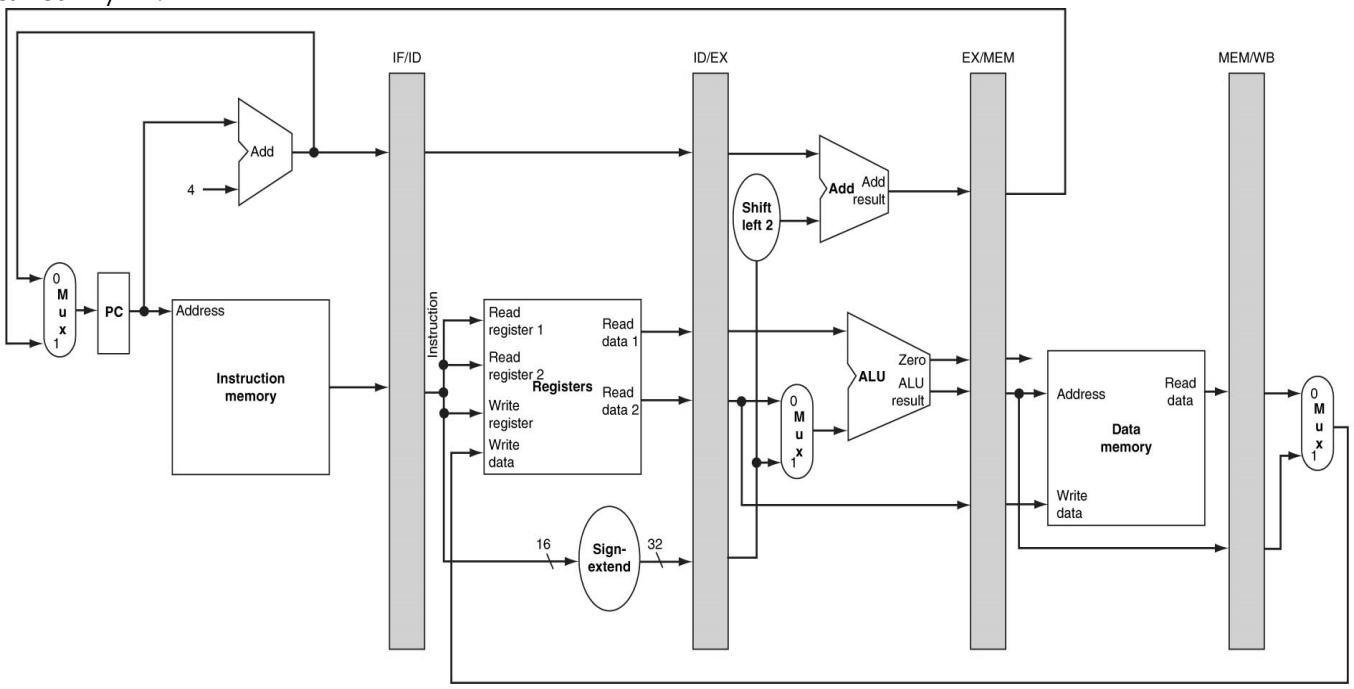


Figure 3.3 The pipelined version of the datapath

Notice that there is no pipeline register at the end of the write-back stage. All instructions must update some state in the processor – the register file, memory or the PC – so a separate pipeline register is redundant to the state that is updated. Every instruction updates the PC, whether by incrementing it or by setting to a branch target address. The PC can be thought of as a pipeline register; one that feeds the IF stage of the pipeline.

Unlike the shaded pipeline registers in the above figure, however, the PC is part of the visible architectural state, its contents must be saved when an exception occurs, while the contents of the pipeline registers can be discarded.

Examples:

Pipelined Datapath for a Load Instruction:

The following sequence of diagrams shows the active portions of the datapath highlighted as a load instruction goes through the five stages of pipelined execution. Load is shown first because it is active in all five stages. It is highlight the right half of registers or memory when they are being read and highlight the left half when they are being written.

The five stages of the load instruction are the following:

1. **Instruction Fetch:**

Figure 3.34 shows the instruction being read from memory using the address in the PC and then being placed in the IF/ID pipeline register. The PC address is incremented by 4 and then written back into the PC to be ready for the next clock cycle. This incremented address is saved in the IF/ID pipeline register in case it is needed later for an instruction, such as beq. The computer cannot know which type of instruction is being fetched, so it must prepare for any instruction, passing potentially needed information down the pipeline.

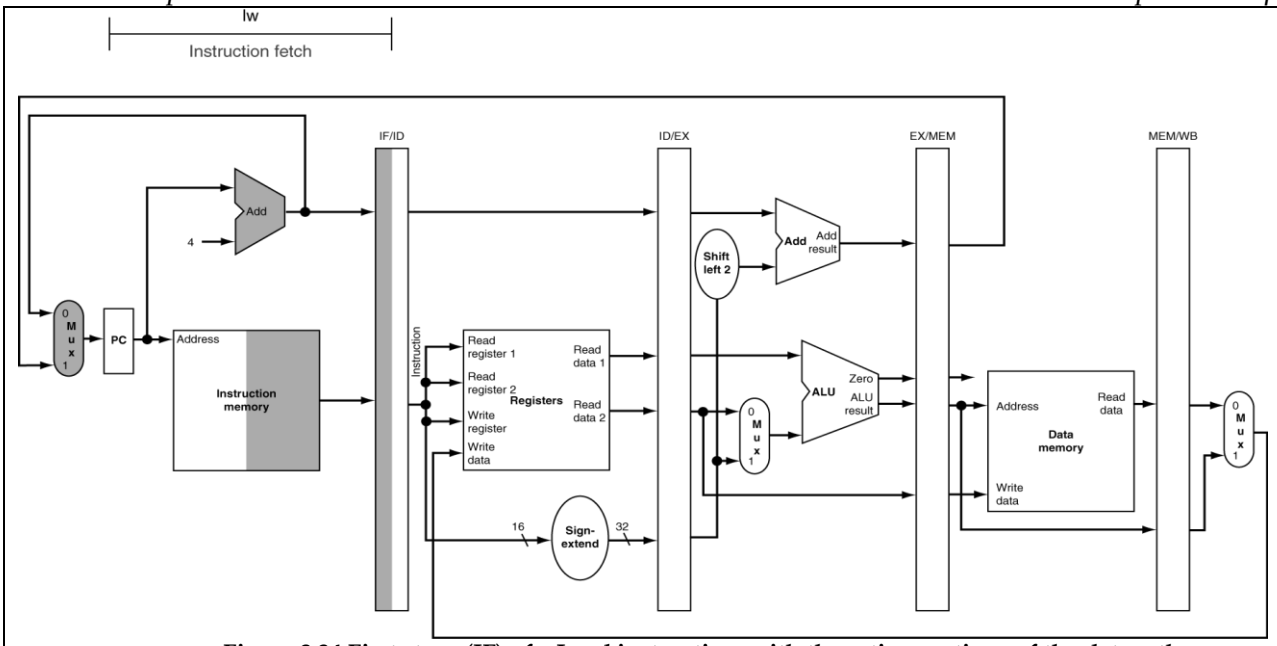


Figure 3.34 First stage (IF) of a Load instruction, with the active portions of the datapath

2. Instruction Decode and Read Register File Read: Figure 3.35 shows the instruction decode portion of the IF/ID pipeline register supplying the 16-bit immediate field, which is sign-extended to 32-bits, and the register numbers to read the two registers. All three values are stored in the ID/EX pipeline register, along with the incremented PC address. We again transfer everything that might be needed by any instruction during a later clock cycle.

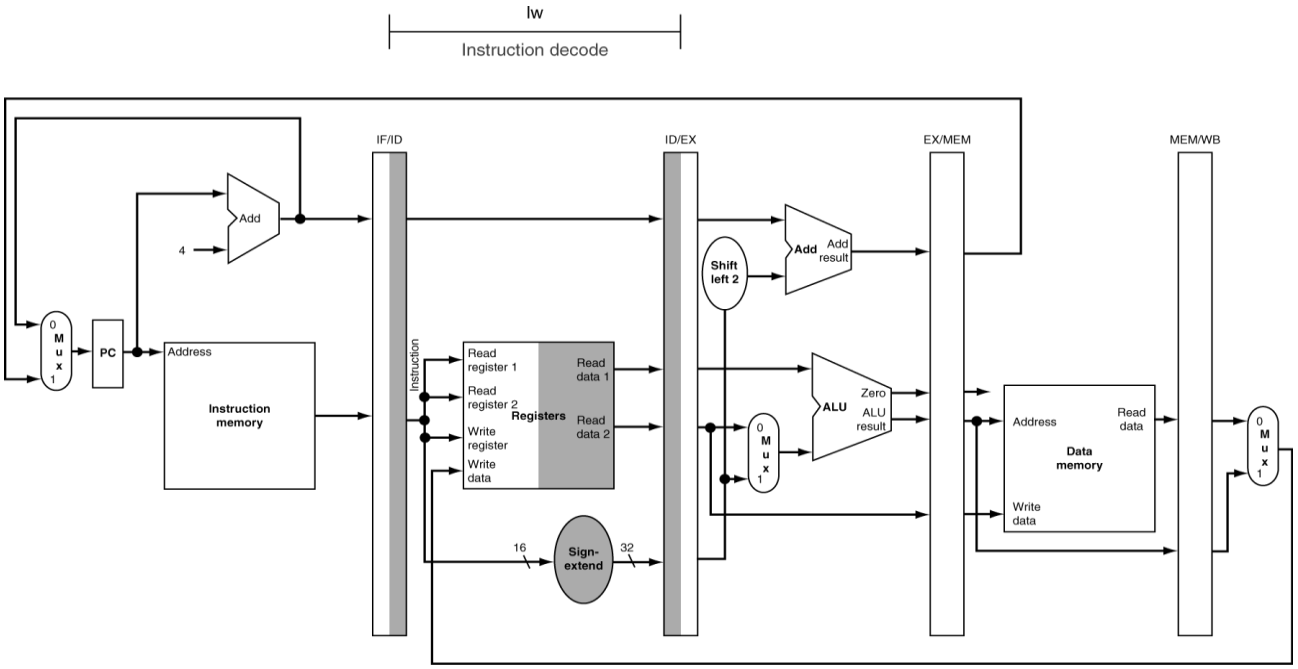


Figure 3.35 Second stage (ID) of a Load instruction, with the active portions of the datapath in Figure 3.33 highlighted

1. Execute or Address Calculation: Figure 3.36 shows that the load reads the contents of register 1 and the sign-extended immediate from the ID/EX pipeline register and adds them using the ALU. That sum is placed in the EX/MEM pipeline register.

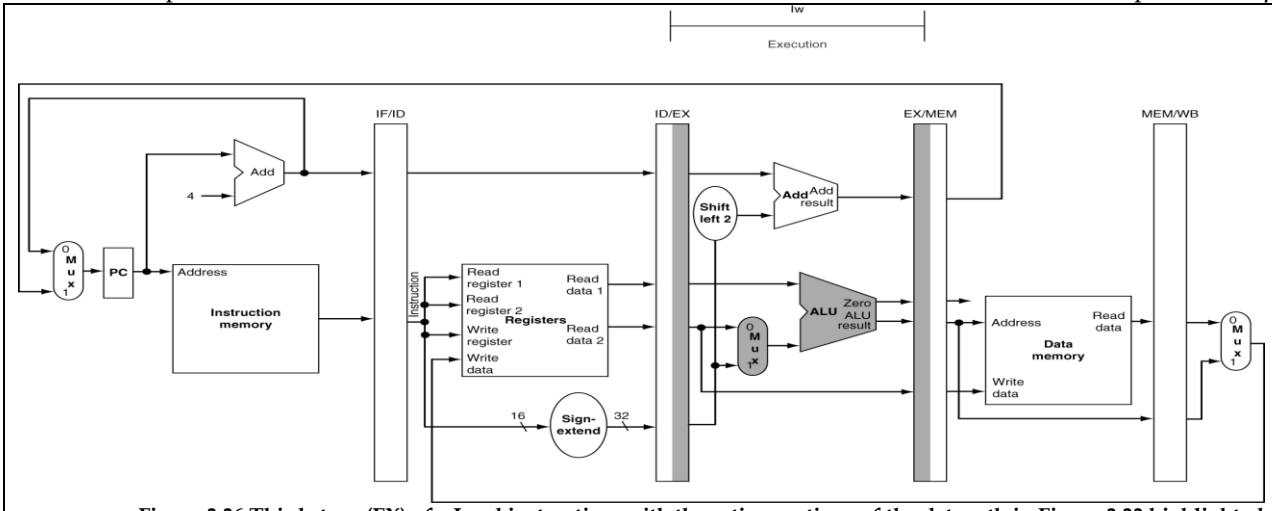


Figure 3.36 Third stage (EX) of a Load instruction, with the active portions of the datapath in Figure 3.33 highlighted

2. Memory Access: Figure 3.37 shows the load instruction reading the data memory using the address from the EX/MEM pipeline register and loading the data into the MEM/WB pipeline register.

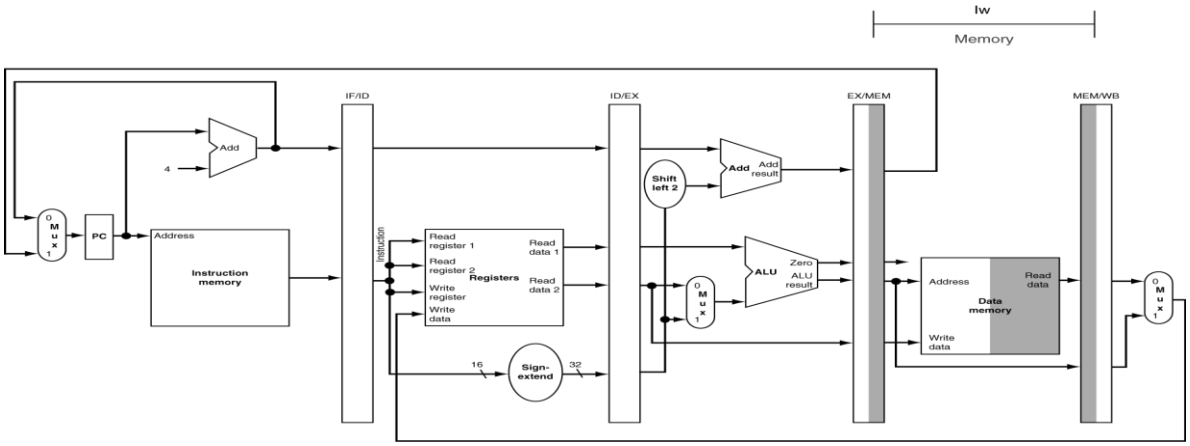


Figure 3.37 Fourth stage (MEM) of a Load instruction, with the active portions of the datapath in Figure 3.33 highlighted

3. Write-Back: Figure 3.38 shows the final step – reading the data from the MEM/WB pipeline register and writing it into the register file in the middle of the figure.

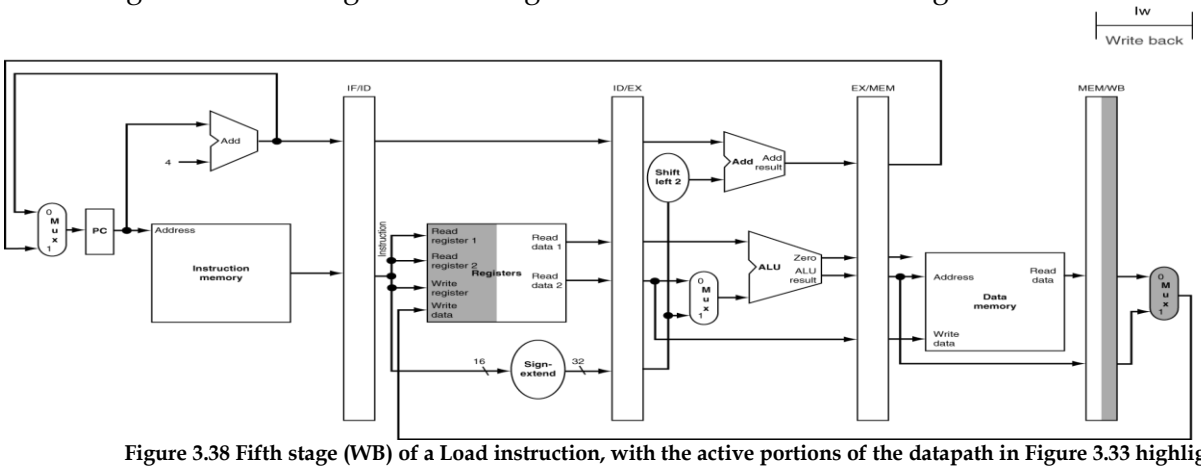


Figure 3.38 Fifth stage (WB) of a Load instruction, with the active portions of the datapath in Figure 3.33 highlighted

Pipelined Datapath for a Store Instruction:

Walking through a store instruction shows the similarity of instruction execution, as well as passing the information for later stages. Here are the five pipe stages of the store instruction:

1. **Instruction Fetch:** the instruction is read from memory using the address in the PC and then is placed in the IF/ID pipeline register. This stage occurs before the instruction is identified, so the figure 3.34 works for store as well.

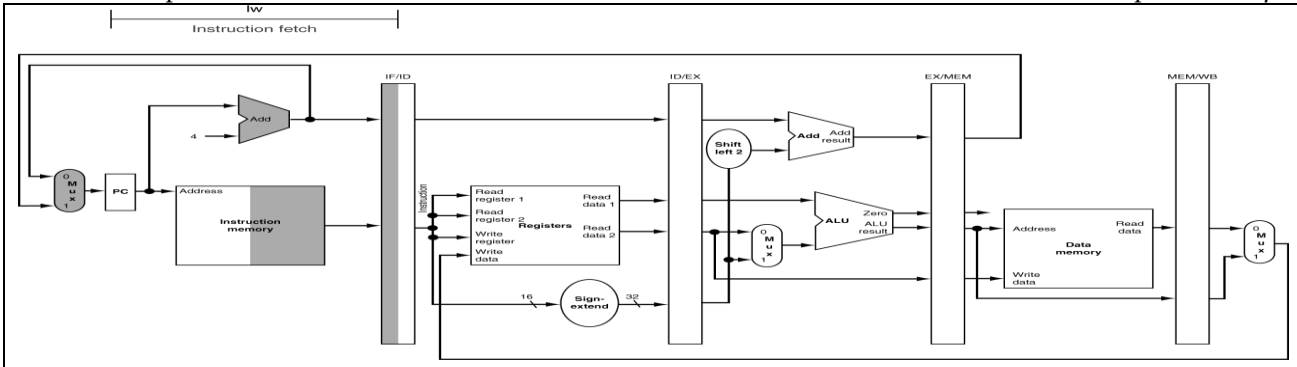


Figure 3.39 First stage (IF) of a Store instruction, with the active portions of the datapath in Figure 3.33 highlighted – same as Figure 3.34

2. **Instruction Decode and Read Register File:** The instruction in the IF/ID pipeline register supplies the register numbers for reading two registers and extends the sign of the 16-bit immediate. These three 32-bit values are all stored in the ID/EX pipeline register. Figure 3.35 for load instruction also shows the operations of the second stage for stores.

These first two stages are executed by all instructions, since it is too early to know the type of the instruction.

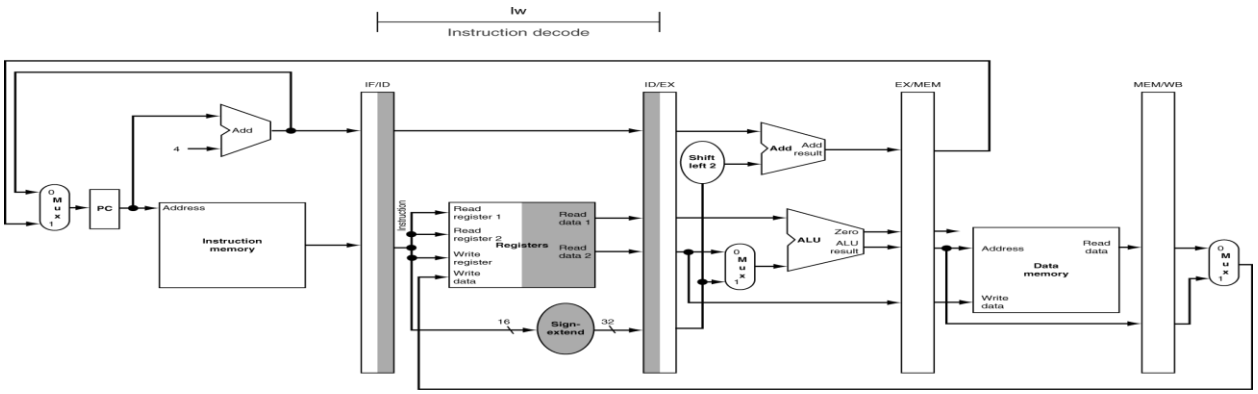


Figure 3.40 Second stage (ID) of a Store instruction, with the active portions of the datapath in Figure 3.33 highlighted – same as Figure 3.35

3. **Execute and Address Calculation:** Figure 3.41 shows the third step of a store instruction in which the effective address is placed in the EX/MEM pipeline register.

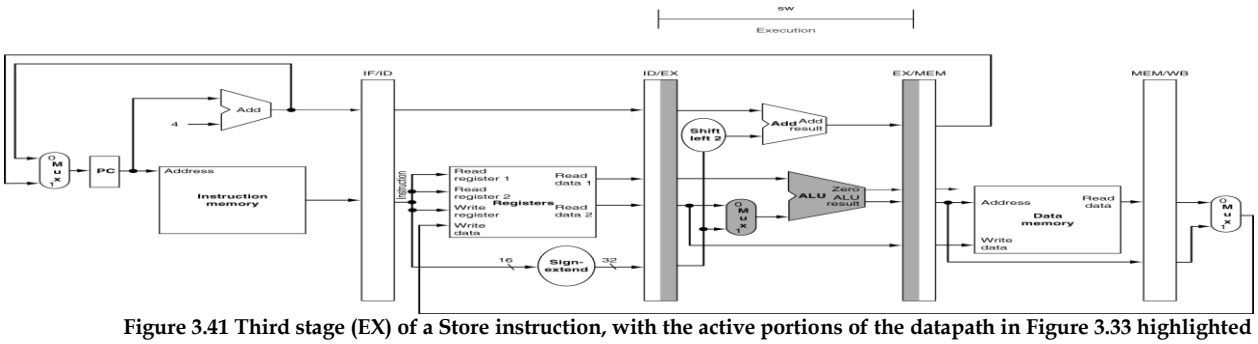


Figure 3.41 Third stage (EX) of a Store instruction, with the active portions of the datapath in Figure 3.33 highlighted

4. **Memory Access:** Figure 3.42 shows the data being written to memory. Note that the register containing the data to be stored was read in an earlier stage and stored in ID/EX. The only way to make the data available during the MEM stage is to place the data into the EX/MEM pipeline register in the EX stage, just as we stored the effective address into EX/MEM.

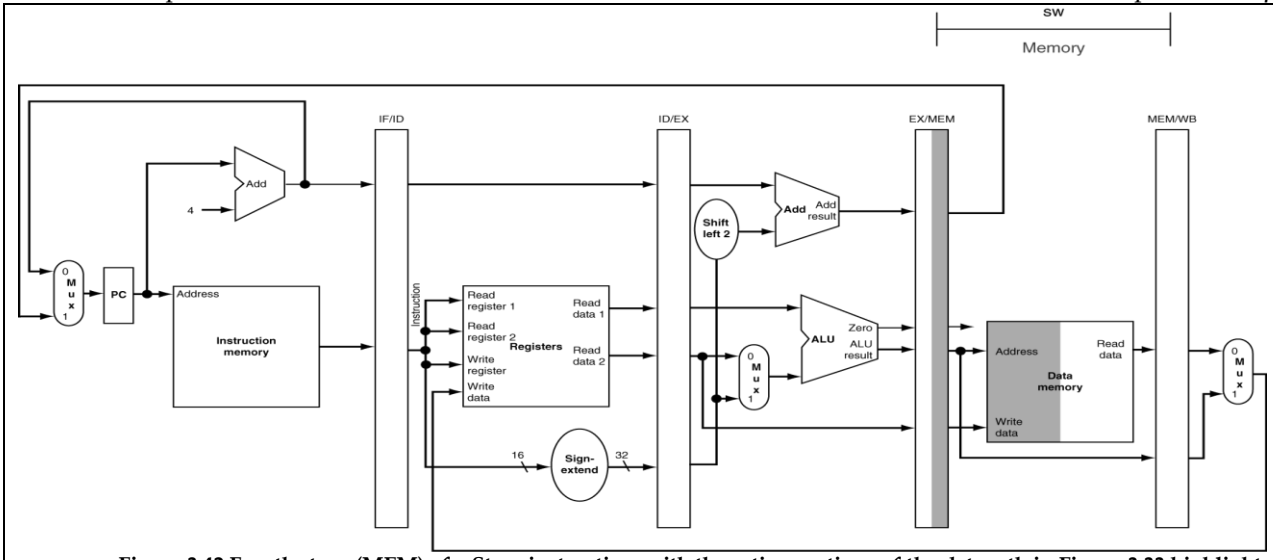


Figure 3.42 Fourth stage (MEM) of a Store instruction, with the active portions of the datapath in Figure 3.33 highlighted

5. **Write-Back:** Figure 3.43 how the final step of the store. For this instruction, nothing happens in the write-back stage. Since every instruction behind the store is already in progress, we have no way to accelerate those instructions. Hence, an instruction passes through a stage even if there is nothing to do, because later instructions are already progressing at the maximum rate.

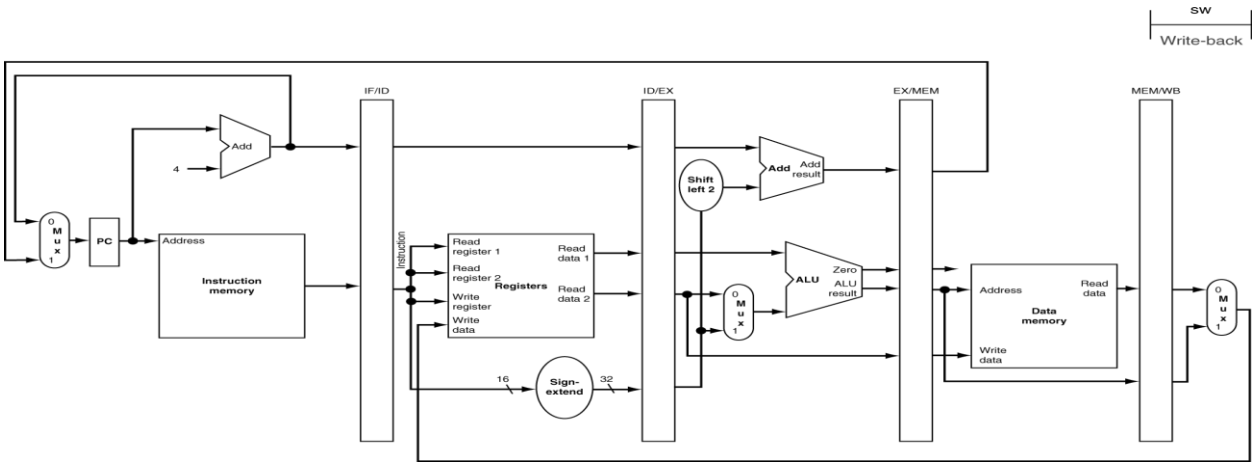


Figure 3.43 Fifth stage (WB) of a Store instruction, with the active portions of the datapath in Figure 3.33 highlighted

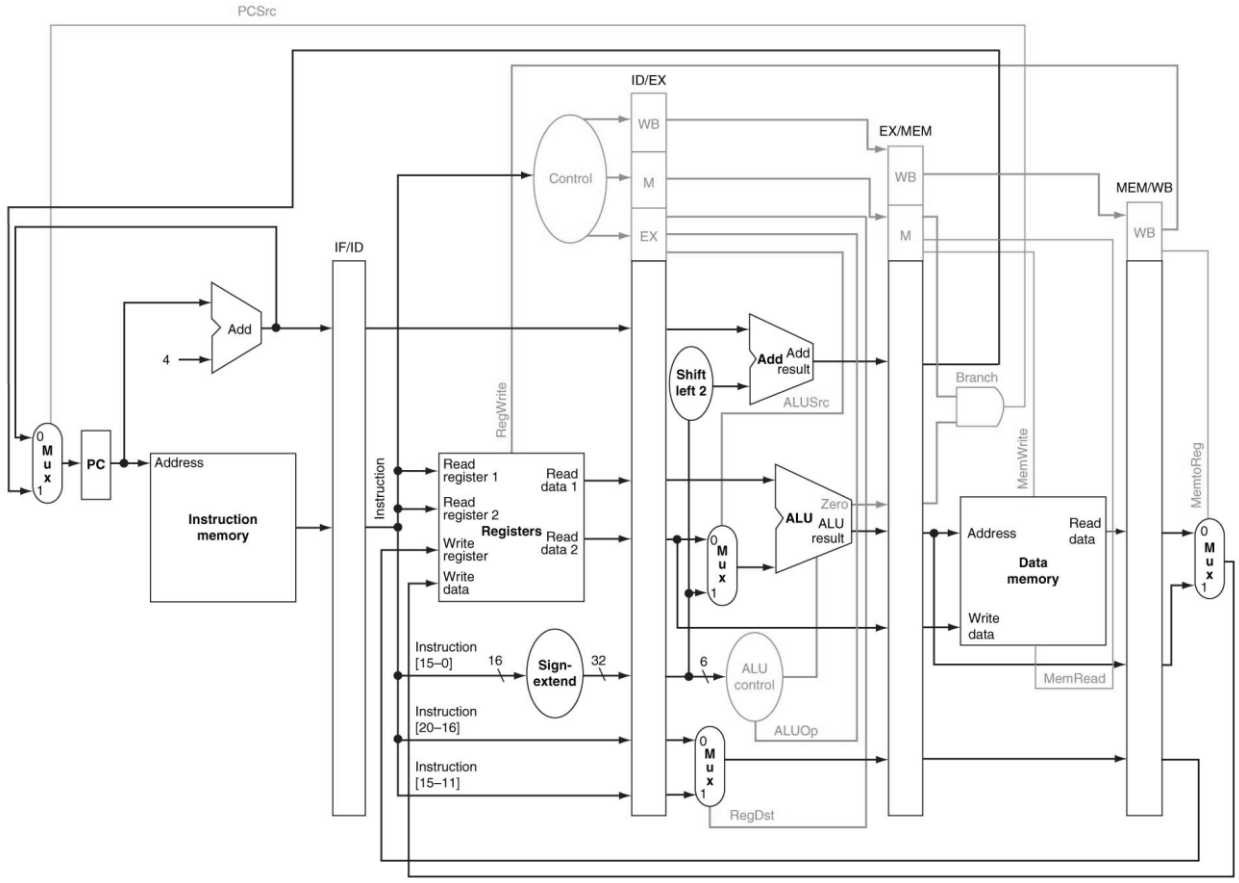
The store instruction again illustrates that to pass information from an early pipe stage to a later pipe stage, the information must be placed in a pipeline register; otherwise the information is lost when the next instruction enters that pipeline stage. For the store instruction we needed to pass one of the registers read in the ID stage to the MEM stage, where it is stored in memory. The data was first placed in the ID/EX pipeline register and then passed to the EX/MEM pipeline register. Both load and store illustrate a second key point: each logical component of the datapath – such as instruction memory, register read ports, ALU, data memory and register write port – can be used only within a single pipeline stage. Otherwise, we would have a structural hazard. Hence these components, and their control, can be associated with a single pipeline stage.

To provide control in a pipelined datapath, the following task has to be followed:

- The first step is to label the control lines on the existing datapath.
- Borrow/ Use the control lines from the simple datapath.
- Use the same ALU control logic, branch logic, destination-register-number multiplexor, and control lines.

Figure 3.49 shows the pipelined datapath of a MIPS processor with the control signal identified. In the case of a single-cycle implementation, we assume that the PC is written on each clock cycle, so

there is no separate write signal for the PC. By the same argument, there are no separate write signals for the pipeline registers (IF/ID, ID/EX. EX/MEM, and MEM/WB), since the pipeline registers are also written during each clock cycle.



The pipelined datapath of Figure 3.49, with the control signals connected to the control portions of the pipe line registers

HAZARD AND ITS TYPES

Types of Hazards

There are three types of hazards. They are

- Structural Hazards
- Data Hazards
- Control Hazards (or) Instruction Hazards

Structural Hazards

When a processor is pipelined, the overlapped execution of instructions requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline. If some combinations of instructions cannot be accommodated because of resources conflicts, the processor is said to have a structural hazard.

It is a situation in which two instructions require the use of a given hardware resource at the same time.

Example: Access to Memory

One instruction may need to access memory as part of the execute or write stage while another instruction is being fetched. If instruction and data resides in the same cache unit, then only one instruction can proceed and another instruction is delayed.

```
ld 50($s0), $s1;
add $s3, $s1, $s2;
```

The execution of these instructions causes the pipeline to stall for one clock cycle, because both instructions require access to the register file in the same clock cycle.Even though the instructions and their data are all available, the pipeline is stalled because one hardware resource (register file – s1) cannot handle two operations at same time.

Data Hazards

Data hazards arise when the execution of an instruction depends on the results of a previous instruction in the pipeline.

Consider the following sequence of instructions:

```
sub   $2, $1, $3
and   $1, $2, $5
or    $13, $6, $2
add   $14, $2, $2
sw    $15, 100($2)
```

Register $2 is used in all the five instruction, so last four instructions has to wait until the first instruction was executed. The last four instructions are all dependent on the result in register $2 of the first instruction. If register $2 had the value 10 before the subtract instruction and -20 afterwards, the programmer intends that -20 will be used in the following instructions that refer to register $2.

Figure 3.52 illustrates the execution of these instructions using a multiple clock-cycle pipeline representation. To demonstrate the execution of this instruction sequence in our current pipeline, the top of this figure shows the value of register $2, which changes during the middle of clock cycle 5, when the sub instruction writes its result.

The last potential hazard can be resolved by the design of the register file hardware: what happens when a register is read and written in the same clock cycle? Assume that the write is in the first half of the clock cycle and the read is in the second half, so the read delivers what is written. As is the case for many implementations of register files, there is no data hazard in this case.

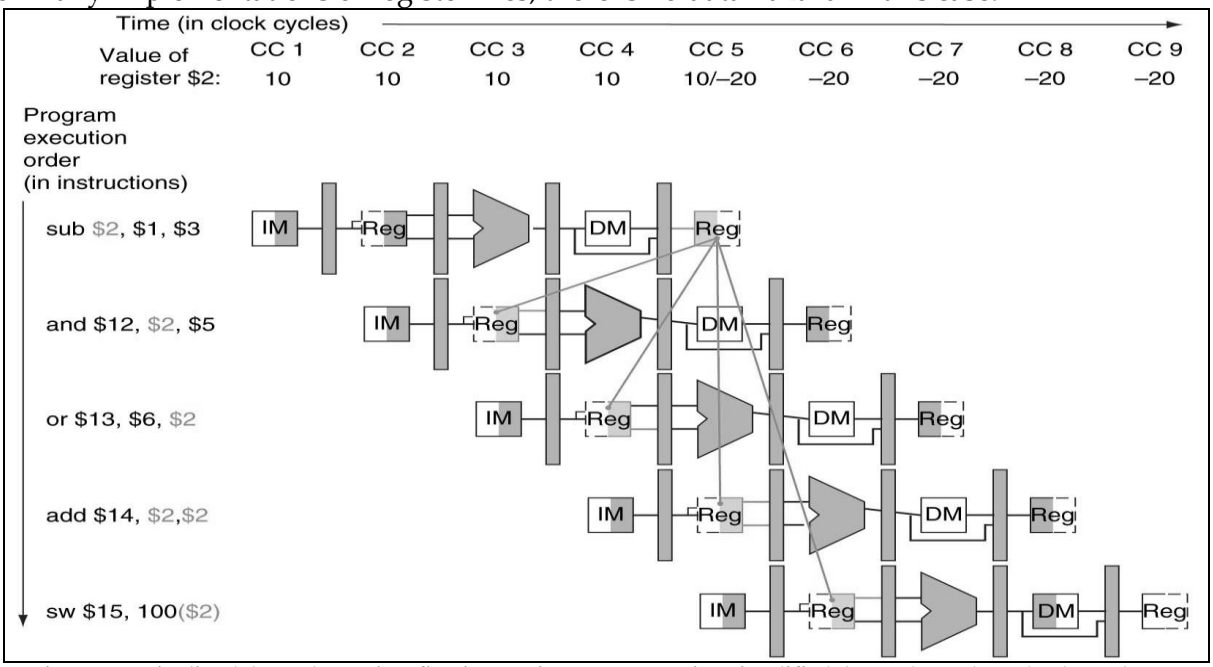


Figure 3.52 Pipelined dependences in a five-instruction sequence using simplified datapaths to show the dependences

This figure shows that the values read for register $2 would not be the result of the sub instruction unless the read occurred during clock cycle 5 or later. Thus, the instructions that would get the correct value of -20 are add and sw; the AND and OR instructions would get the incorrect value of 10. Using this style of drawing, such problems become apparent when a dependence line goes backward in time.

However, the desired result is available at the end of the EX stage or clock cycle 3. Thus, it is possible to execute this segment without stalls if simply forward the data as soon as it is available to any units that need it before it is available to read from the register file. Consider only the challenge of forwarding to an operation in the EX stage, which may be either an ALU operation or an effective address calculation. This means that when an instruction tries to use a register in its EX stage that an earlier instruction intends to write in its WB stage, actually need the values as inputs to the ALU.

A notation that names the fields of the pipeline registers allows for a more precise notation of dependences. For example, "ID/EX.RegisterRs" refers to the number of one register whose value is found in the pipeline register ID/EX; that is, the one from the first read port of the register file. The first part of the name, to the left of the period, is the name of the pipeline register; the second part is the name

of the field in that register.

Using this notation, the two pairs of hazard conditions are

1a. EX/MEM.RegisterRd = ID/EX.RegisterRs

1b. EX/MEM.RegisterRd = ID/EX.RegisterRt

2a. MEM/WB.RegisterRd = ID/EX.RegisterRs

2b. MEM / WB.RegisterRd = ID/EX.RegisterRt

The first hazard will occur in register $2 between the result of sub $2, $1, $3 and the first read operand of and $12, $2, $5. This hazard can be detected when the add instruction is in the EX stage and the prior instruction is in the MEM stage, so this belongs to hazard 1a:

EX/MEM.RegisterRd = ID/EX.RegisterRs = $2.

The remaining hazards are as follows:

- The sub-or is a type 2b hazard: MEM/WB.RegisterRd = ID/EX.RegisterRt = $2.
- The two dependences on sub-add are not hazards because the register file supplies the proper data during the ID stage of add.
- There is no data hazard between sub and sw because sw reads $2 the clock cycle after sub writes $2.

Let's now write both the conditions for detecting hazards and the control signals to resolve them:

Data Hazards and Stalls

One case where forwarding cannot work is when an instruction tries to read a register following a load instruction that writes the same register. Figure 3.57 illustrates the problem. The data is still being read from memory in clock cycle 4 while the ALU is performing the operation for the following instruction. Something must stall the pipeline for the combination of load followed by an instruction that reads its result.

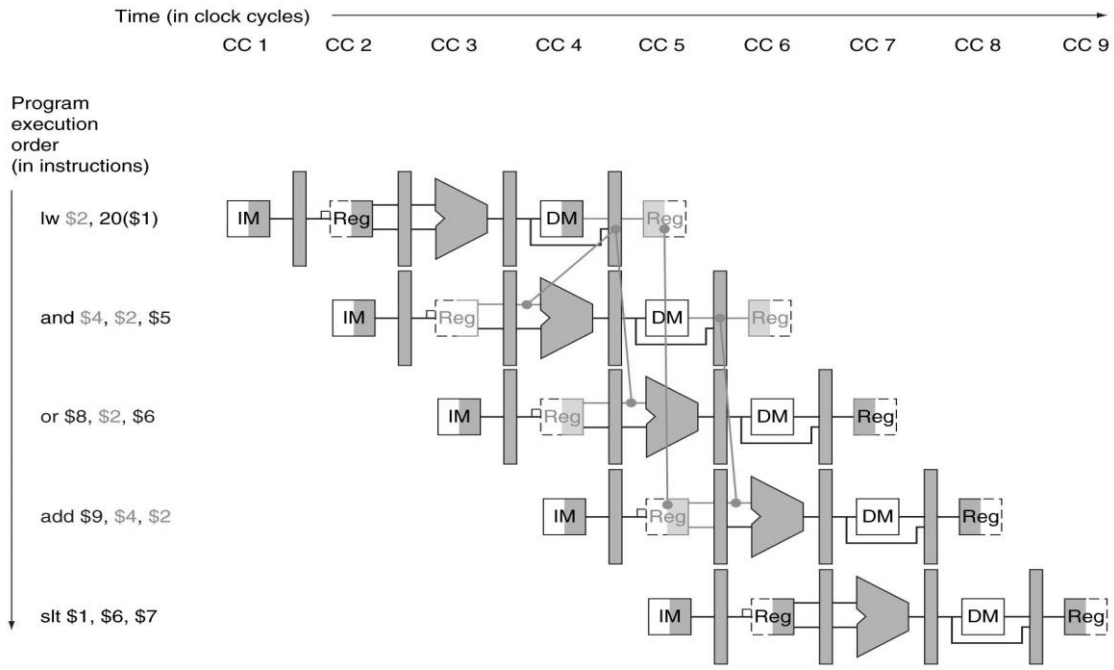


Figure 3.57 A pipelined sequence of instructions

Hence, in addition to a forwarding unit, need a hazard detection unit. It operates during the ID stage so that it can insert the stall between the load and its use. Checking for load instructions, the control for the hazard detection unit is this single condition:

ID/EX.MemRead and

((ID/EX.RegisterRt = IF/ID.RegisterRs) or (ID/EX.RegisterRt = IF/ID.RegisterRt))

If detected, stall and insert bubble

If the instruction in the ID stage is stalled, then the instruction in the IF stage must also be stalled; otherwise, lose the fetched instruction. Preventing these two instructions from making progress is accomplished simply by preventing the PC register and the IF/ID pipeline register from changing. Provided these registers are preserved, the instruction in the IF stage will continue to read using the same PC, and the registers in the ID stage will continue to be read using the same instruction fields in the

IF/ID pipeline register.

The back half of the pipeline starting with the EX stage must be doing something; what it is doing is executing instructions that have no effect: nop's. By deasserting all nine control signals (setting them to 0) in the EX, MEM, and WB stages will create a "do nothing" or nop instruction. By identifying the hazard in the ID stage, we can insert a bubble into the pipeline by changing the EX, MEM, and WB control fields of the ID/EX pipeline register to 0. These control values are forwarded at each clock cycle with the proper effect: no registers or memories are written if the control values are all 0.

Figure 3.58 shows what really happens in the hardware: the pipeline execution slot associated with the AND instruction is turned into a nop and all instructions beginning with the AND instructions are delayed one clock cycle. Like an air bubble in a water pipe, a stall bubble delays everything behind it and proceeds down the instruction pipe one stage each cycle until it exits at the end.

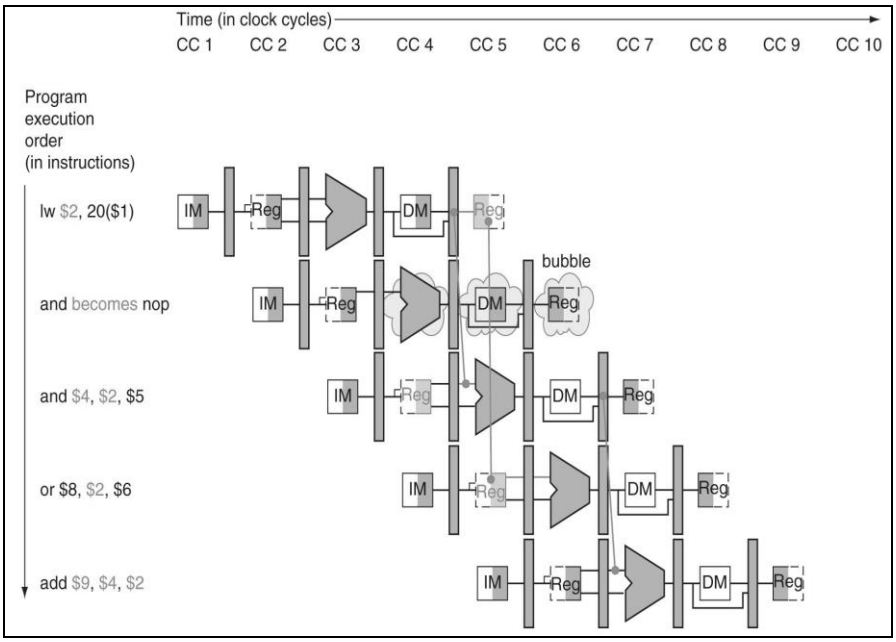


Figure 3.58 The way stalls are really inserted into the pipeline

Figure 3.59 highlights the pipeline connections for both the hazard detection unit and the forwarding unit. As before, the forwarding unit controls the ALU multiplexors to replace the value from a general-purpose register with the value from the proper pipeline register. The hazard detection unit controls the writing of the PC and IF/ID registers plus the multiplexor that chooses between the real control values and all 0's. The hazard detection unit stalls and deasserts the control fields if the load-use hazard test above is true.

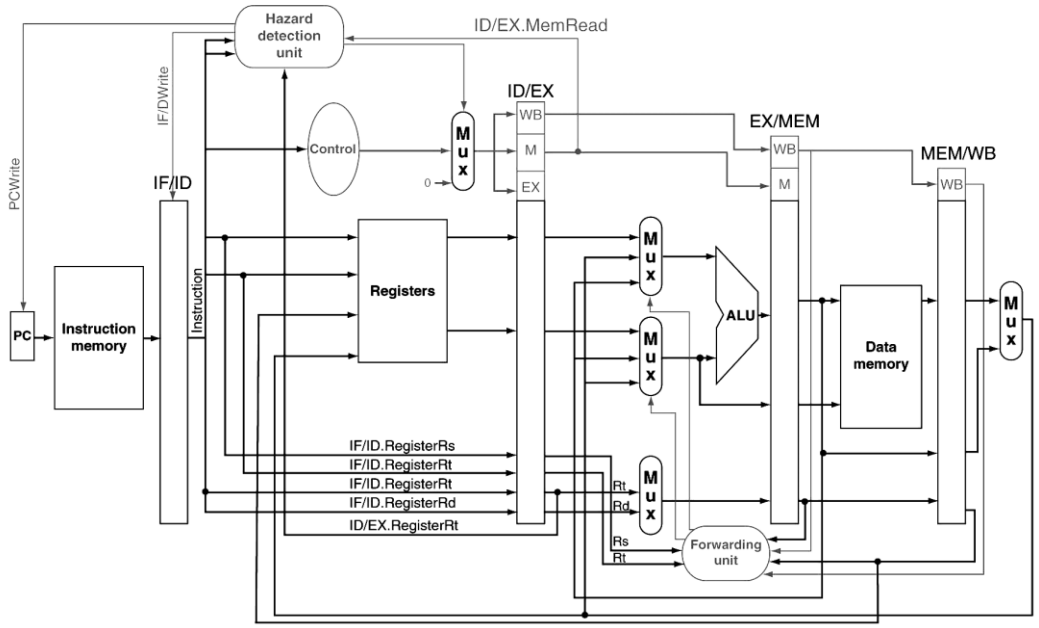


Figure 3.59 Pipelined control overview, showing the two multiplexors for forwarding, the hazard detection unit, and the forwarding unit

Control Hazards

Control hazards occur when execute branch instructions in a pipeline process. It arises from the need to make a decision based on the results of one instruction while others are executing.

Figure 3.60 shows a sequence of instructions and indicates when the branch would occur in this pipeline. An instruction must be fetched at every clock cycle to sustain the pipeline, yet in our design the decision about whether to branch doesn't occur until the MEM pipeline stage. This delay in determining the proper instruction to fetch is called a control hazard or branch hazard or instruction hazard.

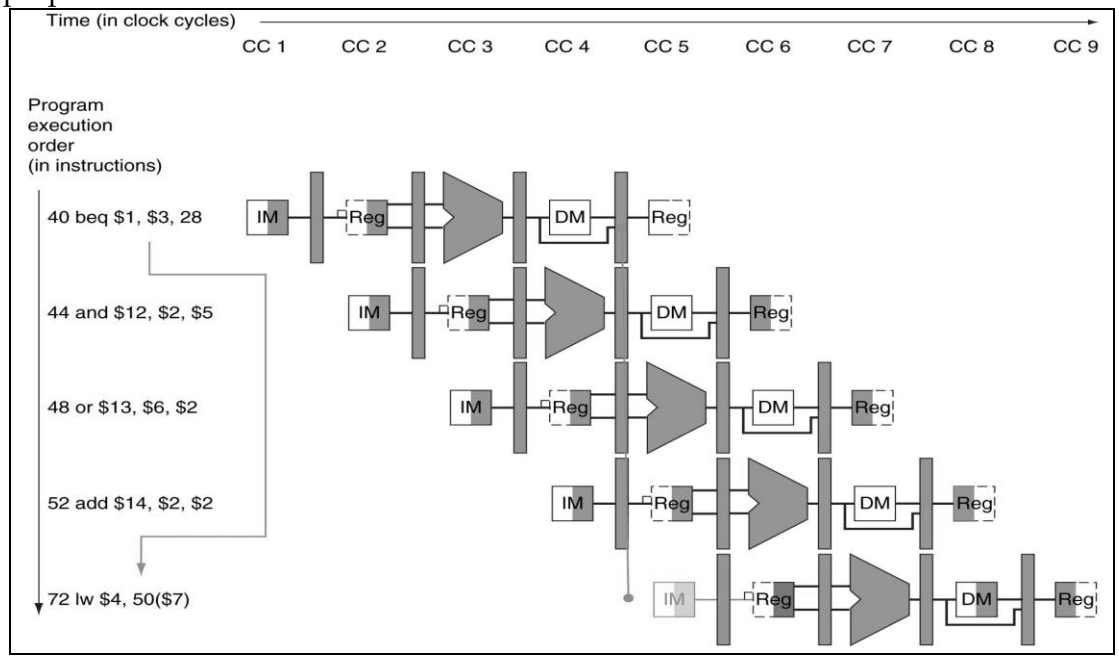


Figure 3.60 The impact of the pipeline on the branch instruction

Control hazards are relatively simple to understand, they occur less frequently than data hazards, and there is nothing as effective against control hazards as forwarding is against data hazards. We look at two schemes for resolving control hazards and one optimization to improve these schemes.

Branch Not Taken:

Stalling until the branch is complete is too slow. A common improvement over branch stalling is to assume that the branch will not be taken and thus continue execution down the sequential instruction stream. If the branch is taken, the instructions that are being fetched and decoded must be discarded. Execution continues at the branch target. If branches are untaken half the time, and if it costs little to discard the instructions, this optimization halves the cost of control hazards.

To discard instructions, merely change the original control values to 0s, similar to the one used to stall for a load-use data hazard. The difference is that change the three instructions in the IF,ID, and EX stages when the branch reaches the MEM stage. Discarding instructions, means it must be able to flush instructions in the IF, ID, and EX stages of the pipeline.

Reducing the Delay of Branches:

One way to improve branch performance is to reduce the cost of the taken branch. So far, we have assumed the next PC for a branch is selected in the MEM stage, but if we move the branch execution earlier in the pipeline, then fewer instructions need be flushed. The MIPS architecture was designed to support fast single-cycle branches that could be pipelined with a small branch penalty.

Moving the branch decision up requires two actions to occur earlier:

- ✓ Computing the Branch Target Address and
- ✓ Evaluating the Branch Decision.

Computing the Branch Target Address:

The easy part of this change is to move up the branch address calculation. We already have the PC value and the immediate field in the IF/ID pipeline register, so we just move the branch adder from the EX stage to the ID stage; of course, the branch target address calculation will be performed for all instructions, but only used when needed.

Evaluating the Branch Decision:

The harder part is the branch decision itself. For branch equal, we would compare the two registers read during the ID stage to see if they are equal. Equality can be tested by first exclusive ORing their respective bits and then ORing all the results. Moving the branch test to the ID stage implies

additional forwarding and hazard detection hardware, since a branch dependent on a result still in the pipeline must still work properly with this optimization.

For example, to implement branch on equal, we will need to forward results to the equality test logic that operates during ID. There are two complicating factors:

- ✓ During ID, we must decode the instruction, decide whether a bypass to the equality unit is needed, and complete the equality comparison so that if the instruction is a branch, we can set the PC to the branch target address. Forwarding for the operands of branches was formerly handled by the ALU forwarding logic, but the introduction of the equality test unit in ID will require new forwarding logic.
- ✓ Because the values in a branch comparison are needed during ID but may be produced later in time, it is possible that a data hazard can occur and a stall will be needed.

Despite these difficulties, moving the branch execution to the ID stage is an improvement, because it reduces the penalty of a branch to only one instruction if the branch is taken, namely, the one currently being fetched.

To flush the instructions in the IF stage, add a control line, called IF.Flush, that zeros the instruction field of the IF/ID pipeline register. Clearing the register transforms the fetched instruction into a nop, an instruction that has no action and changes no state.

Branch Prediction:

Branch prediction is a technique for predicting branch penalty associated with conditional branches is to attempt to predict whether or not a particular branch will be taken or not. There are two forms of branch prediction techniques. They are

- ✓ Static Branch Prediction
- ✓ Dynamic Branch Prediction

Static Branch Prediction

The simplest form of branch prediction is to assume that the branch will not take place and to continue to fetch instructions in sequential address order, until the branch condition is evaluated, instruction execution along the predicted path must be done on a speculative basis.

For the simple five-stage pipeline, possibly coupled compiler-based prediction is probably adequate. With deeper pipelines, the branch penalty increases when measured in clock cycles. Similarly, with multiple issue, the branch penalty increases in terms of instructions lost. This combination means that in an aggressive pipeline, a simple static prediction scheme will probably waste too much performance.

Dynamic Branch Prediction

With more hardware it is possible to try to predict branch behavior during program execution. One approach is to look up the address of the instruction to see if a branch was taken the last time this instruction was executed, and if so, to begin fetching new instructions from the same place as the last time. This technique is called dynamic branch prediction.

One-bit Prediction Scheme:

One implementation of that approach is a branch prediction buffer or branch history table. A branch prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction. The memory contains a bit that says whether the branch was recently taken or not.

This is the simplest form of buffer, in fact, if the prediction is the right one – it may have been put there by another branch that has the same low-order address bits. However, this doesn't affect correctness.

This simple 1-bit prediction scheme has a performance shortcoming: even if a branch is almost always taken, we can predict incorrectly twice, rather than once, when it is not taken.

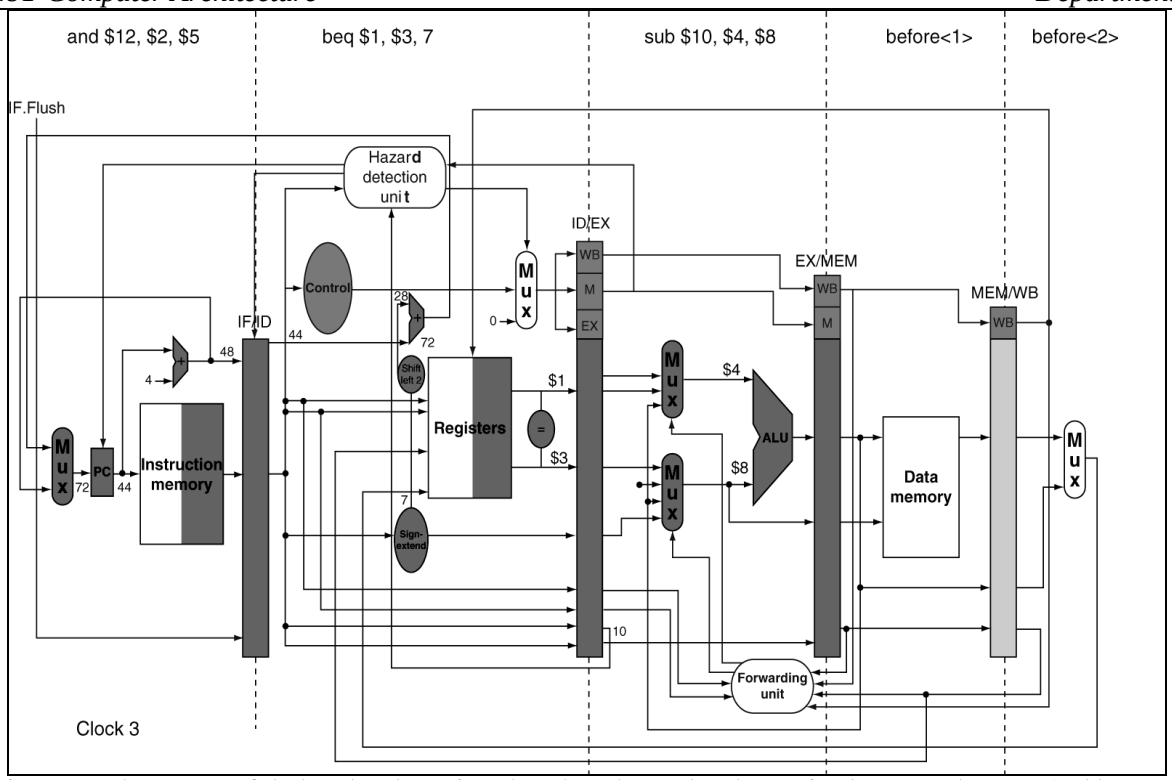


Figure 3.61 The ID stage of clock cycle 3 determines that a branch must be taken, so it selects 72 as the next PC address and zeros the instruction fetched for the next clock cycle

Two-bit Prediction Scheme:

The accuracy of the predictor would match the taken branch frequency for these highly regular branches. To remedy this weakness, the 2-bit prediction schemes are often used. In a 2-bit scheme, a prediction must be wrong twice before it is changed.

DATA HAZARD AND ITS HANDING

Data Hazards

When a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available. A data hazard is any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. As a result, some operation has to be delayed and the pipeline stalls. In a computer pipeline, data hazards arise from the dependence of one instruction on an earlier one that is still in the pipeline. For example, suppose an add instruction followed immediately by a subtract instruction that uses the sum ($s0).

add $s0, $t0, $t1;
sub $t2, $s0, $t3;

Without intervention, a data hazard could severely stall the pipeline. The add instruction doesn't write its result until fifth stage, meaning that it has spent three clock cycles in the pipeline.

PIPELINE CONTROL

Add control to the single-cycle datapath, then add control to the pipelined datapath. To provide control in a pipelined datapath, the following task has to be followed:

- The first step is to label the control lines on the existing datapath.
- Borrow/ Use the control lines from the simple datapath.
- Use the same ALU control logic, branch logic, destination-register-number multiplexor, and control lines.

Figure 3.49 shows the pipelined datapath of a MIPS processor with the control signal identified. In the case of a single-cycle implementation, assume that the PC is written on each clock cycle, so there is no separate write signal for the PC. By the same argument, there are no separate write signals for the pipeline registers (IF/ID, ID/EX. EX/MEM, and MEM/WB), since the pipeline registers are also written during each clock cycle.

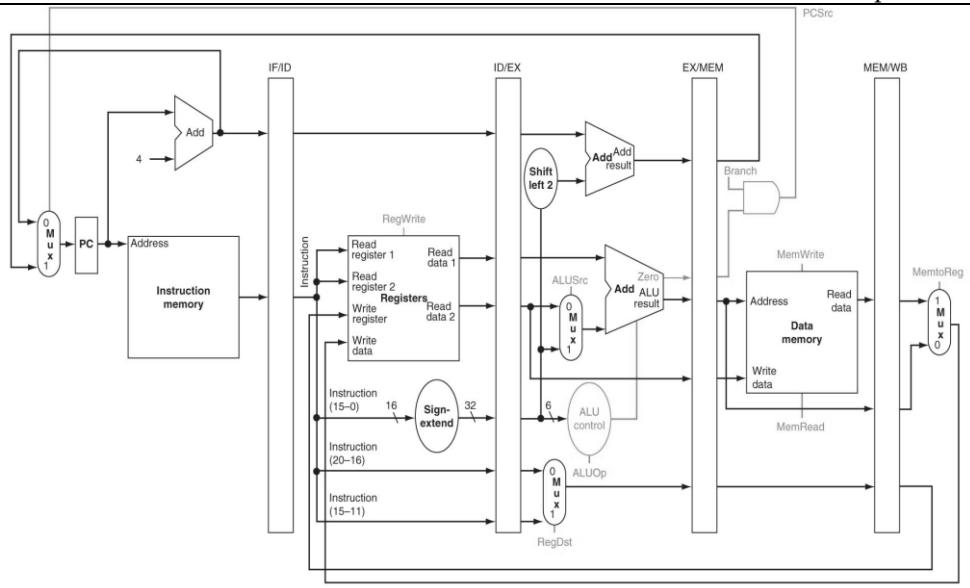


Figure 3.49 The pipelined data path with the control signals identified

To specify control for the pipeline, need only to set the control values during each pipeline stage. Because each control line is associated with a component active in only a single pipeline stage, divide the control lines into five groups according to the pipeline stage.

1. Instruction Fetch: The control signals to read instruction memory and to write the PC are always asserted, so there is nothing special to control in this pipeline stage.
2. Instruction Decode / Register File Read: As in the previous stage, the same thing happens at every clock cycle, so there are no optional control lines to set.
3. Execution / Address Calculation: The signals to be set are RegDst, ALUop, and ALUSrc. These signals select the result register, the ALU operation, and either read data 2 or a sign-extended immediate for the ALU.
4. Memory Access: The control lines set in this stage are Branch, MemRead, and MemWrite. These signals are set by the branch equal, load, and store instructions, respectively. The PCSrc selects the next sequential address unless control asserts Branch and the ALU result was 0.
5. Write Back: The two control lines are MemtoReg, which decides between sending the ALU result or the memory value to the register file, and RegWrite, which writes the chosen value.

Since the control lines start with the EX stage, create the control information during instruction decode. Figure 3.50 shows that these control signals are then used in the appropriate pipeline stage as the instruction moves down the pipeline, just as the destination register number for loads moves down the pipeline.

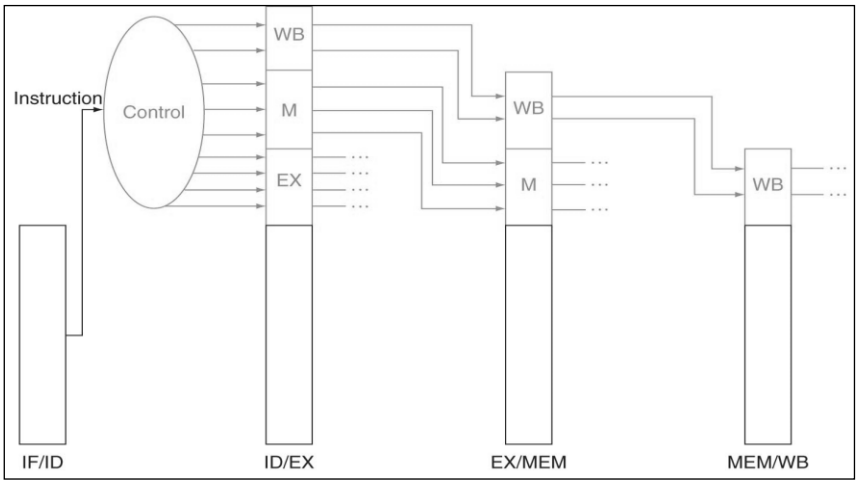


Figure 3.51 shows the full datapath with the extended pipeline registers and with the control lines connected to the proper stage

Implementing control means setting the nine control lines to these values in each stage for each instruction. The simplest way to do this is to extend the pipeline registers to include control information.

CONTROL HAZARDS

Control hazards occur when execute branch instructions in a pipeline process. It arises from the need to make a decision based on the results of one instruction while others are executing. Figure 3.60 shows a sequence of instructions and indicates when the branch would occur in this pipeline. An instruction must be fetched at every clock cycle to sustain the pipeline, yet in our design the decision about whether to branch doesn't occur until the MEM pipeline stage. This delay in determining the proper instruction to fetch is called a control hazard or branch hazard or instruction hazard.

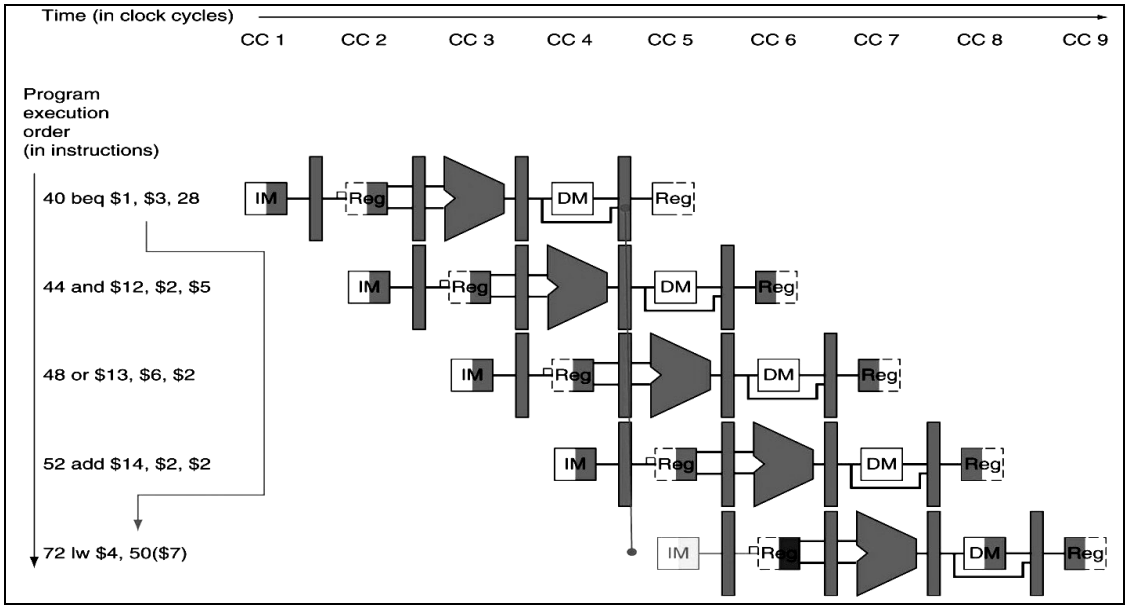


Figure 3.60 The impact of the pipeline on the branch instruction

Control hazards are relatively simple to understand, they occur less frequently than data hazards, and there is nothing as effective against control hazards as forwarding is against data hazards. Two schemes for resolving control hazards and one optimization to improve these schemes.

Branch Not Taken:

Stalling until the branch is complete is too slow. A common improvement over branch stalling is to assume that the branch will not be taken and thus continue execution down the sequential instruction stream. If the branch is taken, the instructions that are being fetched and decoded must be discarded. Execution continues at the branch target. If branches are untaken half the time, and if it costs little to discard the instructions, this optimization halves the cost of control hazards.

To discard instructions, merely change the original control values to 0s, similar to the one did to stall for a load-use data hazard. The difference is that must also change the three instructions in the IF,ID, and EX stages when the branch reaches the MEM stage. Discarding instructions, means must be able to flush instructions in the IF, ID, and EX stages of the pipeline.

Reducing the Delay of Branches:

One way to improve branch performance is to reduce the cost of the taken branch. So far, assumed the next PC for a branch is selected in the MEM stage, but if move the branch execution earlier in the pipeline, then fewer instructions need be flushed. The MIPS architecture was designed to support fast single-cycle branches that could be pipelined with a small branch penalty.

Moving the branch decision up requires two actions to occur earlier:

- ✓ Computing the Branch Target Address and
- ✓ Evaluating the Branch Decision.

Computing the Branch Target Address:

The easy part of this change is to move up the branch address calculation. It is already have the PC value and the immediate field in the IF/ID pipeline register, so just move the branch adder from the EX stage to the ID stage; of course, the branch target address calculation will be performed for all instructions, but only used when needed.

Evaluating the Branch Decision:

The harder part is the branch decision itself. For branch equal, we would compare the two registers read during the ID stage to see if they are equal. Equality can be tested by first exclusive ORing their respective bits and then ORing all the results. Moving the branch test to the ID stage implies

additional forwarding and hazard detection hardware, since a branch dependent on a result still in the pipeline must still work properly with this optimization.

For example, to implement branch on equal, need to forward results to the equality test logic that operates during ID. There are two complicating factors:

1. During ID, decode the instruction, decide whether a bypass to the equality unit is needed, and complete the equality comparison so that if the instruction is a branch, then set the PC to the branch target address. Forwarding for the operands of branches was formerly handled by the ALU forwarding logic, but the introduction of the equality test unit in ID will require new forwarding logic.
2. Because the values in a branch comparison are needed during ID but may be produced later in time, it is possible that a data hazard can occur and a stall will be needed.

Despite these difficulties, moving the branch execution to the ID stage is an improvement, because it reduces the penalty of a branch to only one instruction if the branch is taken, namely, the one currently being fetched.

To flush the instructions in the IF stage, add a control line, called IF.Flush, that zeros the instruction field of the IF/ID pipeline register. Clearing the register transforms the fetched instruction into a nop, an instruction that has no action and changes no state.

Branch Prediction:

Branch prediction is a technique for predicting branch penalty associated with conditional branches is to attempt to predict whether or not a particular branch will be taken or not. There are two forms of branch prediction techniques. They are

1. Static Branch Prediction
2. Dynamic Branch Prediction

Static Branch Prediction :

The simplest form of branch prediction is to assume that the branch will not take place and to continue to fetch instructions in sequential address order, until the branch condition is evaluated, instruction execution along the predicted path must be done on a speculative basis.

Assuming a branch is not taken is one simple form of branch prediction. In that case, predict that branches are untaken, flushing the pipeline when go wrong. For the simple five-stage pipeline, possibly coupled compiler-based prediction is probably adequate. With deeper pipelines, the branch penalty increases when measured in clock cycles. Similarly, with multiple issues, the branch penalty increases in terms of instructions lost. This combination means that in an aggressive pipeline, a simple static prediction scheme will probably waste too much performance.

Dynamic Branch Prediction :

With more hardware it is possible to try to predict branch behavior during program execution. One approach is to look up the address of the instruction to see if a branch was taken the last time this instruction was executed and if so, to begin fetching new instructions from the same place as the last time. This technique is called dynamic branch prediction.

One-bit Prediction Scheme:

One implementation of that approach is a branch prediction buffer or branch history table. A branch prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction. The memory contains a bit that says whether the branch was recently taken or not.

This is the simplest form of buffer, in fact, if the prediction is the right one – it may have been put there by another branch that has the same low-order address bits. However, this doesn't affect correctness.

Prediction is just a hint that we hope is correct, so fetching begins in the predicted direction. If the hint turns out to be wrong, the incorrectly predicted instructions are deleted, the prediction bit is inverted and stored back, and the proper sequence is fetched and executed.

This simple 1-bit prediction scheme has a performance shortcoming: even if a branch is almost always taken, can predict incorrectly twice, rather than once, when it is not taken.

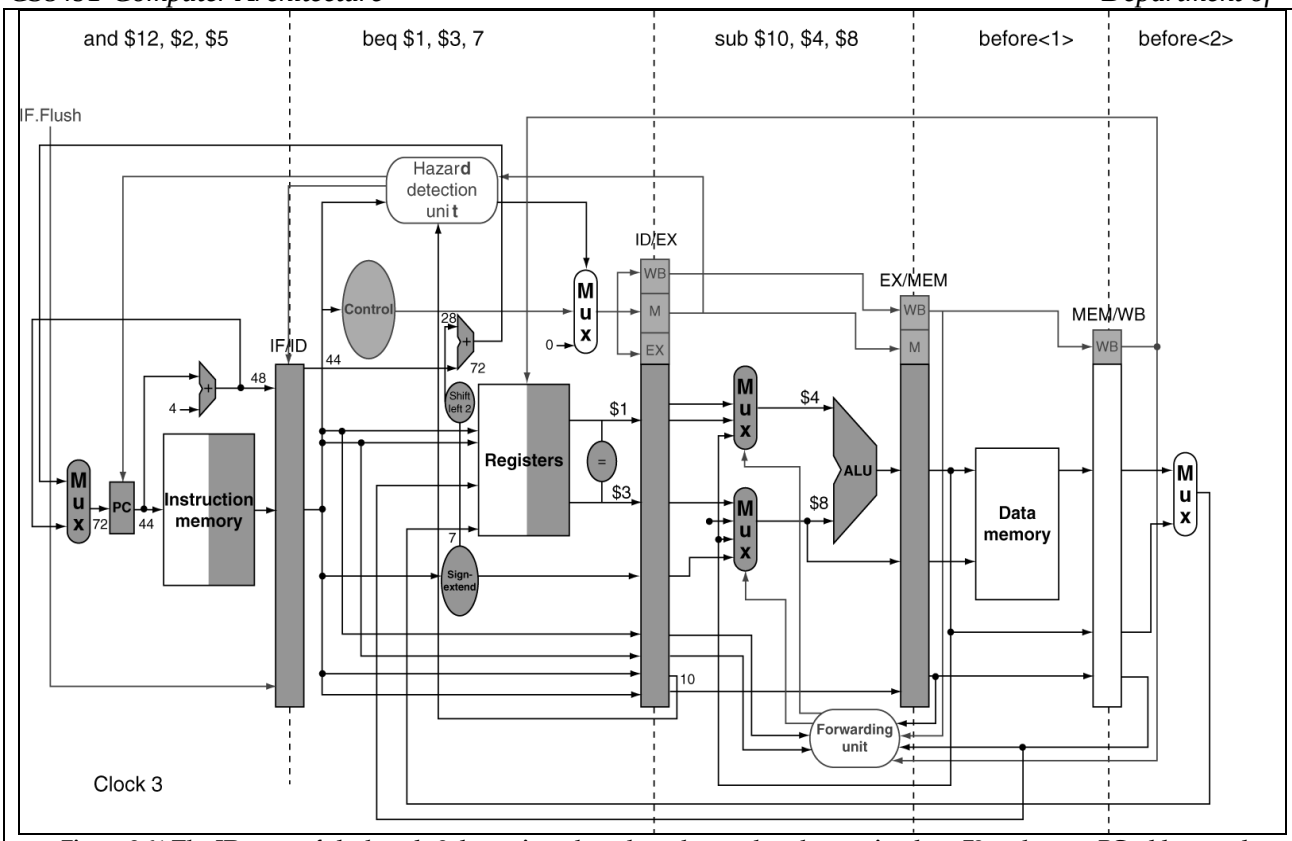


Figure 3.61 The ID stage of clock cycle 3 determines that a branch must be taken, so it selects 72 as the next PC address and zeros the instruction fetched for the next clock cycle

Two-bit Prediction Scheme:

The accuracy of the predictor would match the taken branch frequency for these highly regular branches. To remedy this weakness, the 2-bit prediction schemes are often used. In a 2-bit scheme, a prediction must be wrong twice before it is changed. Figure 3.62 shows the finite-state machine for a 2-bit prediction scheme.

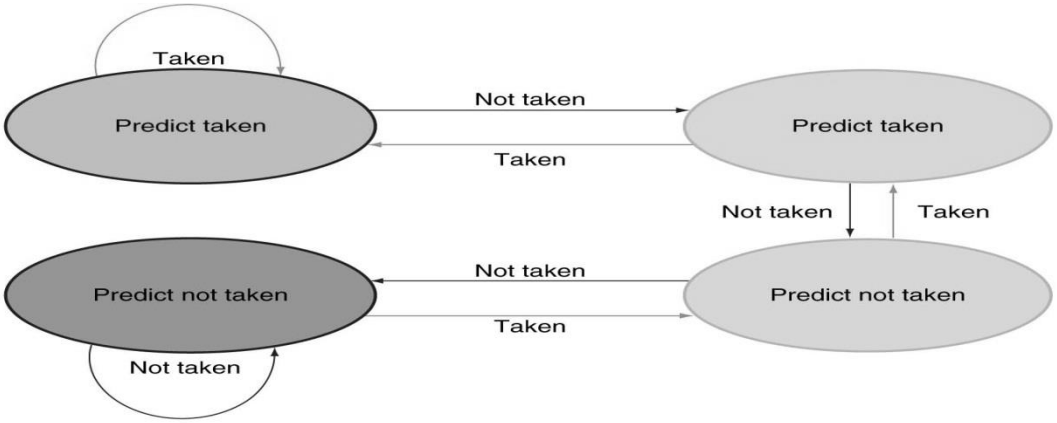


Figure 3.62 The states in a 2-bit prediction scheme

A branch prediction buffer can be implemented as a small, special buffer accessed with the instruction address during the IF pipe stage. If the instruction is predicted as taken, fetching begins from the target as soon as the PC is known; it can be as early as the ID stage. Otherwise, sequential fetching and executing continue. If the prediction turns out to be wrong, the prediction bits are changed.

Delayed Branching Technique

A delayed branch always executes the following instruction, but the second instruction following the branch will be affected by the branch.

The location following a branch instruction is called a branch delay slot. Compilers and assemblers try to place an instruction that always executes after the branch in the branch delay slot. The job of software is to make the successor instructions valid and useful. Figure 3.63 shows the three ways in which the branch delay slots can be scheduled.

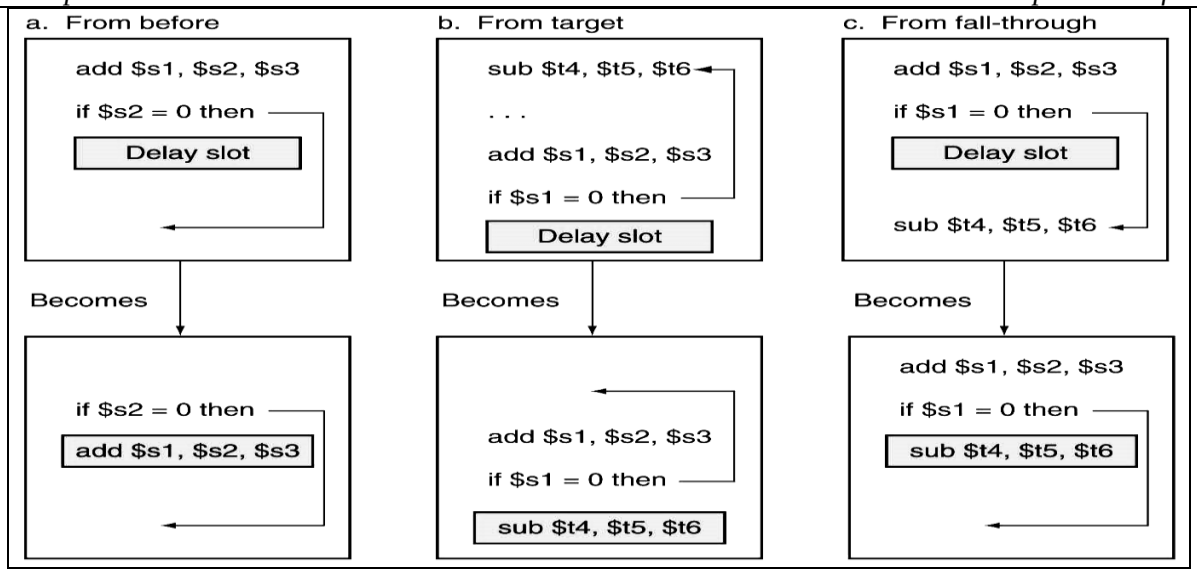


Figure 3.63 Scheduling the branch delay slot

The limitations on delayed branch scheduling arise from the restrictions on the instructions that are scheduled into the delay slots and our ability to predict at compile time whether a branch is likely to be taken or not.

Delayed branching was a simple and effective solution for a five-stage pipeline issuing one instruction each clock cycle. As processors go to both longer pipelines and issuing multiple instructions per clock cycle, the branch delay becomes longer, and a single delay slot is insufficient. Hence, delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches. Simultaneously, the growth in available transistors per chip has made dynamic prediction relatively cheaper.

EXCEPTION HANDLING

Types of Exceptions:

The two types of exceptions that our current implementation can generate are :

a. Execution of an undefined instruction and
b. Arithmetic overflow.

Undefined Instruction Exception

The basic action that the processor must perform when an exception occurs is to save the address of the offending instruction in the exception program counter (EPC) and then transfer control to the operating system at some specified address.

The OS can then take the appropriate action, which may involve providing some service to the user program, taking some predefined action in response to an overflow, or stopping the execution of the program and reporting an error. After performing whatever action is required because of the exception, the operating system can terminate the program or may continue its execution, using the EPC to determine where to restart the execution of the program.

For the OS to handle the exception, it must know the reason for the exception, in addition to the instruction that caused it. There are two main methods used to communicate the reason for an exception.

1. The method used in the MIPS architecture is to include a status register (called the cause register), which holds a field that indicates the reason for the exception.
2. A second method is to use vectored interrupts. In a vectored interrupt, the address to which control is transferred is determined by the cause of the exception.

For example, to accommodate the two exception types listed above, define the following two exception vector addresses:

Exception Type	Exception Vector Address (in hex)
Undefined Instruction	8000 0000
Arithmetic Overflow	8000 0180

The OS knows the reason for the exception by the address at which it is initiated. The addresses are separated by 32 bytes or eight instructions, and the OS must record the reason for the exception and may

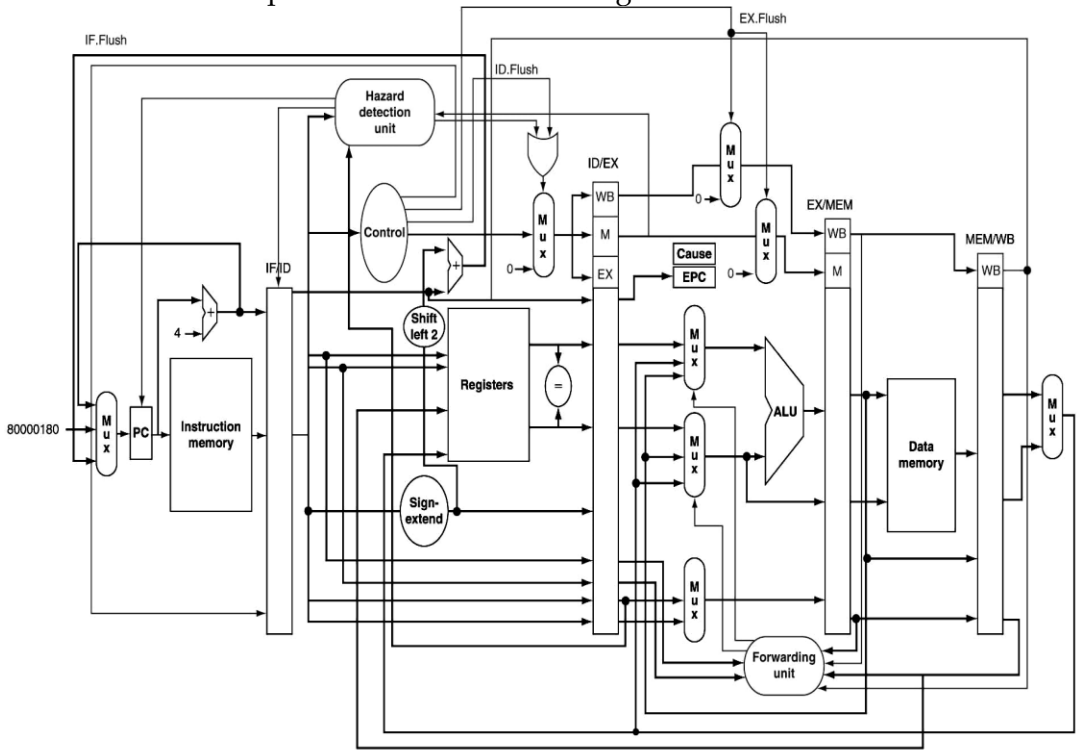
perform some limited processing in this sequence. When the exception is not vectored, a single entry point for all exceptions can be used, and the OS system decodes the status register to find the cause.

To perform the processing required for exceptions a few extra registers and control signals are added to the basic implementation and by slightly extending the control. To implement the exception used in the MIPS architecture, single entry point address 8000 0180 is assumed and two additional registers are added to the MIPS implementation:

1. **EPC:** A 32-bit register used to hold the address of the affected instruction. Such a register is needed even when exceptions are vectored.
2. **Cause Register:** A register used to record the cause of the exception. In the MIPS architecture, this register is 32-bits, although some bits are currently unused. Assume there is a five-bit field that encodes the two possible exception sources mentioned above, with 10 representing an unused instruction and 12 representing arithmetic overflow.

a. Arithmetic Overflow

A pipelined implementation treats exceptions as another form of control hazard. Suppose there is an arithmetic overflow in an add instruction, then flush the instructions that follow the add instruction from the pipeline and begin fetching instructions from the new address. Same mechanism is used for taken branches, but this time the exception causes the deasserting of control lines.



The datapath with controls to handle exceptions

When dealt with branch misprediction, it is shown how to flush the instruction in the IF stage by turning it into a nop. To flush instructions in the ID stage, use the multiplexor already in the ID stage that zeros control signals for stalls. A new control signal, called ID.Flush is ORed with the stall signal from the hazard detection unit to flush during ID. To flush the instruction in the EX phase, use a new signal called EX.Flush to cause new multiplexors to zero the control lines. To start fetching instructions from location 8000 0180, which is the MIPS exception address, simply add an additional input to the PC multiplexor that sends 8000 0180 to the PC.

This example points out a problem with exceptions: if the execution is not stopped in the middle of the instruction, the programmer will not be able to see the original value of register that helped cause the overflow because it will be clobbered as the destination register of the add instruction. Because of careful planning, the overflow exception is detected during the EX stage; hence, it can use the EX.Flush signal to prevent the instruction in the EX stage from writing its result in the WB stage. Many exceptions require that eventually complete the instruction that caused the exception as if it executed normally. The easiest way to do this is to flush the instruction and restart it from the beginning after the exception is handled.

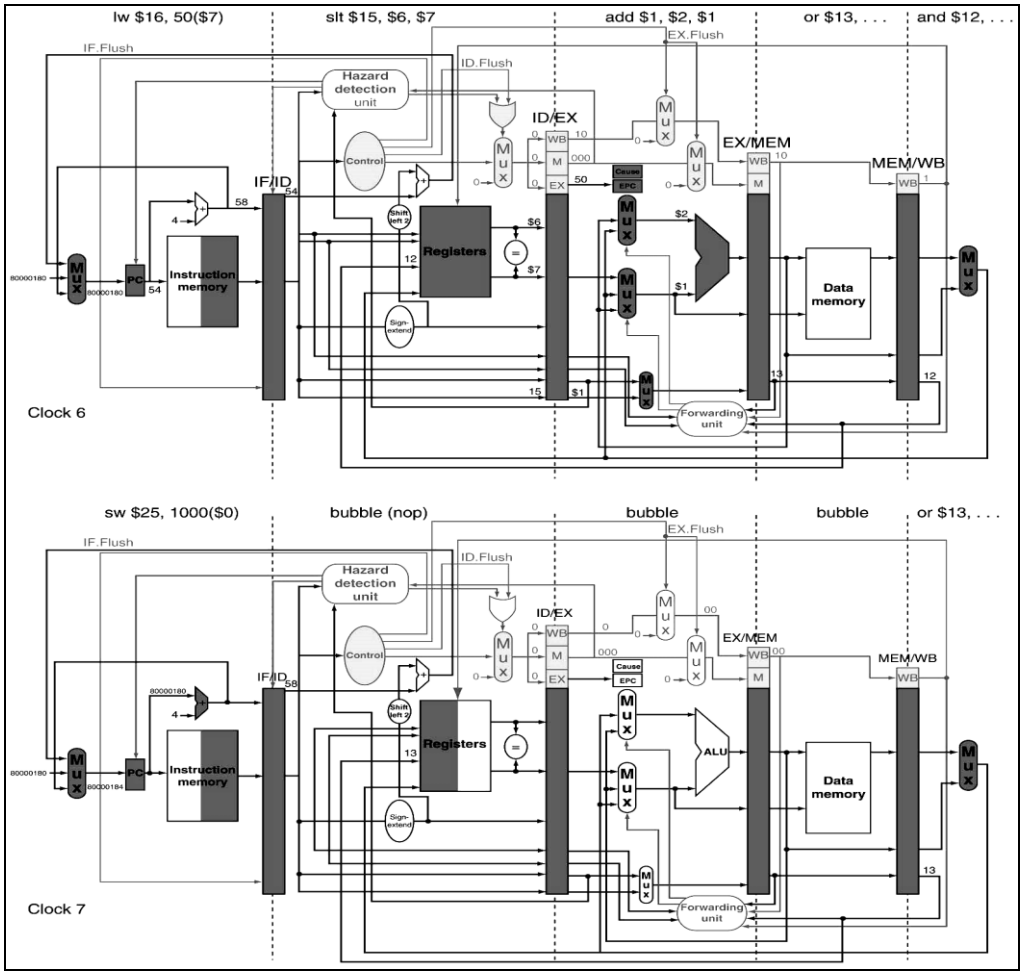
The final step is to save the address of the offending instruction in the exception program counter

(EPC). In reality, save the address+4, so the exception handling routine must first subtract 4 from the saved value.

Figure (a) shows the events, starting with the add instruction in the EX stage. The overflow is detected during that phase, and 8000 0180 is forced into the PC. Clock cycle 7 shows that the add and following instructions are flushed, and the first instruction of the exception code is fetched. With five instructions active in any clock cycle, the challenge is to associate an exception with the appropriate instruction. Moreover, multiple exceptions can occur simultaneously in a single clock cycle. The solution is to prioritize the exceptions so that it is easy to determine which is serviced first. In most MIPS implementations, the hardware sorts exceptions so that the earliest instruction is interrupted.

I/O device requests and hardware malfunctions are not associated with a specific instruction, so the implementation has some flexibility as to when to interrupt the pipeline. Hence, the mechanism used for other exceptions works just fine.

The EPC captures the address of the interrupted instructions, and the MIPS cause register records all possible exceptions in a clock cycle, so the exception software must match the exception to the instruction. An important clue is known in which pipeline stage a type of exception can occur. For example, an undefined instruction is discovered in the ID stage, and invoking the operating system occurs in the EX stage. Exceptions are collected in the Cause register in a pending exception field so that the hardware can interrupt based on later exceptions, once the earliest one has been serviced.



Figure(a): The result of an exception due to arithmetic overflow in the add instruction

Imprecise Exceptions

It is also called as imprecise interrupt. Interrupts or exceptions in pipelined computer that are not associated with the exact instruction that was the cause of the interrupt or exception.

Precise Exceptions:

It is also called as precise interrupt. An interrupt or exception that is always associated with the correct instruction in pipelined computers.

UNIT IV PARALLELISM

Parallel processing challenges – Flynn's classification – SISD, MIMD, SIMD, SPMD, and Vector Architectures - Hardware multithreading – Multi-core processors and other Shared Memory Multiprocessors - Introduction to Graphics Processing Units, Clusters, Warehouse Scale Computers and other Message-Passing Multiprocessors.

CHALLENGES IN PARALLEL PROCESSING

Instruction Level Parallel Processing:

- ✓ Architectural technique that allows the overlap of individual machine operations (add, mul, load, store …)
- ✓ Multiple operations will execute in parallel (simultaneously)

Goal: Speed Up the execution

Example:

load R1 ← R2	add R3 ← R3, "1"
add R3 ← R3, "1"	add R4 ← R3, R2
add R4 ← R4, R2	store [R4] ← R0

Sequential execution (Without ILP)

Add r1, r2 → r8 4 cycles
Add r3, r4 → r7 4 cycles Total of 8 cycles

ILP execution (overlap execution)

Add r1, r2 → r8
Add r3, r4 → r7 Total of 5 cycles

ILP Architectures

Sequential Architectures: the program is not expected to convey any explicit information regarding parallelism. (Superscalar processors)

Dependence Architectures: the program explicitly indicates the dependences that exist between operations (Dataflow processors)

Independence Architectures: the program provides information as to which operations are independent of one another. (VLIW processors)

ILP and parallel processing

- ✓ Overlap individual machine operations (add, mul, load…) so that they execute in parallel
- ✓ Transparent to the user
- ✓ Goal: speed up execution

Parallel Processing

- ✓ Having separate processors getting separate chunks of the program (processors programmed to do so)
- ✓ Nontransparent to the user
- ✓ Goal: speed up and quality up

Challenges in ILP

- ✓ In order to achieve parallelism we should not have dependences among instructions which are executing in parallel:
- ✓ H/W terminology Data Hazards (RAW, WAR, WAW)

S/W terminology Data Dependencies

DEPENDENCY AND THE VARIOUS TYPES OF DEPENDENCIES

- Dependences are a property of programs.
- If two instructions are data dependent they cannot execute simultaneously.
- A dependence results in a hazard and the hazard causes a stall.
- Data dependences may occur through registers or memory.

Types of dependencies

- ✓ Name dependencies
  - o Output dependence

- Anti-dependence
- ✓ Data True dependence
- ✓ Control Dependence
- ✓ Resource Dependence

Name dependency

1. **Output dependence**

When instruction I and J write the same register or memory location. The ordering must be preserved to leave the correct value in the register.

```
add r7,r4,r3
div r7,r2,r8
```

2. **Anti-dependence**

When instruction j writes a register or memory location that instruction I reads.

```
i: add r6,r5,r4
j: sub r5,r8,r11
```

Data Dependence

An instruction *j* is data dependent on instruction *i* if either of the following hold:

- ✓ instruction *i produces a result that may be used by* instruction *j* , or
- ✓ instruction *j is data dependent on instruction k, and instruction k is data dependent on instruction i*

```
LOOP  LD    F0, 0(R1)
ADD   F4, F0, F2
SD    F4, 0(R1)
SUB   R1, R1, -8
BNE   R1, R2, LOOP
```

Control Dependence

A control dependence determines the ordering of an instruction i, with respect to a branch instruction so that the instruction i is executed in correct program order.

Example:

```
If  p1 {  S1; };
If  p2 {  S2; };
```

Two constraints imposed by control dependences:

1. An instruction that is control dependent on a branch cannot be moved *before* the branch
2. An instruction that is not control dependent on a branch cannot be moved *after* the branch

Resource Dependence

An instruction is resource-dependent on a previously issued instruction if it requires a hardware resource which is still being used by a previously issued instruction.

Example:

```
div r1, r2, r3
div r4, r2, r5
```

DYNAMIC & STATIC MULTIPLE ISSUE PROCESSOR AND THEIR SCHEDULING

Goal: Sustain a CPI of less than 1 by issuing and processing multiple instructions per cycle

- SuperScalar – Issue varying number of instructions per clock
- Statically Scheduled
- Dynamically Scheduled
  - VLIW (EPIC) – Issue a fixed number of instructions formatted as one large instruction or instruction "packet" – Similar to static-scheduled superscalar

To give a flavor of static multiple issue, we consider a simple two-issue MIPS processor, where one of the instructions can be an integer A L U operation or branch and the other can be a load or store.

Such a design is like that used in some embedded MIPS processors. Issuing two instructions per cycle will require fetching and decoding 64 bits of instructions.

In many static multiple-issue processors, and essentially all VLIW processors, the layout of simultaneously issuing instructions is restricted to simplify the decoding and instruction issue.

Hence, we will require that the instructions be paired and aligned on a 64-bit boundary, with the A L U or branch portion appearing first. Furthermore, if one instruction of the pair cannot be used, we require that it be replaced with a nop. Thus, the instructions always issue in pairs, possibly with a nop in

one slot.

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

The ALU and data transfer instructions are issued at the same time.

Here we have assumed the same five-stage structure as used for the single-issue pipeline. Although this is not strictly necessary, it does have some advantages.

In particular, keeping the register writes at the end of the pipeline simplifies the handling of exceptions and the maintenance of a precise exception model, which become more difficult in multiple-issue processors.

Multiple Issue Code Scheduling

Scheduling the following loop on a static two-issue pipeline for MIPS.

```
Loop: lw    $t0, 0($s1)    # $t0=array element
      addu  $t0,$t0,$s2# add scalar in $s2
      sw    $t0, 0($s1)# store result
      addi  $s1,$s1,-4# decrement pointer
      bne   $s1,$zero,Loop# branch $s1!=0
```

Reorder the instructions to avoid as many pipeline stalls as possible. Assume branches are predicted, so that control hazards are handled by the hardware.

The first three instructions have data dependences, and so do the last two. The following table shows the best schedule for these instructions. Notice that just one pair of instructions has both issue slots used. I t takes four clocks per loop iteration; at four clocks to execute five instructions, we get the disappointing CPI of 0.8 versus the best case of 0.5., or an I P C of 1.25 versus 2.0. Notice that in computing CPI or IPC, we do not count any nops executed as useful instructions. Doing so would improve CPI, but not performance!

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:		lw $t0, 0($s1)	1
	addi $s1,$s1,-4		2
	addu $t0,$t0,$s2		3
	bne $s1,$zero,Loop	sw $t0, 4($s1)	4

The scheduled code as it would look on a two-issue MIPS pipeline. The empty slots are nops.

Dynamic Multiple-Issue Processors

Dynamic multiple-issue processors are also known as superscalar processors, or simply superscalars. In the simplest superscalar processors, instructions issue in order, and the processor decides whether zero, one, or more instructions can issue in a given clock cycle. Obviously, achieving good performance on such a processor still requires the compiler to try to schedule instructions to move dependences apart and thereby improve the instruction issue rate. Even with such compiler scheduling, there is an important difference between this simple superscalar and a VLIW processor: the code, whether scheduled or not, is guaranteed by the hardware to execute correctly. Furthermore, compiled code will always run correctly independent of the issue rate or pipeline structure of the processor. In some VLIW designs, this has not been the case, and recompilation was required when moving across different processor models; in other static issue processors, code would run correctly across different implementations, but often so poorly as to make compilation effectively required.

Dynamic Pipeline Scheduling

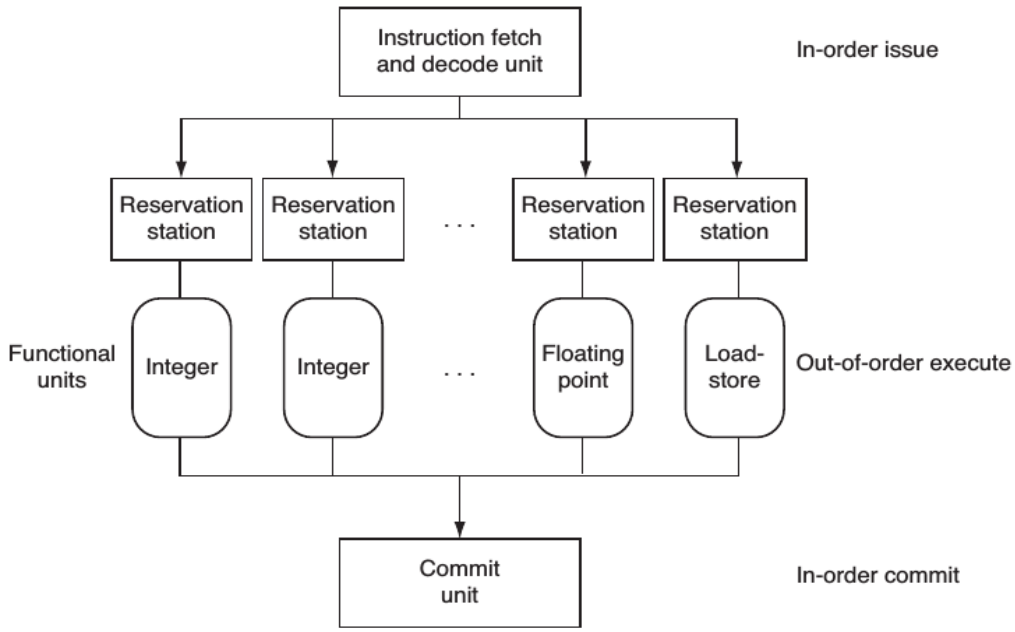


FIGURE 4.72 The three primary units of a dynamically scheduled pipeline. The final step of updating the state is also called retirement or graduation.

Dynamic pipeline scheduling chooses which instructions to execute next, possibly reordering them to avoid stalls. In such processors, the pipeline is divided into three major units: an instruction fetch and issue unit, multiple functional units (a dozen or more in high-end designs in 2008), and a commit unit.

- The first unit fetches instructions, decodes them, and sends each instruction to a corresponding functional unit for execution. Each functional unit has buffers, called reservation stations, which hold the operands and the operation.
- As soon as the buffer contains all its operands and the functional unit is ready to execute, the result is calculated.
- When the result is completed, it is sent to any reservation stations waiting for this particular result as well as to the commit unit, which buffers the result until it is safe to put the result into the register file or, for a store, into memory.
- The buffer in the commit unit, often called the reorder buffer, is also used to supply operands, in much the same way as forwarding logic does in a statically scheduled pipeline.
- Once a result is committed to the register file, it can be fetched directly from there, just as in a normal pipeline.

FLYNN'S CLASSIFICATION

Flynn's Taxonomy In 1966, Michael Flynn proposed a classification for computer architectures based on the number of instruction steams and data streams.

Flynn's Classification

		Data Streams	
		Single	**Multiple**
Instruction Streams	Single	SISD: Intel Pentium 4	SIMD: SSE instructions of x86
	Multiple	MISD: No examples today	MIMD: Intel Xeon e5345 (Clovertown)

FIGURE 7.6 Hardware categorization and examples based on number of instruction streams and data streams: SISD, SIMD, MISD, and MIMD.

Flynn's Classification Of Computer Architectures

SISD or Single Instruction stream, **Single Data stream**. A uniprocessor.

SIMD or Single Instruction stream, Multiple Data streams. A multiprocessor. The same instruction is applied to many data streams, as in a vector processor or array processor.

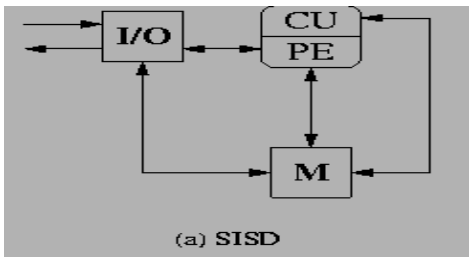
MIMD or Multiple Instruction streams, Multiple Data streams. A multiprocessor.

SPMD Single Program, Multiple Data streams. The conventional MIMD programming model, where a single program runs across all processors.

VECTOR

SISD

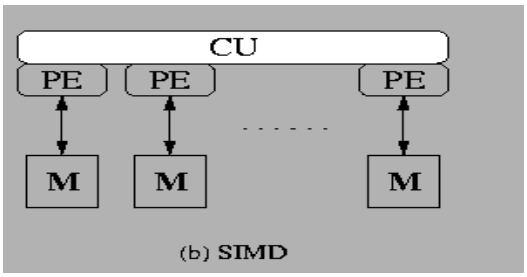
- ✓ **SISD** (**S**inge-**I**nstruction stream, **S**inge-**D**ata stream)
- ✓ SISD corresponds to the traditional mono-processor (von Neumann computer). A single data stream is being processed by one instruction stream OR
- ✓ A single-processor computer (uni-processor) in which a single stream of instructions is generated from the program.



where CU= Control Unit, PE= Processing Element, M= Memory

SIMD

- ✓ **SIMD** (**S**ingle-**I**nstruction stream, **M**ultiple-**D**ata streams)
- ✓ Each instruction is executed on a different set of data by different processors i.e multiple processing units of the same type process on multiple-data streams.
- ✓ This group is dedicated to array processing machines.
- ✓ Sometimes, vector processors can also be seen as a part of this group.

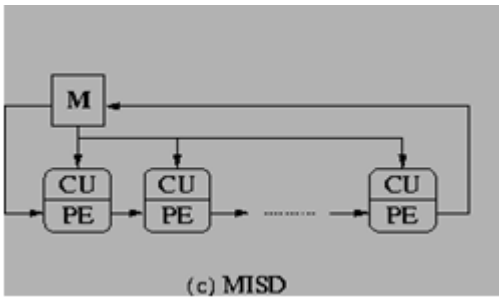


SIMD in x86: Multimedia Extensions

The most widely used variation of SIMD is found in almost every microprocessor today, and is the basis of the hundreds of MMX and SSE instructions of the x86 microprocessor. They were added to improve performance of multimedia programs. These instructions allow the hardware to have many ALUs operate simultaneously or, equivalently, to partition a single, wide ALU into many parallel smaller ALUs that operate simultaneously. For example, you could consider a single hardware component to be one 64-bit ALU or two 32-bit ALUs or four 16-bit ALUs or eight 8-bit ALUs. Loads and stores are simply as wide as the widest ALU, so the programmer can think of the same data transfer instruction as transferring either a single 64-bit data element or two 32-bit data elements or four 16-bit data elements or eight 8-bit data elements.

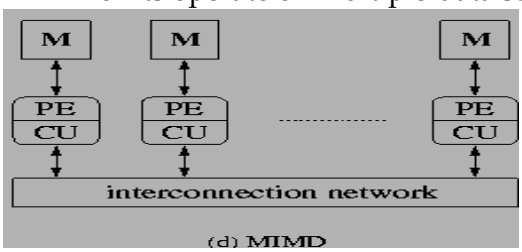
MISD

- ✓ **MISD** (**M**ultiple-**I**nstruction streams, **S**inge-**D**ata stream)
- ✓ Each processor executes a different sequence of instructions.
- ✓ In case of MISD computers, multiple processing units operate on one single-data stream
- ✓ In practice, this kind of organization has never been used.



MIMD

- ✓ **MIMD** (**M**ultiple-**I**nstruction streams, **M**ultiple-**D**ata streams)
- ✓ Each processor has a separate program.
- ✓ An instruction stream is generated from each program.
- ✓ Each instruction operates on different data.
- ✓ This last machine type builds the group for the traditional multi-processors. Several processing units operate on multiple-data streams.



VECTOR

An older and more elegant interpretation of SIMD is called a vector architecture, which has been closely identified with Cray Computers. It is again a great match to problems with lots of data-level parallelism. Rather than having 64 ALUs perform 64 additions simultaneously, like the old array processors, the vector architectures pipelined the ALU to get good performance at lower cost. The basic philosophy of vector architecture is to collect data elements from memory, put them in order into a large set of registers, operate on them sequentially in registers, and then write the results back to memory. A key feature of vector architectures is a set of vector registers. Thus, a vector architecture might have 32 vector registers, each with 64 64-bit elements.

Vector versus Scalar

Vector instructions have several important properties compared to conventional instruction set architectures, which are called scalar architectures in this context:

1. A single vector instruction specifies a great deal of work—it is equivalent to executing an entire loop. The instruction fetch and decode bandwidth needed is dramatically reduced.
2. By using a vector instruction, the compiler or programmer indicates that the computation of each result in the vector is independent of the computation of other results in the same vector, so hardware does not have to check for data hazards within a vector instruction.
3. Vector architectures and compilers have a reputation of making it much easier than MIMD multiprocessors to write efficient applications when they contain data-level parallelism.
4. Hardware need only check for data hazards between two vector instructions once per vector operand, not once for every element within the vectors. Reduced checking can save power as well.
5. Vector instructions that access memory have a known access pattern. If the vector's elements are all adjacent, then fetching the vector from a set of heavily interleaved memory banks works very

well. Thus, the cost of the latency to

6. main memory is seen only once for the entire vector, rather than once for each word of the vector.
7. Because an entire loop is replaced by a vector instruction whose behavior is predetermined, control hazards that would normally arise from the loop branch are nonexistent.
8. The savings in instruction bandwidth and hazard checking plus the efficient use of memory bandwidth give vector architectures advantages in power and energy versus scalar architectures.
9. For these reasons, vector operations can be made faster than a sequence of scalar operations on the same number of data items, and designers are motivated to include vector units if the application domain can use them frequently.

Vector versus Multimedia Extensions

Like multimedia extensions found in the x86 SSE instructions, a vector instruction specifies multiple operations. However, multimedia extensions typically specify a few operations while vector specifies dozens of operations. Unlike multimedia extensions, the number of elements in a vector operation is not in the opcode but in a separate register. This means different versions of the vector architecture can be implemented with a different number of elements just by changing the contents of that register and hence retain binary compatibility. In contrast, a new large set of opcodes is added each time the "vector" length changes in the multimedia extension architecture of the x86.

Also unlike multimedia extensions, the data transfers need not be contiguous. Vectors support both strided accesses, where the hardware loads every nth data element in memory, and indexed accesses, where hardware finds the addresses of the items to be loaded in a vector register.

Like multimedia extensions, vector easily captures the flexibility in data widths, so it is easy to make an operation work on 32 64-bit data elements or 64 32-bit data elements or 128 16-bit data elements or 256 8-bit data elements. Generally, vector architectures are a very efficient way to execute data parallel processing programs; they are better matches to compiler technology than multimedia extensions; and they are easier to evolve over time than the multimedia extensions to the x86 architecture.

HARDWARE MULTITHREADING

Hardware multithreading allows multiple threads to share the functional units of a single processor in an overlapping fashion. To permit this sharing, the processor must duplicate the independent state of each thread.

For example,

1. Each thread would have a separate copy of the register file and the PC.
2. The memory itself can be shared through the virtual memory mechanisms, which already support multiprogramming.

There are two main approaches to hardware multithreading.

Fine-grained Multithreading:

It switches between threads on each instruction, resulting in interleaved execution of multiple threads. This interleaving is often done in a roundrobin fashion, skipping any threads that are stalled at that time. To make fine-grained multithreading practical, the processor must be able to switch threads on every clock cycle.

Advantage

- It can hide the throughput losses that arise from both short and long stalls, since instructions from other threads can be executed when one thread stalls.

Disadvantage

- It slows down the execution of the individual threads, since a thread that is ready to execute without stalls will be delayed by instructions from other threads.

Coarse-grained multithreading :

It switches threads only on costly stalls, such as second-level cache misses. This change relieves the need to have thread switching be essentially free and is much less likely to slow down the execution of an individual thread, since instructions from other threads will only be issued when a thread encounters a costly stall. Coarse-grained multithreading suffers, however, from a major drawback: it is limited in its ability to overcome throughput losses, especially from shorter stalls.

Limitation:

- The pipeline start-up costs of coarse-grained multithreading.
- Because a processor with coarse-grained multithreading issues instructions from a single thread, when a stall occurs, the pipeline must be emptied or frozen.

- The new thread that begins executing after the stall must fill the pipeline before instructions will be able to complete.
- Due to this start-up overhead, coarse-grained multithreading is much more useful for reducing the penalty of high-cost stalls, where pipeline refill is negligible compared to the stall time.

Simultaneous multithreading (SMT):

It is a variation on hardware multithreading that uses the resources of a multiple-issue, dynamically scheduled processor to exploit thread-level parallelism at the same time it exploits instruction-level parallelism.

Motivation of SMT:

SMT is that multiple-issue processors often have more functional unit parallelism available than a single thread can effectively use. Furthermore, with register renaming and dynamic scheduling, multiple instructions from independent threads can be issued without regard to the dependences among them. The resolution of the dependences can be handled by the dynamic scheduling capability. SMT is always executing instructions from multiple threads, leaving it up to the hardware to associate instruction slots and renamed registers with their proper threads.

For example,

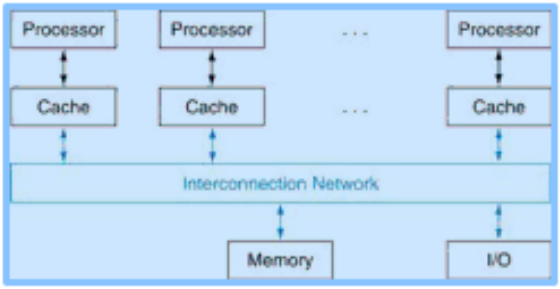
1. The Sun UltraSPARC T2 (Niagara 2) microprocessor is an example of a return to simpler microarchitectures and hence the use of fine-grained multithreading.
2. Second, a key performance challenge is tolerating latency due to cache misses. Fine-grained computers like the UltraSPARC T2 switch to another thread on a miss, which is probably more effective in hiding memory latency than trying to fill unused issue slots as in SMT.
3. The goal of hardware multithreading is to use hardware more efficiently by sharing components between different tasks. Multicore designs share resources as well. For example, two processors might share a floating- point unit or an L3 cache. Such sharing reduces some of the benefits of multithreading compared with providing more non-multithreaded cores.

MULTI-CORE PROCESSORS AND OTHER SHARED MEMORY MULTIPROCESSORS

A shared memory multiprocessor (SMP) is one that offers the programmer a single physical address space across all processors—which is nearly always the case for multicore chips—although a more accurate term would have been shared-address multiprocessor. Processors communicate through shared variables in memory, with all processors capable of accessing any memory location via loads and stores. Figure 6.7 shows the classic organization of an SMP. Note that such systems can still run independent jobs in their own virtual address spaces, even if they all share a physical address space.

Centralized Shared Memory Architecture

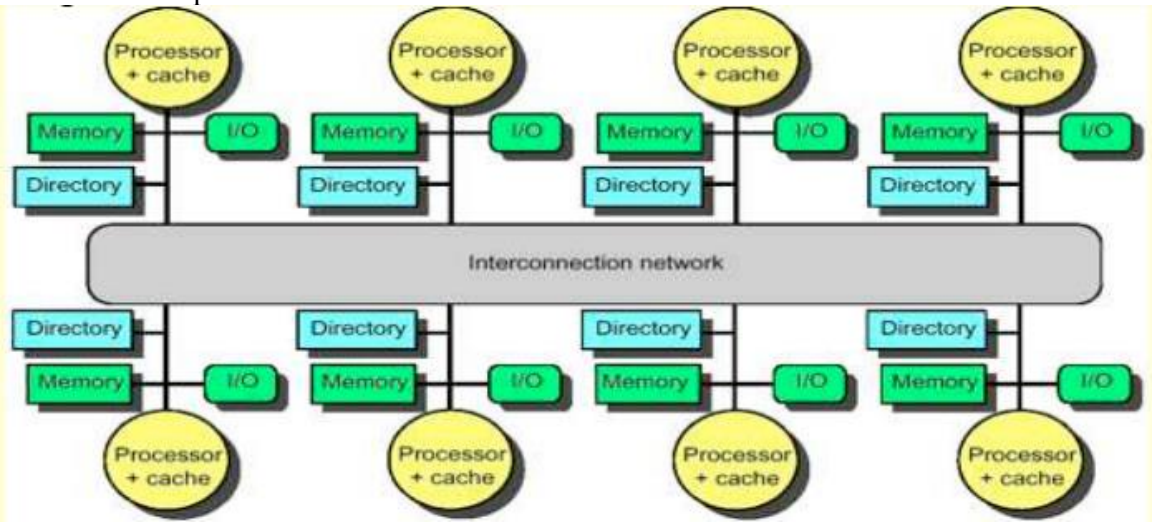
- Processors share a single centralized memory through a bus interconnect
- Feasible for small processor count to limit memory contention
- Caches serve to:
  - ✓ Increase bandwidth versus bus/memory
  - ✓ Reduce latency of access
  - ✓ Valuable for both private data and shared data
  - ✓ Access to shared data is optimized by replication
- Decreases latency
- Increases memory bandwidth
- Reduces contention
- Replication introduces the problem of cache coherence



Shared Memory Architecture

- Each processor has its own memory system, which it can access directly
- To obtain data that is stored in some other processor's memory, a processor must communicate

with that to request the data



Advantages

- Each processor has its own local memory system
- More total bandwidth in the memory system than in a centralized memory system
- The latency to complete a memory request is lower— each processor's memory is located physically close to it

Disadvantages

- Only some of the data in the memory is directly accessible by each processor, since a processor can only read and write its local memory system
- Requires communication through the network and leads to the coherence problem— major source of complexity in shared-memory systems
- Possible that could exist in different processors' memories • Leads to different processors having different values for the same variable

MULTI-CORE COMPUTING

All computers are now parallel computers. Multi-core processors represent an important new trend in computer architecture. Decreased power consumption and heat generation. Minimized wire lengths and interconnect latencies. They enable true thread-level parallelism with great energy efficiency and scalability. To utilize their full potential, applications will need to move from a single to a multi-threaded model. Parallel programming techniques likely to gain importance. The difficult problem is not building multi-core hardware, but programming it in a way that lets mainstream applications benefit from the continued exponential growth in CPU performance. The software industry needs to get back into the state where existing applications run faster on new hardware.

Challenges resulting from Multicore:

- Relies on effective exploitation of multiple-thread parallelism
  - Need for parallel computing model and parallel programming model
- Aggravates memory wall
- Memory bandwidth
  - Way to get data out of memory banks
  - Way to get data into multi-core processor array
- Memory latency
- Fragments L3 cache
- Pins become strangle point
  - Rate of pin growth projected to slow and flatten
  - Rate of bandwidth per pin (pair) projected to grow slowly
- Requires mechanisms for efficient inter-processor coordination
  - Synchronization
  - Mutual exclusion

- Context switching

Advantages:

1. Cache coherency circuitry can operate at a much higher clock rate than is possible if the signals have to travel off-chip.
2. Signals between different CPUs travel shorter distances, those signals degrade less.
3. These higher quality signals allow more data to be sent in a given time period since individual signals can be shorter and do not need to be repeated as often.
4. A dual-core processor uses slightly less power than two coupled single-core processors.

Disadvantages

1. Ability of multi-core processors to increase application performance depends on the use of multiple threads within applications.
2. Most Current video games will run faster on a 3 GHz single-core processor than on a 2GHz dual-core processor (of the same core architecture.
3. Two processing cores sharing the same system bus and memory bandwidth limits the real-world performance advantage.
4. If a single core is close to being memory bandwidth limited, going to dual-core might only give 30% to 70% improvement.
5. If memory bandwidth is not a problem, a 90% improvement can be expected.

INTRODUCTION TO GRAPHICS PROCESSING UNITS

A major justification for adding SIMD instructions to existing architectures was that many microprocessors were connected to graphics displays in PCs and workstations, so an increasing fraction of processing time was used for graphics. Hence, as Moore's law increased the number of transistors available to microprocessors, it made sense to improve graphics processing.

Moreover, at the very high end were expensive graphics cards typically from Silicon Graphics that could be added to workstations, to enable the creation of photographic quality images. These high-end graphics cards were popular for creating computer-generated images that later found their way into television advertisements and then into movies. Thus, video graphics controllers had a target to shoot for as processing resources increased, much as supercomputers provided a rich resource of ideas for microprocessors to borrow in the quest for greater performance.

A major driving force for improving graphics processing was the computer game industry, both on PCs and in dedicated game consoles such as the Sony PlayStation. The rapidly growing game market encouraged many companies to make increasing investments in developing faster graphics hardware, and this positive feedback led graphics processing to improve at a faster rate than general-purpose processing in mainstream microprocessors.

Given that the graphics and game community had different goals than the microprocessor development community, it evolved its own style of processing and terminology. As the graphics processors increased in power, they earned the name Graphics Processing Units or GPUs to distinguish themselves from CPUs. Here are some of the key characteristics as to how GPUs vary from CPUs:

■ GPUs are accelerators that supplement a CPU, so they do not need be able to perform all the tasks of a CPU. This role allows them to dedicate all their resources to graphics. It's fine for GPUs to perform some tasks poorly or not at all, given that in a system with both a CPU and a GPU, the CPU can do them if needed. Thus, the CPU-GPU combination is one example of heterogeneous multiprocessing, where not all the processors are identical. (Another example is the IBM Cell architecture which was also designed to accelerate 2D and 3D graphics.)

■ The programming interfaces to GPUs are high-level application programming interfaces (APIs), such as OpenGL and Microsoft's DirectX, coupled with high-level graphics shading languages, such as NVIDIA's C for Graphics (Cg) and Microsoft's High Level Shader Language (HLSL). The language compilers target industry-standard intermediate languages instead of machine instructions. GPU driver software generates optimized GPU-specific machine instructions. While these APIs and languages evolve rapidly to embrace new GPU resources enabled by Moore's law, the freedom from backward binary instruction compatibility enables GPU designers to explore new architectures without the fear that they will be saddled with implementing failed experiments forever. This environment leads to more rapid

innovation in GPUs than in CPUs.

■ Graphics processing involves drawing vertices of 3D geometry primitives such as lines and triangles and shading or rendering pixel fragments of geometric primitives. Video games, for example, draw 20 to 30 times as many pixels as vertices.

■ Each vertex can be drawn independently, and each pixel fragment can be rendered independently. To render millions of pixels per frame rapidly, the GPU evolved to execute many threads from vertex and pixel shader programs in parallel.

■ The graphics data types are vertices, consisting of (x, y, z, w) coordinates, and pixels, consisting of (red, green, blue, alpha) color components. GPUs represent each vertex component as a 32-bit floating-point number. Each of the four pixel components was originally an 8-bit unsigned integer, but recent GPUs now represent each component as single-precision floating-point number between 0.0 and 1.0.

■ The working set can be hundreds of megabytes, and it does not show the same temporal locality as data does in mainstream applications. Moreover, there is a great deal of data-level parallelism in these tasks. These differences led to different styles of architecture

■ Perhaps the biggest difference is that GPUs do not rely on multilevel caches to overcome the long latency to memory, as do CPUs. Instead, GPUs rely on having enough threads to hide the latency to memory. That is, between the time of a memory request and the time that data arrives, the GPU executes hundreds or thousands of threads that are independent of that request.

■ GPUs rely on extensive parallelism to obtain high performance, implementing many parallel processors and many concurrent threads.

■ The GPU main memory is thus oriented toward bandwidth rather than latency. There are even separate DRAM chips for GPUs that are wider and have higher bandwidth than DRAM chips for CPUs. In addition, GPU memories have traditionally had smaller main memories than conventional microprocessors. In 2008, GPUs typically have 1 GB or less, while CPUs have 2 to 32 GB. Finally, keep in mind that for general-purpose computation, you must include the time to transfer the data between CPU memory and GPU memory, since the GPU is a coprocessor.

■ Given the reliance on many threads to deliver good memory bandwidth, GPUs can accommodate many parallel processors as well as many threads. Hence, each GPU processor is highly multithreaded.

■ In the past, GPUs relied on heterogeneous special purpose processors to deliver the performance needed for graphics applications. Recent GPUs are heading toward identical general-purpose processors to give more flexibility in programming, making them more like the multicore designs found in mainstream computing.

■ Given the four-element nature of the graphics data types, GPUs historically have SIMD instructions, like CPUs. However, recent GPUs are focusing more on scalar instructions to improve programmability and efficiency.

■ Unlike CPUs, there has been no support for double precision floating-point arithmetic, since there has been no need for it in the graphics applications. In 2008, the first GPUs to support double precision in hardware were announced. Nevertheless, single precision operations will still be eight to ten times faster than double precision, even on these new GPUs, while the difference in performance for CPUs is limited to benefits in transferring fewer bytes in the memory system due to using narrow data.

COMPUTER ARCHITECTURE OF WAREHOUSE-SCALE COMPUTERS

Networks are the connective tissue that binds 50,000 servers together. Analogous to the memory hierarchy of Chapter 2, WSCs use a hierarchy of networks. Figure 6.5 shows one example. Ideally, the combined network would provide nearly the performance of a custom high-end switch for 50,000 servers at nearly the cost per port of a commodity switch designed for 50 servers.

The 19-inch (48.26-cm) rack is still the standard framework to hold servers, despite this standard going back to railroad hardware from the 1930s. Servers are measured in the number of rack units (U) that they occupy in a rack. One U is 1.75 inches (4.45 cm) high, and that is the minimum space a server can occupy. A 7-foot (213.36-cm) rack offers 48 U, so it's not a coincidence that the most popular switch for a rack is a 48-port Ethernet switch. This product has become a commodity that costs as little as $30 per port for a 1 Gbit/sec Ethernet link in 2011 Note that the bandwidth within the rack is the same for each server, so it does not matter where the software places the sender and the receiver as long as they are within the same rack. This flexibility is ideal from a software perspective. These switches typically offer two to eight uplinks, which leave the rack to go to the next higher switch in the network hierarchy. Thus, the bandwidth leaving the rack is 6 to 24 times smaller—48/8 to 48/2—than the bandwidth within

the rack. This ratio is called oversubscription. Alas, large oversubscription means programmers must be aware of the performance consequences when placing senders and receivers in different racks. This increased software-scheduling burden is another argument for network switches designed specifically for the datacenter.

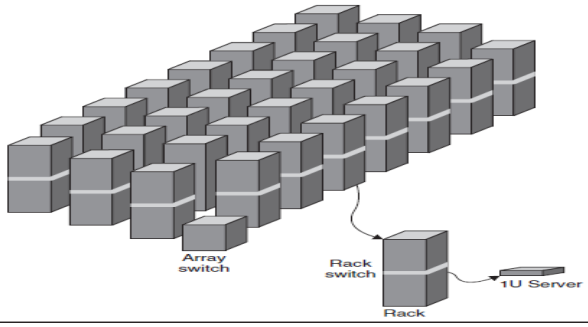


Figure 6.5 Hierarchy of switches in a WSC. (Based on Figure 1.2 of Barroso and Hölzle [2009].)

Storage

A natural design is to fill a rack with servers, minus whatever space you need forthe commodity Ethernet rack switch. This design leaves open the question of where the storage is placed. From a hardware construction perspective, the simplest solution would be to include disks inside the server, and rely on Ethernet connectivity for access to information on the disks of remote servers. The alternative would be to use network attached storage (NAS), perhaps over a storage network like Infiniband. The NAS solution is generally more expensive per terabyte of storage, but it provides many features, including RAID techniques to improve dependability of the storage. As you might expect from the philosophy expressed in the prior section, WSCs generally rely on local disks and provide storage software that handles connectivity and dependability. For example, GFS uses local disks and maintains at least three replicas to overcome dependability problems. This redundancy covers not just local disk failures, but also power failures to racks and to whole clusters. The eventual consistency flexibility of GFS lowers the cost of keeping replicas consistent, which also reduces the network bandwidth requirements of the storage system. Local access patterns also mean high bandwidth to local storage. Beware that there is confusion about the term cluster when talking about the architecture of a WSC. WSC is just an extremely large cluster. Cluster to mean the next-sized grouping of computers, in this case about 30 racks. In this chapter, to avoid confusion we will use the term array to mean a collection of racks, preserving the original meaning of the word cluster to mean anything from a collection of networked computers within a rack to an entire warehouse full of networked computers.

Array Switch

The switch that connects an array of racks is considerably more expensive than the 48-port commodity Ethernet switch. This cost is due in part because of the higher connectivity and in part because the bandwidth through the switch must be much higher to reduce the oversubscription problem. A switch that has 10 times the bisection bandwidth—basically, the worst-case internal bandwidth—of a rack switch costs about 100 times as much.

One reason is that the cost of switch bandwidth for n ports can grow as n2. Another reason for the high costs is that these products offer high profit margins for the companies that produce them. They justify such prices in part by providing features such as packet inspection that are expensive because they must Operate at very high rates. For example, network switches are major users of content-addressable memory chips and of field-programmable gate arrays (FPGAs), which help provide these features, but the chips themselves are expensive. While such features may be valuable for Internet settings, they are generally unused inside the datacenter.

WSC Memory Hierarchy

Figure 6.6 shows the latency, bandwidth, and capacity of memory hierarchy inside a WSC, and Figure 6.7 shows the same data visually. These figures are based on the following assumptions.

	Local	Rack	Array
DRAM latency (microseconds)	0.1	100	300
Disk latency (microseconds)	10,000	11,000	12,000
DRAM bandwidth (MB/sec)	20,000	100	10
Disk bandwidth (MB/sec)	200	100	10
DRAM capacity (GB)	16	1040	31,200
Disk capacity (GB)	2000	160,000	4,800,000

Figure 6.6 Latency, bandwidth, and capacity of the memory hierarchy of a WSC

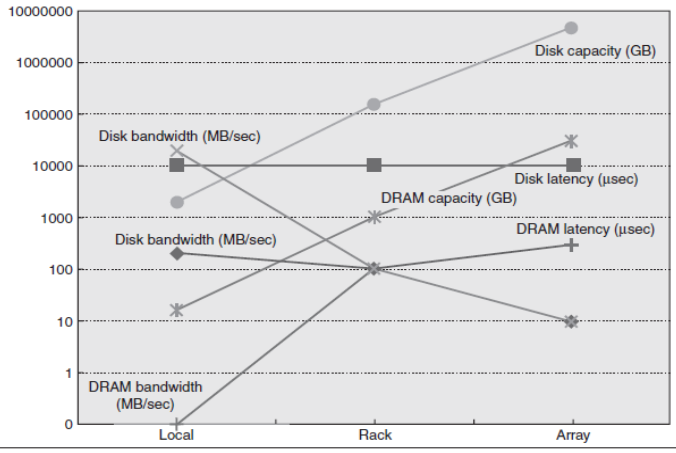


Figure 6.7 Graph of latency, bandwidth, and capacity of the memory hierarchy of a WSC for data in Figure 6.6

Each server contains 16 GBytes of memory with a 100-nanosecond access time and transfers at 20 Bytes/sec and 2 terabytes of disk that offers a 10-millisecond access time and transfers at 200 Bytes/sec. There are two sockets per board, and they share one 1 Gbit/sec Ethernet port.

■ Every pair of racks includes one rack switch and holds 80 2U servers. Networking software plus switch overhead increases the latency to DRAM to 100 microseconds and the disk access latency to 11 milliseconds. Thus, the total storage capacity of a rack is roughly 1 terabyte of DRAM and 160 terabytes of disk storage. The 1 Gbit/sec Ethernet limits the remote bandwidth to DRAM or disk within the rack to 100 MBytes/sec.

■ The array switch can handle 30 racks, so storage capacity of an array goes up by a factor of 30: 30 terabytes of DRAM and 4.8 petabytes of disk. The array switch hardware and software increases latency to DRAM within an array to 500 microseconds and disk latency to 12 milliseconds. The bandwidth of the array switch limits the remote bandwidth to either array DRAM or array disk to 10 MBytes/sec.

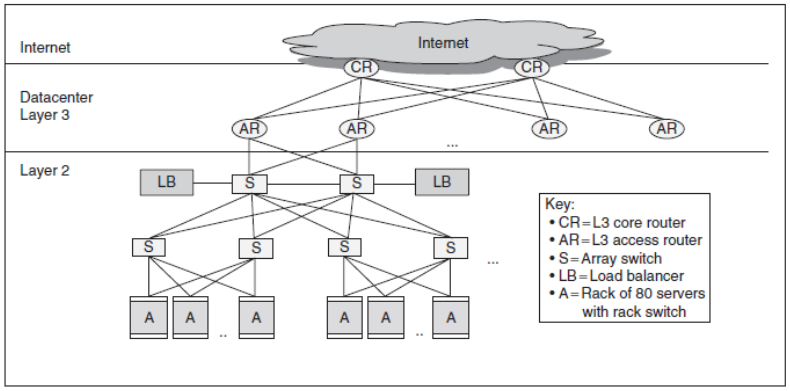


Figure 6.8 The Layer 3 network used to link arrays together and to the Internet [Greenberg et al. 2009]. Some WSCs use a separate *border router* to connect the Internet to the datacenter Layer 3 switches.

Figures 6.6 and 6.7 show that network overhead dramatically increases latency from local DRAM to rack DRAM and array DRAM, but both still have more than 10 times better latency than the local disk. The network collapses the difference in bandwidth between rack DRAM and rack disk and between array DRAM and array disk. The WSC needs 20 arrays to reach 50,000 servers, so there is one more level of the

networking hierarchy. Figure 6.8 shows the conventional Layer 3 routers to connect the arrays together and to the Internet. Most applications fit on a single array within a WSC. Those that need more than one array use sharding or partitioning, meaning that the dataset is split into independent pieces and then distributed to different arrays. Operations on the whole dataset are sent to the servers hosting the pieces, and the results are coalesced by the client computer.

other Message-Passing Multiprocessors

Message passing is a technique for invoking behavior (i.e., running a program) on a computer. In contrast to the traditional technique of calling a program by name, message passing uses an object model to distinguish the general function from the specific implementations. The invoking program sends a message and relies on the object to select and execute the appropriate code. The justifications for using an intermediate layer essentially falls into two categories: encapsulation and distribution.

Encapsulation is the idea that software objects should be able to invoke services on other objects without knowing or caring about how those services are implemented. Encapsulation can reduce the amount of coding logic and make systems more maintainable. E.g., rather than having IF-THEN statements that determine which subroutine or function to call a developer can just send a message to the object and the object will select the appropriate code based on its type.

One of the first examples of how this can be used was in the domain of computer graphics. There are all sorts of complexities involved in manipulating graphic objects. For example, simply using the right formula to compute the area of an enclosed shape will vary depending on if the shape is a triangle, rectangle, eclipse, or circle. In traditional computer programming this would result in long IF-THEN statements testing what sort of object the shape was and calling the appropriate code. The object-oriented way to handle this is to define a class called Shape with subclasses such as Rectangle and Ellipse (which in turn have subclasses Square and Circle) and then to simply send a message to any Shape asking it to compute its area. Each Shape object will then invoke the subclass's method with the formula appropriate for that kind of object.

Distributed message passing provides developers with a layer of the architecture that provides common services to build systems made up of sub-systems that run on disparate computers in different locations and at different times. When a distributed object is sending a message, the messaging layer can take care of issues such as:

- Finding the app using different operating systems and programming languages, at different locations from where the message originated.
- Saving the message on a queue if the appropriate object to handle the message is not currently running and then invoking the message when the object is available. Also, storing the result if needed until the sending object is ready to receive it.
- Controlling various transactional requirements for distributed transactions, e.g. ACID-testing the data.

UNIT V MEMORY AND I/O SYSTEMS

Memory Hierarchy - memory technologies – cache memory – measuring and improving cache performance – virtual memory, TLB's – Accessing I/O Devices – Interrupts – Direct Memory Access – Bus structure – Bus operation – Arbitration – Interface circuits - USB.

MEMORY HIERARCHY

- A memory hierarchy consists of multiple levels of memory with different speeds and sizes. The faster memories are more expensive per bit than the slower memories and thus are smaller.
- A structure that uses multiple levels of memories; as the distance from the processor increases, the size of the memories and the access time both increase.

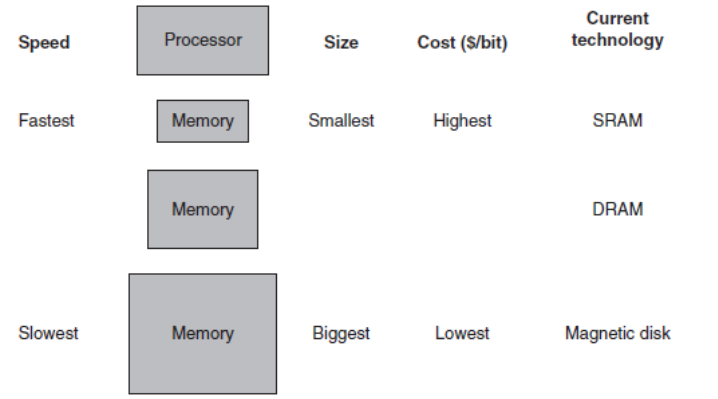


Fig:1 The basic structure of memory hierarchy

Today, there are three primary technologies used in building memory hierarchies.

- Main memory is implemented from DRAM (dynamic random access memory), while levels closer to the processor (caches) use SRAM (static random access memory). The third technology, used to implement the largest and slowest level in the hierarchy, is usually magnetic disk. (Flash memory is used instead of disks in many embedded devices)
- DRAM is less costly per bit than SRAM, although it is substantially slower. The price difference arises because DRAM uses significantly less area per bit of memory, and DRAMs thus have larger capacity.
- Because of the differences in cost and access time, it is advantageous to build memory as a hierarchy of levels. In fig 1 it shows the faster memory is close to the processor and the slower, less expensive memory is below it.
- The goal is to present the user with as much memory as is available in the cheapest technology, while providing access at the speed offered by the fastest memory.
- The data is similarly hierarchical: a level closer to the processor is generally a subset of any level further away, and all the data is stored at the lowest level.
- The upper level—the one closer to the processor—is smaller and faster than the lower level, since the upper level uses technology that is more expensive.
- The below fig shows that the minimum unit of information that can be either present or not present in the two-level hierarchy is called a **block** or a **line**.

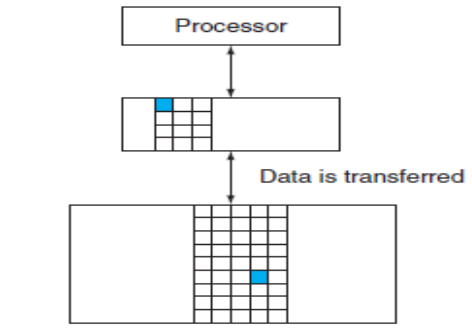


Fig 2 Every pair of levels in the memory hierarchy can be thought of as having an upper and lower level. Transfer an entire block when we copy something between levels.

- **Hit**: If the data requested by the processor appears in some block in the upper level,this is called a *hit*..
- **Miss**: If the data is not found in the upper level, the request is called a *miss*.
- The lower level in the hierarchy is then accessed to retrieve the block containing the requested data.
- The **hit rate**, or *hit ratio*, is the fraction of memory accesses found in the upper level; it is often used as a measure of the performance of the memory hierarchy.
- The **miss rate** (1 hit rate) is the fraction of memory accesses not found in the upper level.

Since performance is the major reason for having a memory hierarchy, the time to service hits and misses is important:

- **Hit time** is the time to access the upper level of the memory hierarchy, which includes the time needed to determine whether the access is a hit or a miss.
- The **miss penalty** is the time to replace a block in the upper level with the corresponding block from the lower level, plus the time to deliver this block to the processor.
- Because the upper level is smaller and built using faster memory parts, the hit time will be much smaller than the time to access the next level in the hierarchy, which is the major component of the miss penalty.
- Programs exhibit both temporal locality, the tendency to reuse recently accessed data items, and spatial locality, the tendency to reference data items that are close to other recently accessed items.
- Memory hierarchies take advantage of temporal locality by keeping more recently accessed data items closer to the processor. Memory hierarchies take advantage of spatial locality by moving blocks consisting of multiple contiguous words in memory to upper levels of the hierarchy.

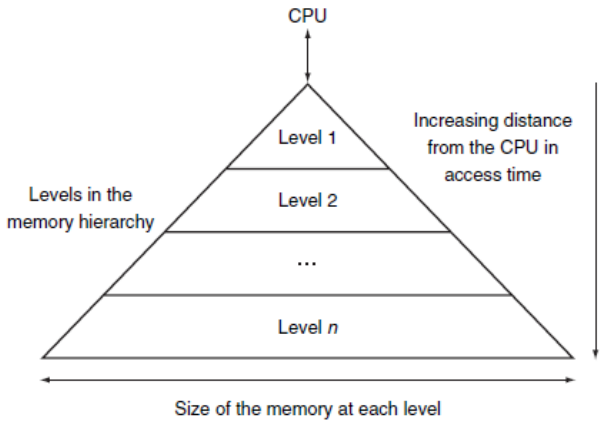


Fig.3 This diagram shows the structure of a memory hierarchy: as the distance from the processor increases, so does the size.

- The above fig shows that a memory hierarchy uses smaller and faster memory technologies close to the processor. Thus, accesses that hit in the highest level of the hierarchy can be processed

quickly. Accesses that miss go to lower levels of the hierarchy, which are larger but slower.

- If the hit rate is high enough, the memory hierarchy has an effective access time close to that of the highest (and fastest) level and a size equal to that of the lowest level.
- In most systems, the memory is a true hierarchy, meaning that data cannot be present in level *i* unless it is also present in level *i* + 1.

MEMORY TECHNOLOGIES

SRAM Technology

1) The first letter of SRAM stands for *static*.
2) The dynamic nature of the circuits in DRAM requires data to be written back after being read—hence the difference between the access time and the cycle time as well as the need to refresh.
3) SRAMs don't need to refresh and so the access time is very close to the cycle time. SRAMs typically use six transistors per bit to prevent the information from being disturbed when read.
4) SRAM needs only minimal power to retain the charge in standby mode. SRAM designs are concerned with speed and capacity, while in DRAM designs the emphasis is on cost per bit and capacity.
5) For memories designed in comparable technologies, the capacity of DRAMs is roughly 4–8 times that of SRAMs. The cycle time of SRAMs is 8–16 times faster than DRAMs, but they are also 8–16 times as expensive.

DRAM Technology

1. As early DRAMs grew in capacity, the cost of a package with all the necessary address lines was an issue. The solution was to multiplex the address lines, thereby cutting the number of address pins in half.
2. One-half of the address is sent first, called the *row accessstrobe* (RAS). The other half of the address, sent during the *column access strobe* (CAS), follows it.
3. These names come from the internal chip organization, since the memory is organized as a rectangular matrix addressed by rows and columns.

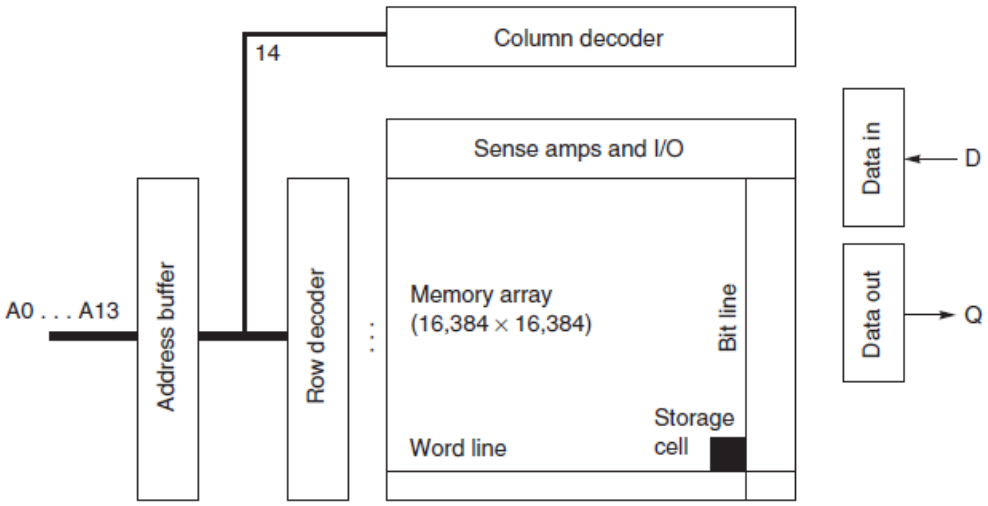


Figure 5.12 Internal organization of a 64M bit DRAM. DRAMs often use banks of memory arrays internally and select between them. For example, instead of one 16,384 × 16,384 memory, a DRAM might use 256 1024 × 1024 arrays or 16 2048 × 2048 arrays.

4.
5. An additional requirement of DRAM derives from the property signified by its first letter, *D,* for *dynamic.*
6. To pack more bits per chip, DRAMs use only a single transistor to store a bit. Reading that bit destroys the information, so it must be restored. This is one reason the DRAM cycle time is much longer than the access time.
7. In addition, to prevent loss of information when a bit is not read or written, the bit must be "refreshed" periodically. Fortunately, all the bits in a row can be refreshed simultaneously just by

reading that row. Hence, every DRAM in the memory system must access every row within a certain time window, such as 8 ms.

8. Memory controllers include hardware to refresh the DRAMs periodically. This requirement means that the memory system is occasionally unavailable because it is sending a signal telling every chip to refresh. The time for a refresh is typically a full memory access (RAS and CAS) for each row of the DRAM. Since the memory matrix in a DRAM is conceptually square, the number of steps min a refresh is usually the square root of the DRAM capacity.
9. DRAM designers try to keep time spent refreshing to less than 5% of the total time. So far we have presented main memory as if it operated like a Swiss train, consistently delivering the goods exactly according to schedule.
10. Refresh belies that analogy, since some accesses take much longer than others do. Thus, refresh is another reason for variability of memory latency and hence cache miss penalty.
11. Although we have been talking about individual chips, DRAMs are commonly sold on small boards called *dual inline memory modules* (DIMMs). DIMMs typically contain 4–16 DRAMs, and they are normally organized to be 8 bytes wide (+ ECC) for desktop systems.

Improving Memory Performance inside a DRAM Chip

1. To improve bandwidth, there has been a variety of evolutionary innovations over time.
2. The first was timing signals that allow repeated accesses to the row buffer without another row access time, typically called fast page mode. Such a buffer comes naturally, as each array will buffer 1024–2048 bits for each access. Conventional DRAMs had an asynchronous interface to the memory controller, and hence every transfer involved overhead to synchronize with the controller.
3. The second major change was to add a clock signal to the DRAM interface, so that the repeated transfers would not bear that overhead. Synchronous DRAM (SDRAM) is the name of this optimization. SDRAMs typically also had a programmable register to hold the number of bytes requested, and hence can send many bytes over several cycles per request.
4. The third major DRAM innovation to increase bandwidth is to transfer data on both the rising edge and falling edge of the DRAM clock signal, thereby doubling the peak data rate. This optimization is called double data rate (DDR).

Measuring and improving the performance of cache memory.

CACHE PERFORMANCE

Cache Performance can be improved by

- ***Reducing the miss rate***: larger block size, larger cache size, and higher associativity
- ***Reducing the miss penalty***: multilevel caches and giving reads priority over writes
- ***Reducing the time to hit in the cache***: avoiding address translation when indexing the cache

Average memory access time (AMAT) is a useful measure to evaluate the performance of a memory-hierarchy configuration.

AMAT=Time for a hit +miss rate* miss penalty

Classifying Misses: 3 Cs

Compulsory

- ✓ Cold start misses or first reference misses: The first access to a block can NOT be in the cache, so there must be a compulsory miss.
- ✓ These are suffered regardless of cache size.

Capacity

- ✓ If the cache is too small to hold all of the blocks needed during execution of a program, misses occur on blocks that were discarded earlier.
- ✓ In other words, this is the difference between the compulsory miss rate and the miss rate of a finite size fully associative cache.

Conflict

- ✓ If the cache has sufficient space for the data, but the block can NOT be kept because the set is full, a conflict miss will occur.
- ✓ This is the difference between the miss rate of a non-fully associative cache and a fully-

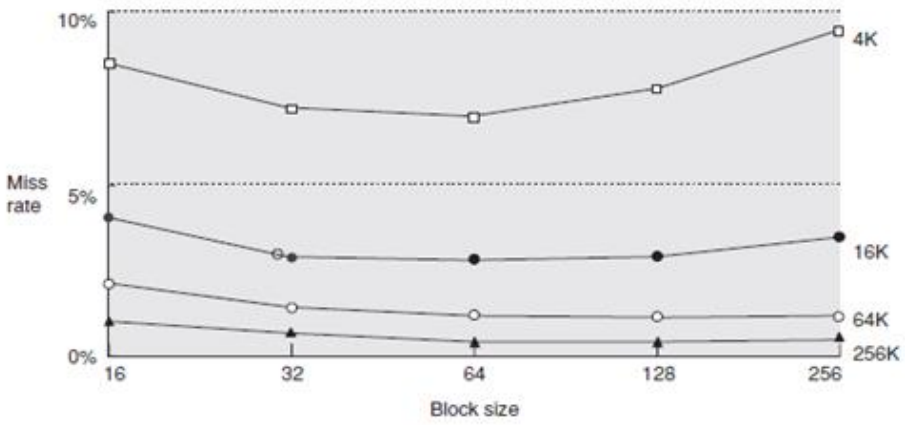
associative cache.
- ✓ These misses are also called collision or interference misses.

Six Basic Cache Optimizations (6 ways to improve cache performance)

First Optimization: Larger Block Size to Reduce Miss Rate
Second Optimization: Larger Caches to Reduce Miss Rate
Third Optimization: Higher Associativity to Reduce Miss Rate
Fourth Optimization: Multilevel Caches to Reduce Miss Penalty
Fifth Optimization: Giving Priority to Read Misses over Writes to Reduce Miss Penalty
Sixth Optimization: Avoiding Address Translation during Indexing of the Cache to Reduce Hit Time

First Optimization: Larger Block Size to Reduce Miss Rate

The simplest way to reduce miss rate is to increase the block size. Larger block sizes will reduce also compulsory misses. This reduction occurs because the principle of locality has two components: temporal locality and spatial locality.



Miss rate versus block size for five different-sized caches. Note that miss rate actually goes up if the block size is too large relative to the cache size. Each line represents a cache of different size. Figure C.11 shows the data used to plot these lines. Unfortunately, SPEC2000 traces would take too long if block size were included, so these data are based on SPEC92 on a DECstation 5000 [Gee et al. 1993].

- ➢ Larger blocks take advantage of spatial locality. At the same time, larger blocks increase the miss penalty. Since they reduce the number of blocks in the cache, larger blocks may increase conflict misses and even capacity misses if the cache is small.
- ➢ Clearly, there is little reason to increase the block size to such a size that it increases the miss rate. There is also no benefit to reducing miss rate if it increases the average memory access time.

Second Optimization: Larger Caches to Reduce Miss Rate

When a miss occurs, the cache controller must select a block to be replaced with the desired data. A benefit of direct-mapped placement is that hardware decisions are simplified. Only one block frame is checked for a hit, and only that block can be replaced. With fully associative or set-associative placement, there are many blocks to choose from on a miss. There are three primary strategies employed for selecting which block to replace:

Random:
- ✓ To spread allocation uniformly, candidate blocks are randomly selected.
- ✓ Some systems generate pseudorandom block numbers to get reproducible behavior, which is particularly useful when debugging hardware.

Least-recently used(LRU):
- ✓ To reduce the chance of throwing out information that will be needed soon, accesses to blocks are recorded.
- ✓ Relying on the past to predict the future, the block replaced is the one that has been unused for the longest time.

LRU relies on a corollary of locality:
- ✓ If recently used blocks are likely to be used again, then a good candidate for disposal is the least-recently used block.

First in, first out (FIFO):

- ✓ Because LRU can be complicated to calculate, this approximates LRU by determining the *oldest* block rather than the LRU.
- ✓ A virtue of random replacement is that it is simple to build in hardware.
- ✓ As the number of blocks to keep track of increases, LRU becomes increasingly expensive and is frequently only approximated.

Third Optimization: Higher Associativity to Reduce Miss Rate

To show the benefit of associativity, conflict misses are divided into misses caused by each decrease in associativity.

Here are the four divisions of conflict misses and how they are calculated:

Eight-way: Conflict misses due to going from fully associative (no conflicts) to eight-way associative

Four-way: Conflict misses due to going from eight-way associative to four way associative

Two-way: Conflict misses due to going from four-way associative to two way associative

One-way: Conflict misses due to going from two-way associative to one way associative (direct mapped)

Fourth Optimization: Multilevel Caches to Reduce Miss Penalty

Let's start with the definition of *average memory access time* for a two-level cache. Using the subscripts L1 and L2 to refer respectively, to a first-level and a second-level cache, the original formula is

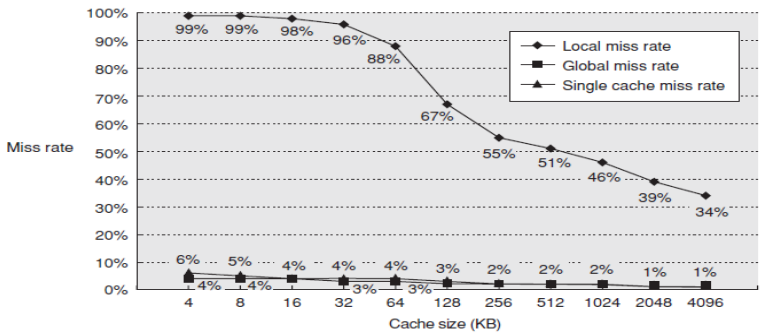
Average memory access time = Hit timeL1 + Miss rateL1 × Miss penaltyL1 and
Miss penaltyL1 = Hit timeL2 + Miss rateL2 × Miss penaltyL2, so

Average memory access time = Hit timeL1 + Miss rateL1 × (Hit timeL2 + Miss rateL2 × Miss penaltyL2)

In this formula, the second-level miss rate is measured on the leftovers from the first-level cache.

To avoid ambiguity, these terms are adopted here for a two-level cache system:

- ✓ ***Local miss rate***: This rate is simply the number of misses in a cache divided by the total number of memory accesses to this cache.
- ✓ ***Global miss rate***: The number of misses in the cache divided by the total number of memory accesses generated by the processor. Using the terms above, the global miss rate for the first-level cache is still just Miss rate L1, but for the second-level cache it is Miss rate L1 × Miss rate L2.



Miss rates versus cache size for multilevel caches

Fifth Optimization: Giving Priority to Read Misses over Writes to Reduce Miss Penalty

This optimization serves reads before writes have been completed. The simplest way out of this dilemma is for the read miss to wait until the write buffer is empty. The alternative is to check the contents of the write buffer on a read miss, and if there are no conflicts and the memory system is available, let the read miss continue. Virtually all desktop and server processors use the latter approach, giving reads priority over writes. The cost of writes by the processor in a write-back cache can also be reduced. Suppose a read miss will replace a dirty memory block. Instead of writing the dirty block to memory, and then reading memory, we could copy the dirty block to a buffer, then read memory, and *then* write memory. This way the processor read, for which the processor is probably waiting, will finish sooner. Similar to the previous situation, if a read miss occurs, the processor can either stall until the buffer is empty or check the addresses of the words in the buffer for conflicts. Now that we have five optimizations that reduce cache miss penalties or miss rates, it is time to look at reducing the final component of average memory access time. Hit time is critical because it can affect the clock rate of the processor; in many processors today the cache access time limits the clock cycle rate, even for processors that take multiple clock cycles to access the cache. Hence, a fast hit time is multiplied in importance beyond the average memory access time formula because it helps everything.

Sixth Optimization: Avoiding Address Translation during Indexing of the Cache to Reduce Hit Time

Page tables are usually so large that they are stored in main memory and are sometimes paged themselves. Paging means that every memory access logically takes at least twice as long, with one memory access to obtain the physical address and a second access to get the data. We use locality to avoid the extra memory access. By keeping address translations in a special cache, a memory access rarely requires a second access to translate the data. This special address translation cache is referred to as a *translation lookasidebuffer* (TLB), also called a *translation buffer* (TB). A TLB entry is like a cache entry where the tag holds portions of the virtual address and the data portion holds a physical page frame number, protection field, valid bit, and usually a use bit and dirty bit. To change the physical page frame number or protection of an entry in the page table, the operating system must make sure the old entry is not in the TLB; otherwise, the system won't behave properly. Note that this dirty bit means the corresponding *page* is dirty, not that the address translation in the TLB is dirty nor that a particular block in the data cache is dirty. The operating system resets these bits by changing the value in the page table and then invalidates the corresponding TLB entry. When the entry is reloaded from the page table, the TLB gets an accurate copy of the bits.

Cache Memory

The Cache memory stores a reasonable number of blocks at a given time but this number is small compared to the total number of blocks available in Main Memory. The correspondence between main memory block and the block in cache memory is specified by a mapping function. The Cache control hardware decides that which block should be removed to create space for the new block that contains the referenced word. The collection of rule for making this decision is called the **replacement algorithm.**

Mapping Function
Direct Mapping

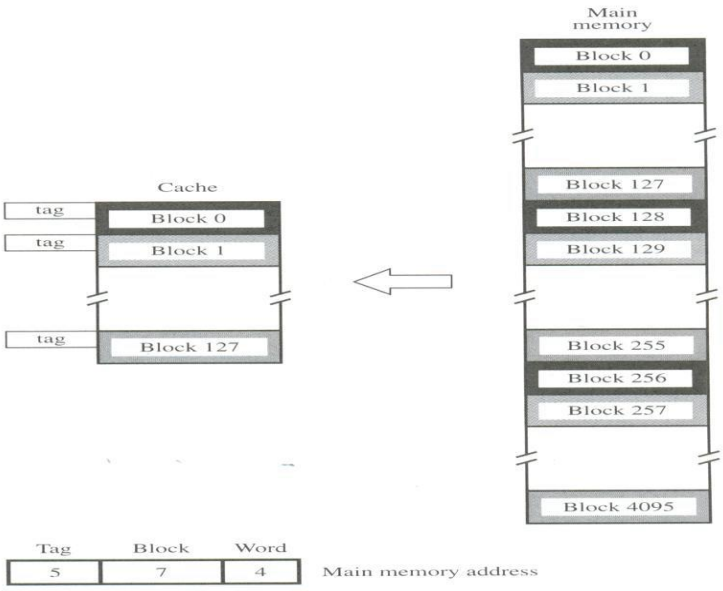


Fig: Direct Mapped Cache

It is the simplest technique in which block j of the main memory maps onto block „j" modulo 128 of the cache. Thus whenever one of the main memory blocks 0,128,256 is loaded in the cache,it is stored in block 0. Block 1,129,257 are stored in cache block 1 and so on. The contention may arise when,

- When the cache is full
- When more than one memory block is mapped onto a given cache block position.

The contention is resolved by allowing the new blocks to overwrite the currently resident block. Placement of block in the cache is determined from memory address. The memory address is divided into 3 fields. They are,

- **Low Order 4 bit field(word)->**Selects one of 16 words in a block.
- **7 bit cache block field->**When new block enters cache,7 bit determines the cache position in which this block must be stored.
- **5 bit Tag field->**The high order 5 bits of the memory address of the block is stored in 5 tag

bits associated with its location in the cache.

As execution proceeds, the high order 5 bits of the address is compared with tag bits associated with that cache location. If they match, then the desired word is in that block of the cache. If there is no match, then the block containing the required word must be first read from the main memory and loaded into the cache.

Merit:

It is easy to implement

Demerit:

It is not very flexible.

Associative Mapping:

In this method, the main memory block can be placed into any cache block position. 12 tag bits will identify a memory block when it is resolved in the cache. The tag bits of an address received from the processor are compared to the tag bits of each block of the cache to see if the desired block is persent. This is called **associative mapping. I**t gives complete freedom in choosing the cache location. A new block that has to be brought into the cache has to replace(eject)an existing block if the cache is full. In this method, the memory has to determine whether a given block is in the cache. A search of this kind is called an **associative Search.**

Merit:

It is more flexible than direct mapping technique.

Demerit:

Its cost is high.

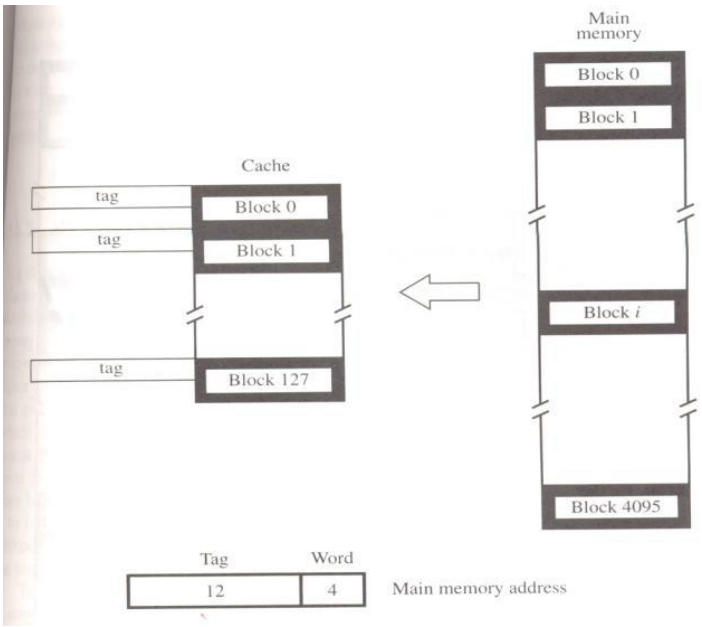


Fig: Associative Mapped Cache

Set-Associative Mapping:

It is the combination of direct and associative mapping. The blocks of the cache are grouped into sets and the mapping allows a block of the main memory to reside in any block of the specified set. In this case, the cache has two blocks per set, so the memory blocks 0, 64,128……..4032 maps into cache set „0" and they can occupy either of the two block position within the set. 6 bit set field->Determines which set of cache contains the desired block. 6 bit tag field. The tag field of the address is compared to the tags of the two blocks of the set to clock if the desired block is present.

No of blocks per set	No of set field
2	6
3	5
8	4
128	no set field

The cache which contains 1 block per set is called **Direct Mapping.** A cache that has „k" blocks per set is called as „**k-way set associative cache**". Each block contains a control bit called a **valid bit**. The Valid bit indicates that whether the block contains valid data. The dirty bit indicates that whether the block has been modified during its cache residency. Valid bit=0->When power is initially applied to system. Valid bit =1->When the block is loaded from main memory at first time. If the main memory block is updated by a source & if the block in the source is already exists in the cache,then the valid bit will be cleared to „0". If Processor & DMA uses the same copies of data then it is called as the **Cache Coherence Problem.**

Merit:

- The Contention problem of direct mapping is solved by having few choices for block placement.
- The hardware cost is decreased by reducing the size of associative search

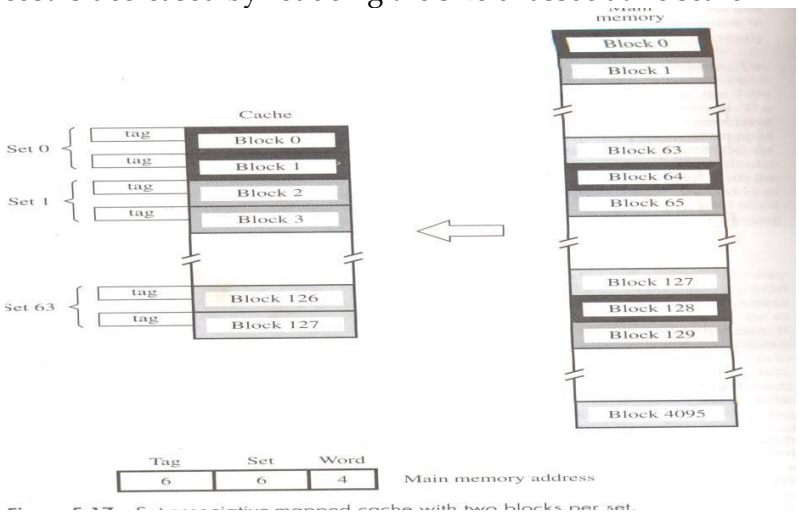


Fig: Set-Associative Mapping

ADDRESS TRANSLATION OF VIRTUAL MEMORY

Address Translation:

In address translation, all programs and data are composed of fixed length units called **Pages.** The Page consists of a block of words that occupy contiguous locations in the main memory. The pages are commonly range from 2K to 16K bytes in length. The **cache bridge** speed up the gap between main memory and secondary storage and it is implemented in software techniques. Each virtual address generated by the processor contains V**irtual Page Number** (Low order bit) and **offset** (High order bit). Virtual Page number+ Offset specifies the location of a particular byte (or word) within a page.

Page Table:

It contains the information about the main memory address where the page is stored & the current status of the page.

Page Frame:

An area in the main memory that holds one page is called the page frame.

Page Table Base Register:

It contains the starting address of the page table. **Virtual Page Number + Page Table Base register,** Gives the address of the corresponding entry in the page table. (ie) it gives the starting address of the page if that page currently resides in memory.

Control Bits in Page Table:

The Control bit specifies the status of the page while it is in main memory.

Function:

The control bit indicates the validity of the page (i.e) it checks whether the page is actually loaded in the main memory. It also indicates that whether the page has been modified during its residency in the memory; this information is needed to determine whether the page should be written back to the disk before it is removed from the main memory to make room for another page. The Page table information is used by MMU for every read & write access. The Page table is placed in the main memory but a copy of the small portion of the page table is located within MMU. This small portion or small cache is called Translation **LookAside Buffer(TLB).** This portion consists of the page table enteries that corresponds to the most recently accessed pages and also contains the virtual address of the entry.

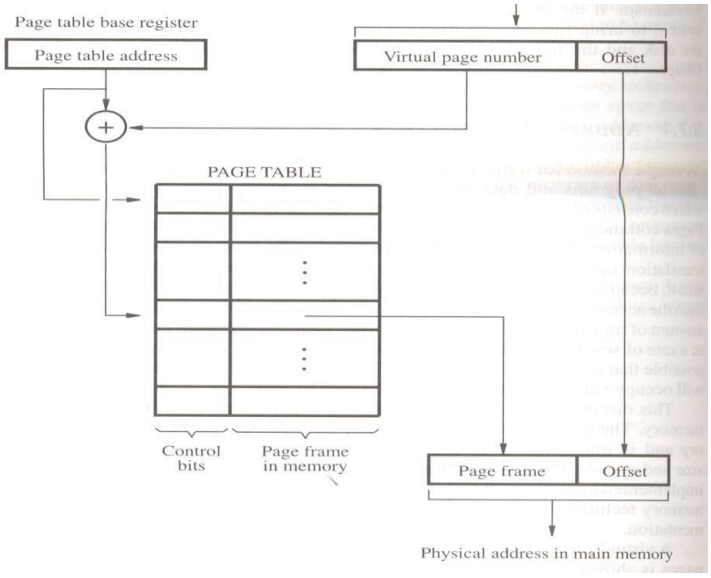


Fig: Virtual Memory Address Translation

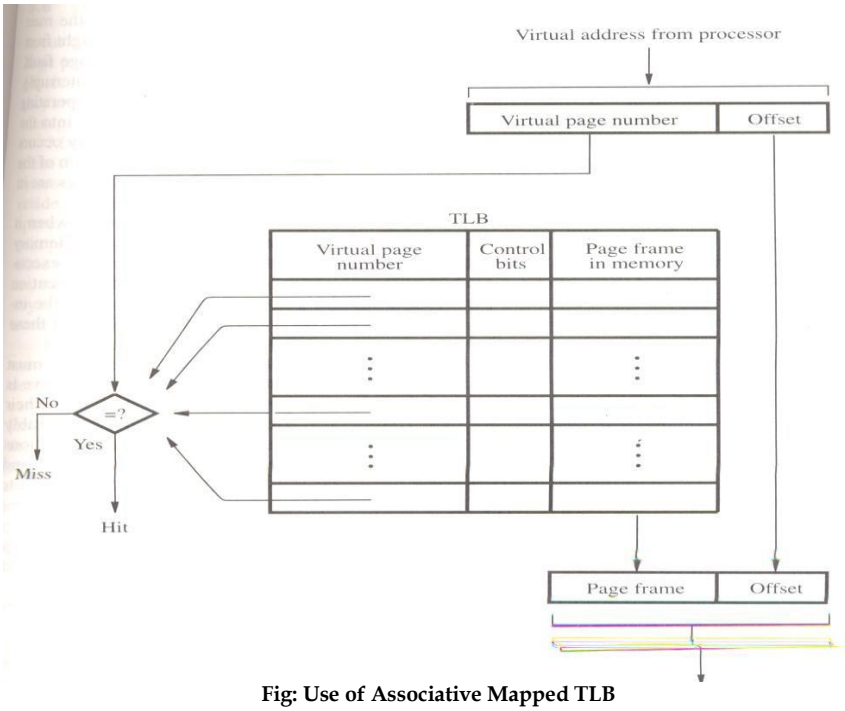


Fig: Use of Associative Mapped TLB

When the operating system changes the contents of page table, the control bit in TLB will invalidate the corresponding entry in the TLB. Given a virtual address, the MMU looks in TLB for the referenced page. If the page table entry for this page is found in TLB, the physical address is obtained immediately. If there is a miss in TLB, then the required entry is obtained from the page table in the main memory & TLB is updated. When a program generates an access request to a page that is not in the main memory, then Page Fault will occur. The whole page must be brought from disk into memry before an access can proceed. When it detects a page fault, the MMU asks the operating system to generate an interrupt. The operating System suspend the execution of the task that caused the page fault and begin execution of another task whose pages are in main memory because the long delay occurs while page transfer takes place. When the task resumes, either the interrupted instruction must continue from the point of interruption or the instruction must be restarted. If a new page is brought from the disk when the main memory is full, it must replace one of the resident pages. In that case, it uses LRU algorithm which removes the least referenced Page. A modified page has to be written back to the disk before it is removed from the main memory. In that case, write –through protocol is used.

VIRTUAL MEMORY

Techniques that automatically move program and data blocks into the physical main memory when they are required for execution is called the **Virtual Memory.** The binary address that the processor issues either for instruction or data are called the **virtual / Logical address**. The virtual address is translated into physical address by a combination of hardware and software components. This kind of address translation is done by **MMU**(Memory Management Unit). When the desired data are in the main memory, these data are fetched /accessed immediately. If the data are not in the main memory, the MMU causes the Operating system to bring the data into memory from the disk. Transfer of data between disk and main memory is performed using DMA scheme.

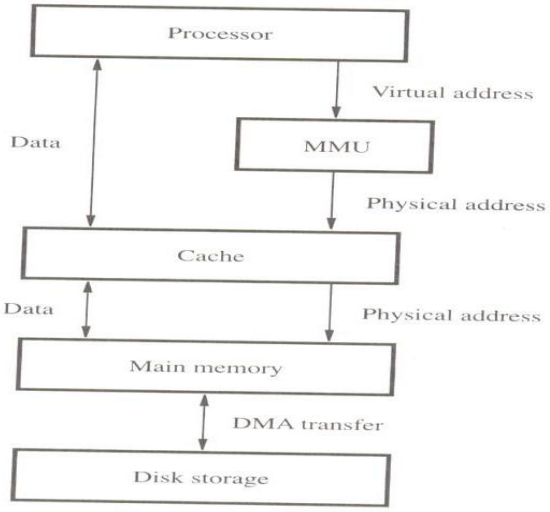


Fig: Virtual Memory Organisation

Address Translation:

In address translation, all programs and data are composed of fixed length units called **Pages.** The Page consists of a block of words that occupy contiguous locations in the main memory. The pages are commonly range from 2K to 16K bytes in length. The **cache bridge** speed up the gap between main memory and secondary storage and it is implemented in software techniques. Each virtual address generated by the processor contains V**irtual Page Number** (Low order bit) and **offset**(High order bit). Virtual Page number+ Offset specifies the location of a particular byte (or word) within a page.

Page Table:

It contains the information about the main memory address where the page is stored & the current status of the page.

Page Frame:

An area in the main memory that holds one page is called the page frame.

Page Table Base Register:

It contains the starting address of the page table. **Virtual Page Number + Page Table Base register,** Gives the address of the corresponding entry in the page table. (ie) it gives the starting address of the page if that page currently resides in memory.

Control Bits in Page Table:

The Control bit specifies the status of the page while it is in main memory.

Function:

The control bit indicates the validity of the page ie) it checks whether the page is actually loaded in the main memory. It also indicates that whether the page has been modified during its residency in the memory; this information is needed to determine whether the page should be written back to the disk before it is removed from the main memory to make room for another page. The Page table information is used by MMU for every read & write access. The Page table is placed in the main memory but a copy of the small portion of the page table is located within MMU. This small portion or small cache is called Translation **LookAside Buffer (TLB).** This portion consists of the page table enteries that corresponds to the most recently accessed pages and also contains the virtual address of the entry.

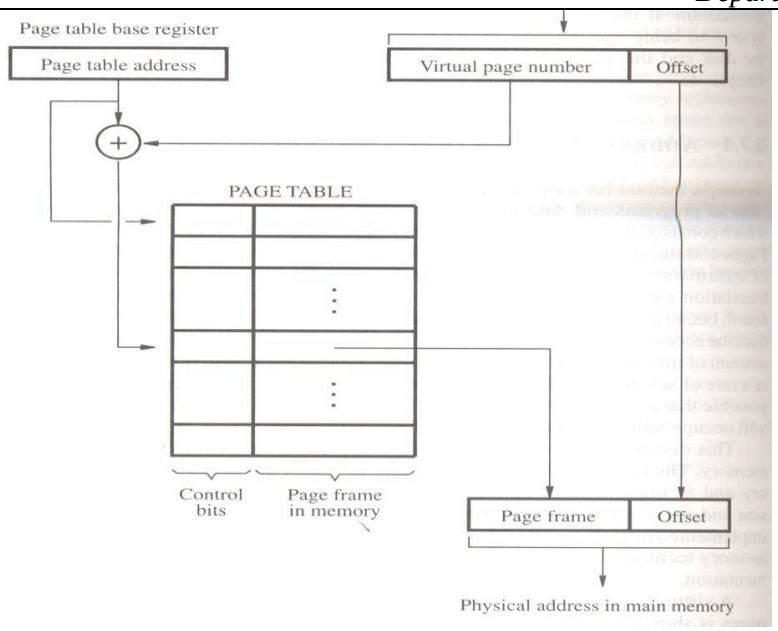


Fig: Virtual Memory Address Translation

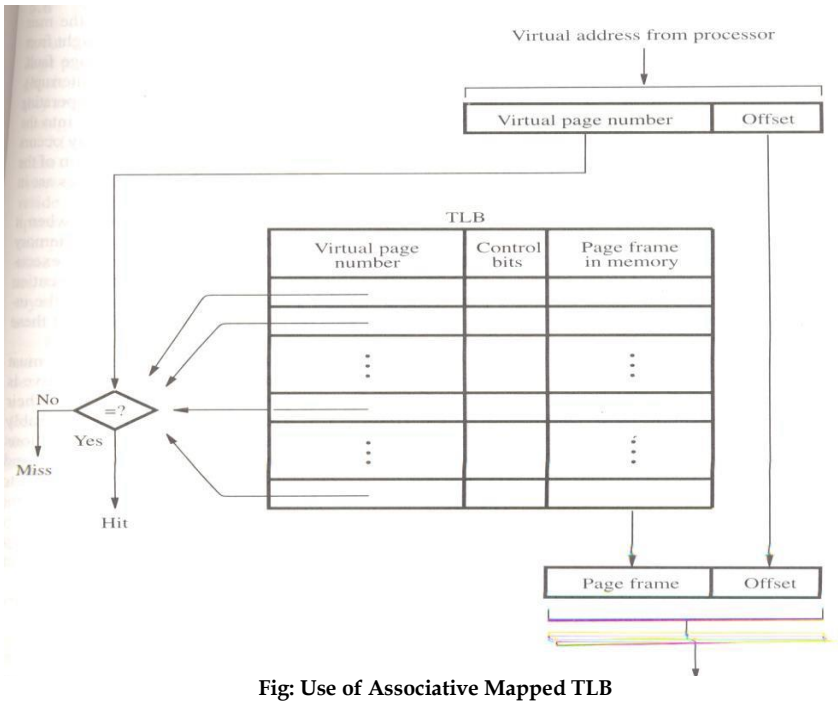


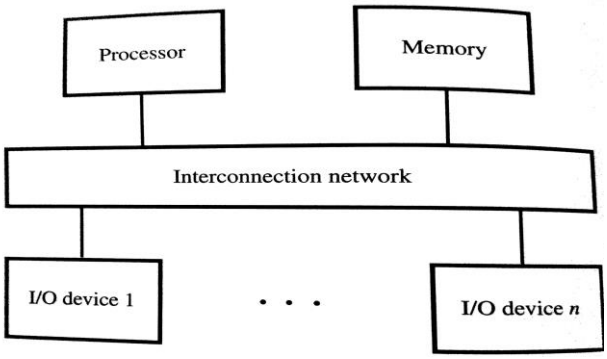
Fig: Use of Associative Mapped TLB

When the operating system changes the contents of page table, the control bit in TLB will invalidate the corresponding entry in the TLB. Given a virtual address, the MMU looks in TLB for the referenced page. If the page table entry for this page is found in TLB, the physical address is obtained immediately. If there is a miss in TLB, then the required entry is obtained from the page table in the main memory & TLB is updated. When a program generates an access request to a page that is not in the main memory, then Page Fault will occur. The whole page must be brought from disk into memory before an access can proceed. When it detects a page fault, the MMU asks the operating system to generate an interrupt. The operating System suspend the execution of the task that caused the page fault and begin execution of another task whose pages are in main memory because the long delay occurs while page transfer takes place. When the task resumes, either the interrupted instruction must continue from the point of interruption or the instruction must be restarted. If a new page is brought from the disk when the main memory is full, it must replace one of the resident pages. In that case, it uses LRU algorithm which removes the least referenced Page. A modified page has to be written back to the disk before it is removed from the main memory. In that case, write –through protocol is used.

Accessing I/O Devices

The components of a computer system communicate with each other through an interconnection network. The interconnection network consists of circuits needed to transfer information between the processor, the memory unit, and a number of I/O devices. Load and Store instructions use addressing modes to generate effective addresses that identify the desired locations. idea of using addresses to access various locations in the memory can be extended to deal with the I/O devices. Each I/O device must appear to the processor as consisting of some addressable locations, just like the memory. Some addresses in the address space of the processor are assigned to these I/O locations, rather than to the main memory.

These locations are usually implemented as bit storage circuits organized in the form of registers called I/O registers. Since the I/O devices and the memory share the same address space, this arrangement is called memory-mapped I/O. With memory-mapped I/O, any machine instruction that can access memory can be used to transfer data to or from an I/O device.



A computer system

For example, if DATAIN is the address of a register in an input device, the instruction

Load R2, DATAIN reads the data from the DATAIN register and loads them into processor register R2. Similarly, the instruction

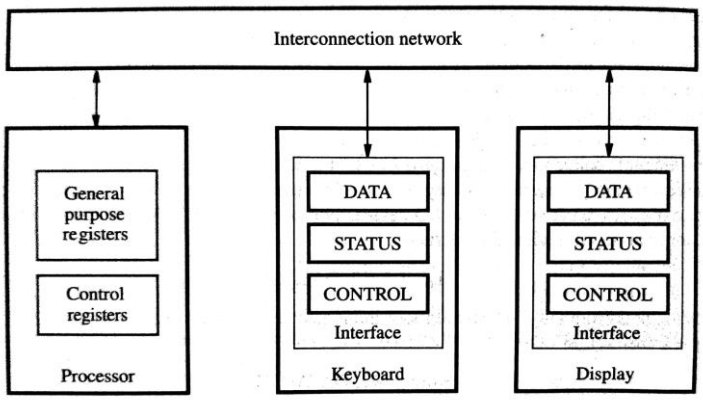
Store R2, DATAOUT sends the contents of register R2 to location DATAOUT, which is a register in an output device.

I/O Device Interface

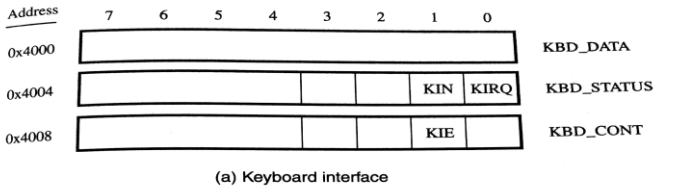
An I/O device is connected to the interconnection network by using a circuit, called the device interface, which provides the means for data transfer. The interface includes some registers that can be accessed by the processor. One register may serve as a buffer for data transfers, another may hold information about the current status of the device, and yet another may store the information that controls the operational behaviour of the device. These data, status, and control registers are accessed by program instructions as if they were memory locations. Typical transfers of information are between I/O registers and the registers in the processor.

Program-Controlled I/O:

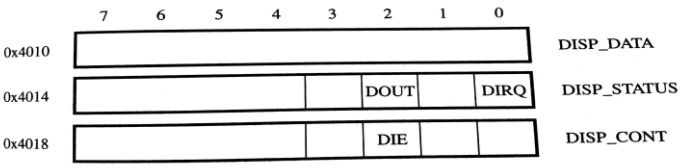
Consider a task that reads characters typed on a keyboard, stores these data in the memory, and displays the same characters on a display screen. A simple way of implementing this task is to write a program that performs all functions needed to realize the desired action. This method is known as program-controlled I/O. In addition to transferring each character from the keyboard into the memory, and then to the display, it is necessary to ensure that this happens at the right time. An input character must be read in response to a key being pressed. For output, a character must be sent to the display only when the display device is able to accept it. The rate of data transfer from the keyboard to a computer is limited by the typing speed of the user, which is unlikely to exceed a few characters per second. The rate of output transfers from the computer to the display is much higher. It is determined by the rate at which characters can be transmitted to and displayed on the display device The difference in speed between the processor and I/O devices creates the need for mechanisms to synchronize the transfer of data between them.



One solution to this problem involves a signaling protocol. KBD_DATA be the address label of an 8-bit register that holds the generated character. Also, let a signal indicating that a key has been pressed be provided by setting to 1 a flip-flop called KIN, which is a part of an eight-bit status register, KBD_STATUS. The processor can read the *status flag* KIN to determine when a character code has been placed in KBD_DATA. When the processor reads the status flag to determine its state, we say that the processor *polls* the I/O device. The display includes an 8-bit register, which we will call DISP_DATA, used to receive characters from the processor. It also must be able to indicate that it is ready to receive the next character, this can be done by using a status flag called DOUT, which is one bit in a status register, DISP_STATUS.



(a) Keyboard interface



(b) Display interface

INTERRUPTS

An interrupt is an event that causes the execution of one program to be suspended and the execution of another program to begin. In program- controlled I/O, when the processor continuously monitors the status of the device, the processor will not perform any function. An alternate approach would be for the I/O device to alert the processor when it becomes ready. The Interrupt request line will send a hardware signal called the interrupt signal to the processor. On receiving this signal, the processor will perform the useful function during the waiting period. The routine executed in response to an interrupt request is called Interrupt Service Routine. The interrupt resembles the subroutine calls.

The processor first completes the execution of instruction i. Then it loads the PC (Program Counter) with the address of the first instruction of the ISR. After the execution of ISR, the processor has to come back to instruction i + 1. Therefore, when an interrupt occurs, the current contents of PC which point to i +1 is put in temporary storage in a known location. A return from interrupt instruction at the end of ISR reloads the PC from that temporary storage location, causing the execution to resume at instruction i+1. When the processor is handling the interrupts, it must inform the device that its request has been recognized so that it removes its interrupt request signal. This may be accomplished by a special control signal called the interrupt acknowledge signal. The task of saving and restoring the information can be done automatically by the processor. The processor saves only the contents of program counter & status register (ie) it saves only the minimal amount of information to maintain the integrity of the program execution. Saving registers also increases the delay between the time an interrupt request is received and the start of the execution of the ISR. This delay is called the Interrupt Latency. Generally, the long interrupt latency in unacceptable. The concept of interrupts is used in Operating System and in Control Applications, where processing of certain routines must be accurately

timed relative to external events. This application is also called as real-time processing.

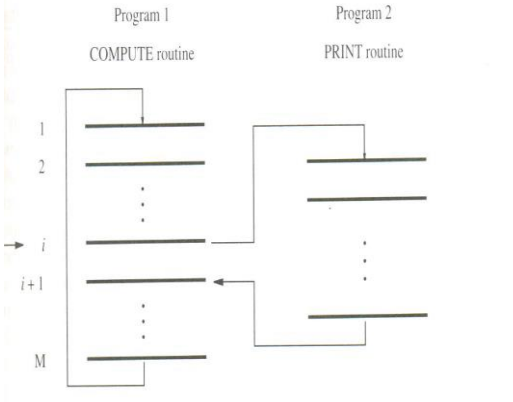


Fig: Transfer of control through the use of interrupts

The processor first completes the execution of instruction i. Then it loads the PC (Program Counter) with the address of the first instruction of the ISR. After the execution of ISR, the processor has to come back to instruction i + 1. Therefore, when an interrupt occurs, the current contents of PC which point to i +1 is put in temporary storage in a known location. A return from interrupt instruction at the end of ISR reloads the PC from that temporary storage location, causing the execution to resume at instruction i+1. When the processor is handling the interrupts, it must inform the device that its request has been recognized so that it removes its interrupt request signal. This may be accomplished by a special control signal called the interrupt acknowledge signal. The task of saving and restoring the information can be done automatically by the processor. The processor saves only the contents of program counter & status register (ie) it saves only the minimal amount of information to maintain the integrity of the program execution. Saving registers also increases the delay between the time an interrupt request is received and the start of the execution of the ISR. This delay is called the Interrupt Latency. Generally, the long interrupt latency in unacceptable. The concept of interrupts is used in Operating System and in Control Applications, where processing of certain routines must be accurately timed relative to external events. This application is also called as real-time processing.

Interrupt Hardware

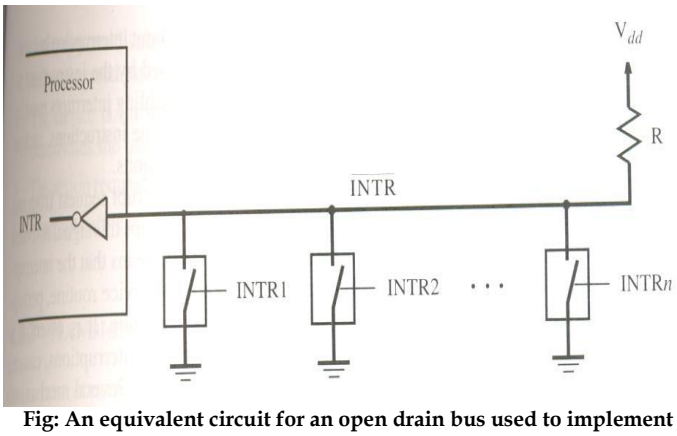


Fig: An equivalent circuit for an open drain bus used to implement a common interrupt request line.

A single interrupt request line may be used to serve n devices. All devices are connected to the line via switches to ground. To request an interrupt, a device closes its associated switch, the voltage on INTR line drops to 0(zero). If all the interrupt request signals (INTR1 to INTRn) are inactive, all switches are open and the voltage on INTR line is equal to Vdd. When a device requests an interrupts, the value of INTR is the logical OR of the requests from individual devices.

(ie)INTR = INTR1+……+INTRn

$\overline{INTR}$->It is used to name the INTR signal on common line it is active in the low voltage state.

Open collector (bipolar ckt) or Open drain (MOS circuits) is used to drive INTR line. The Output of the Open collector (or) Open drain control is equal to a switch to the ground that is open when gates input is in 0 state and closed when the gates input is in 1 state. Resistor R is called a pull-up resistor because it pulls the line voltage up to the high voltage state when the switches are open. Enabling and Disabling Interrupts: The arrival of an interrupt request from an external device causes the processor to suspend

the execution of one program & start the execution of another because the interrupt may alter the sequence of events to be executed. INTR is active during the execution of Interrupt Service Routine. There are 3 mechanisms to solve the problem of infinite loop which occurs due to successive interruptions of active INTR signals.

The following are the typical scenario.

- The device raises an interrupt request.
- The processor interrupts the program currently being executed.
- Interrupts are disabled by changing the control bits is PS (Processor Status register)
- The device is informed that its request has been recognized & in response, it deactivates the INTR signal.
- The actions are enabled & execution of the interrupted program is resumed.

Edge-triggered

The processor has a special interrupt request line for which the interrupt handling circuit responds only to the leading edge of the signal. Such a line said to be edge-triggered.

Handling Multiple Devices

When several devices requests interrupt at the same time, it raises some questions. They are

- How can the processor recognize the device requesting an interrupt?
- Given that the different devices are likely to require different ISR, how can the processor obtain the starting address of the appropriate routines in each case?
- Should a device be allowed to interrupt the processor while another interrupt is being serviced?
- How should two or more simultaneous interrupt requests be handled?

Polling Scheme:

If two devices have activated the interrupt request line, the ISR for the selected device (first device) will be completed & then the second request can be serviced. The simplest way to identify the interrupting device is to have the ISR polls all the encountered with the IRQ bit set is the device to be serviced. IRQ (Interrupt Request) -> when a device raises an interrupt requests, the status register IRQ is set to 1.

Merit:

It is easy to implement.

Demerit:

The time spent for interrogating the IRQ bits of all the devices that may not be requesting any service.

Vectored Interrupt:

Here the device requesting an interrupt may identify itself to the processor by sending a special code over the bus & then the processor start executing the ISR. The code supplied by the processor indicates the starting address of the ISR for the device. The code length ranges from 4 to 8 bits. The location pointed to by the interrupting device is used to store the staring address to ISR. The processor reads this address, called the interrupt vector & loads into PC. The interrupt vector also includes a new value for the Processor Status Register. When the processor is ready to receive the interrupt vector code, it activate the interrupt acknowledge (INTA) line.

Interrupt Nesting: Multiple Priority Scheme

In multiple level priority scheme, we assign a priority level to the processor that can be changed under program control. The priority level of the processor is the priority of the program that is currently being executed. The processor accepts interrupts only from devices that have priorities higher than its own. At the time the execution of an ISR for some device is started, the priority of the processor is raised to that of the device. The action disables interrupts from devices at the same level of priority or lower.

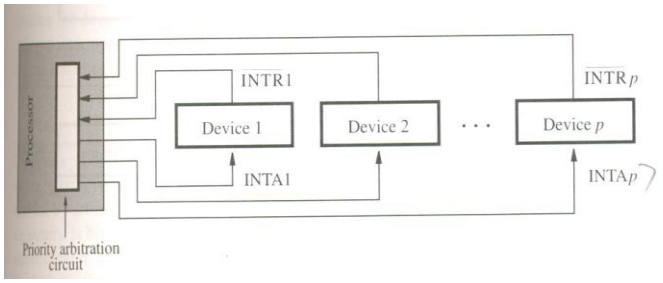
Privileged Instruction

The processor priority is usually encoded in a few bits of the Processor Status word. It can also be changed by program instruction & then it is write into PS. These instructions are called privileged instruction. This can be executed only when the processor is in supervisor mode. The processor is in supervisor mode only when executing OS routines. It switches to the user mode before beginning to execute application program.

Privileged Exception

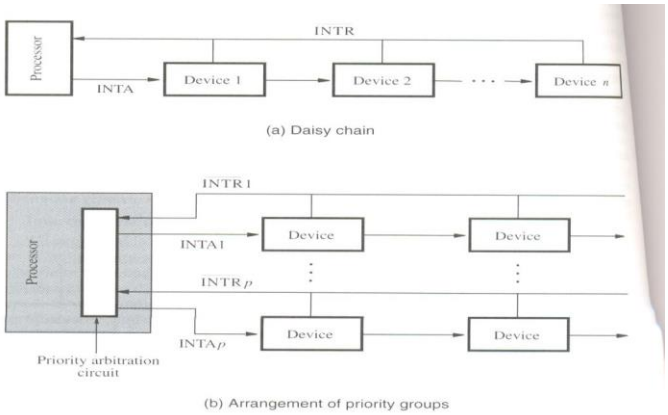
User program cannot accidently or intentionally change the priority of the processor & disrupts the system operation. An attempt to execute a privileged instruction while in user mode, leads to a special type of interrupt called the privileged exception.

Fig: Implementation of Interrupt Priority using individual Interrupt request acknowledge lines



Each of the interrupt request line is assigned a different priority level. Interrupt request received over these lines are sent to a priority arbitration circuit in the processor. A request is accepted only if it has a higher priority level than that currently assigned to the processor.

Simultaneous Requests: Daisy Chain



The interrupt request line INTR is common to all devices. The interrupt acknowledge line INTA is connected in a daisy chain fashion such that INTA signal propagates serially through the devices.

When several devices raise an interrupt request, the INTR is activated & the processor responds by setting INTA line to 1. this signal is received by device. Device1 passes the signal on to device2 only if it does not require any service. If devices1 has a pending request for interrupt blocks that INTA signal & proceeds to put its identification code on the data lines. Therefore, the device that is electrically closest to the processor has the highest priority.

Merits

It requires fewer wires than the individual connections.

Arrangement of Priority Groups

Here the devices are organized in groups & each group is connected at a different priority level. Within a group, devices are connected in a daisy chain. At the devices end, an interrupt enable bit in a control register determines whether the device is allowed to generate an interrupt requests. At the processor end, either an interrupt enable bit in the PS (Processor Status) or a priority structure determines whether a given interrupt requests will be accepted.

Initiating the Interrupt Process

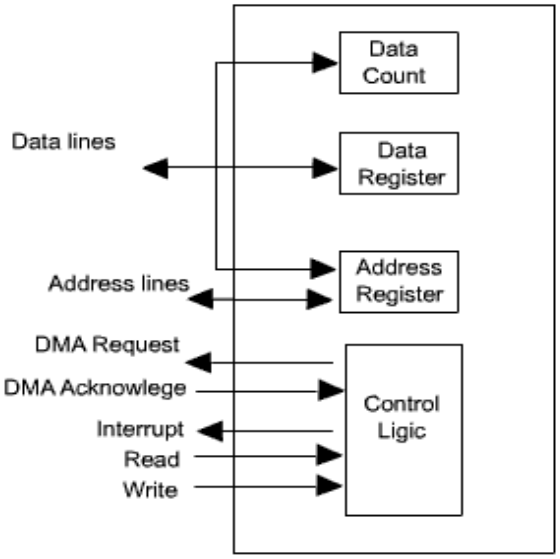
Load the starting address of ISR in location INTVEC (vectored interrupt). Load the address LINE in a memory location PNTR. The ISR will use this location as a pointer to store the i/p characters in the memory. Enable the keyboard interrupts by setting bit 2 in register CONTROL to 1. Enable interrupts in the processor by setting to 1, the IE bit in the processor status register PS.

Exception of ISR

Read the input characters from the keyboard input data register. This will cause the interface circuits to remove its interrupt requests. Store the characters in a memory location pointed to by PNTR & increment PNTR. When the end of line is reached, disable keyboard interrupt & inform program main. Return from interrupt.

DIRECT MEMORY ACCESS

A special control unit may be provided to allow the transfer of large block of data at high speed directly between the external device and main memory, without continuous intervention by the processor. This approach is called DMA. DMA transfers are performed by a control circuit called the DMA Controller.



To initiate the transfer of a block of words , the processor sends,

- ✓ Starting address
- ✓ Number of words in the block
- ✓ Direction of transfer

• When a block of data is transferred , the DMA controller increment the memory address for successive words and keep track of number of words and it also informs the processor by raising an interrupt signal.

• While DMA control is taking place, the program requested the transfer cannot continue and the processor can be used to execute another program.

• After DMA transfer is completed, the processor returns to the program that requested the transfer.

- ✓ Registers in a DMA Interface

R/W determines the direction of transfer :

- ➢ When R/W =1, DMA controller read data from memory to I/O device.
- ➢ R/W =0, DMA controller perform write operation.
- ➢ Done Flag=1, the controller has completed transferring a block of data and is ready to receive another command.
- ➢ IE=1, it causes the controller to raise an interrupt (interrupt Enabled) after it has completed transferring the block of data.
- ➢ IRQ=1, it indicates that the controller has requested an interrupt

Use of DMA controllers in a computer system

- ➢ A DMA controller connects a high speed network to the computer bus, and the disk controller for two disks, also has DMA capability and it provides two DMA channels.
- ➢ To start a DMA transfer of a block of data from main memory to one of the disks, the program write's the address and the word count information into the registers of the corresponding channel of the disk controller.
- ➢ When DMA transfer is completed, it will be recorded in status and control registers of the DMA channel (ie) Done bit=IRQ=IE=1.

Cycle Stealing:

Requests by DMA devices for using the bus are having higher priority than processor requests. Top priority is given to high speed peripherals such as, Disk and High speed Network Interface and Graphics display device. Since the processor originates most memory access cycles, the DMA controller can be said to steal the memory cycles from the processor. This interviewing technique is called Cycle stealing.

Burst Mode:

The DMA controller may be given exclusive access to the main memory to transfer a block of data without interruption. This is known as Burst/Block Mode.

Bus Master:

The device that is allowed to initiate data transfers on the bus at any given time is called the bus master

Bus Arbitration:

It is the process by which the next device to become the bus master is selected and the bus mastership is transferred to it.

Types:

There are 2 approaches to bus arbitration. They are

i)Centralized arbitration (A single bus arbiter performs arbitration)

ii)Distributed arbitration (all devices participate in the selection of next bus master).

Centralized Arbitration:

Here the processor is the bus master and it may grants bus mastership to one of its DMA controller. A DMA controller indicates that it needs to become the bus master by activating the Bus Request line (BR) which is an open drain line. The signal on BR is the logical OR of the bus request from all devices connected to it. When BR is activated the processor activates the Bus Grant Signal (BGI) and indicated the DMA controller that they may use the bus when it becomes free. This signal is connected to all devices using a daisy chain arrangement. If DMA requests the bus, it blocks the propagation of Grant Signal to other devices and it indicates to all devices that it is using the bus by activating open collector line, Bus Busy (BBSY).

A simple arrangement for bus arbitration using a daisy chain

Sequence of signals during transfer of bus mastership for the devices. The timing diagram shows the sequence of events for the devices connected to the processor is shown. DMA controller 2 requests and acquires bus mastership and later releases the bus. During its tenture as bus master, it may perform one or more data transfer. After it releases the bus, the processor resources bus mastership.

Distributed Arbitration:

It means that all devices waiting to use the bus have equal responsibility in carrying out the arbitration process.

Each device on the bus is assigned a 4 bit id. When one or more devices request the bus, they assert the Start-Arbitration signal & place their 4 bit ID number on four open collector lines, ARB0 to ARB3. A winner is selected as a result of the interaction among the signals transmitted over these lines. The net outcome is that the code on the four lines represents the request that has the highest ID number. The drivers are of open collector type. Hence, if the i/p to one driver is equal to 1, the i/p to another driver connected to the same bus line is equal to 0 (ie) bus the is in low-voltage state Eg: Assume two devices A & B have their ID 5 (0101), 6(0110) and their code is 0111.

Each device compares the pattern on the arbitration line to its own ID starting from MSB. If it detects a difference at any bit position, it disables the drivers at that bit position. It does this by placing „0" at the i/p of these drivers. In our eg. A detects a difference in line ARB1, hence it disables the drivers on lines ARB1 & ARB0. This causes the pattern on the arbitration line to change to 0110 which means that B has won the contention.

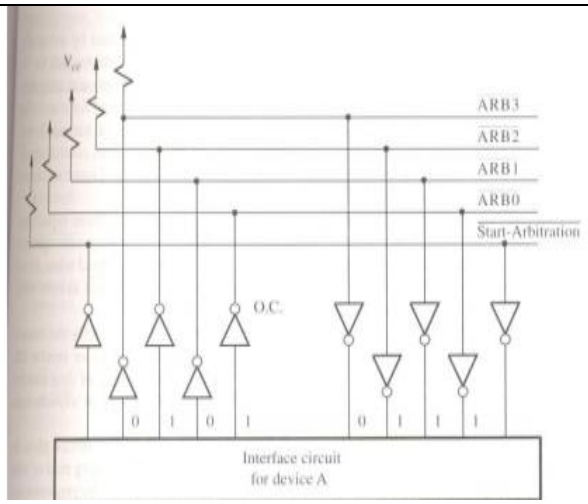
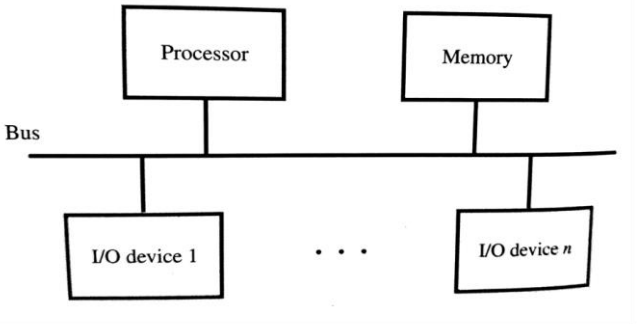


Fig: A distributed arbitration Scheme

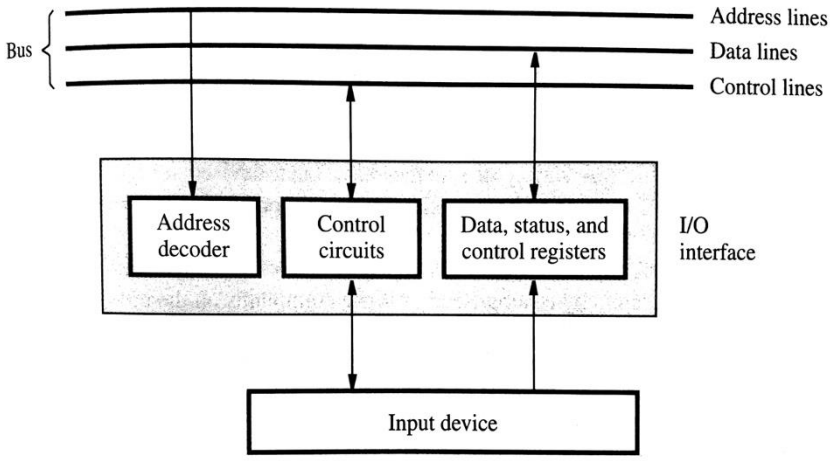
BUS STRUCTURE

The bus consists of three sets of lines used to carry address, data, and control signals. I/O device interfaces are connected to these lines

- Each I/O device is assigned a unique set of addresses for the registers in its interface.
- When the processor places a particular address on the address lines, it is examined by the address decoders of all devices on the bus



- The device that recognizes this address responds to the commands issued on the control lines.
- The processor uses the control lines to request either a Read or a Write operation, and the requested data are transferred over the data lines.

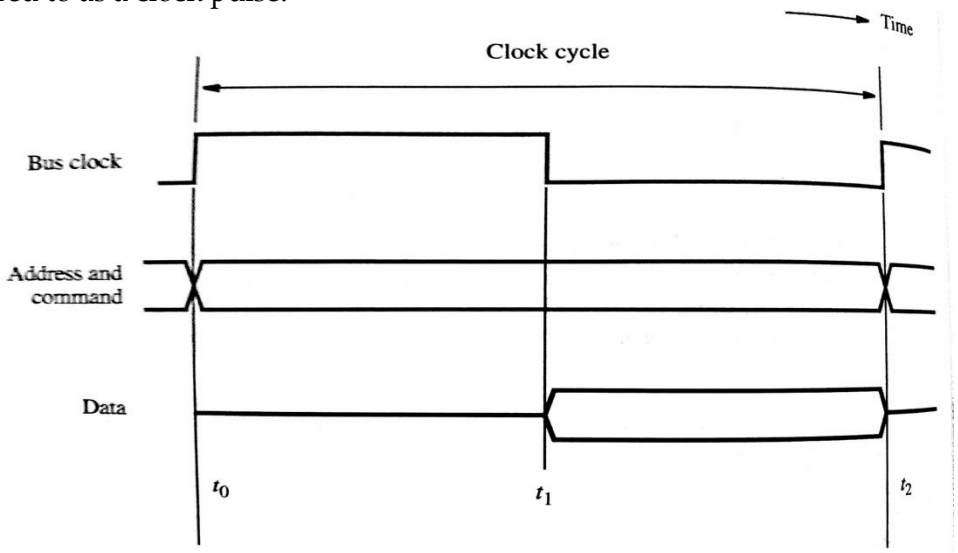


BUS OPEARTION

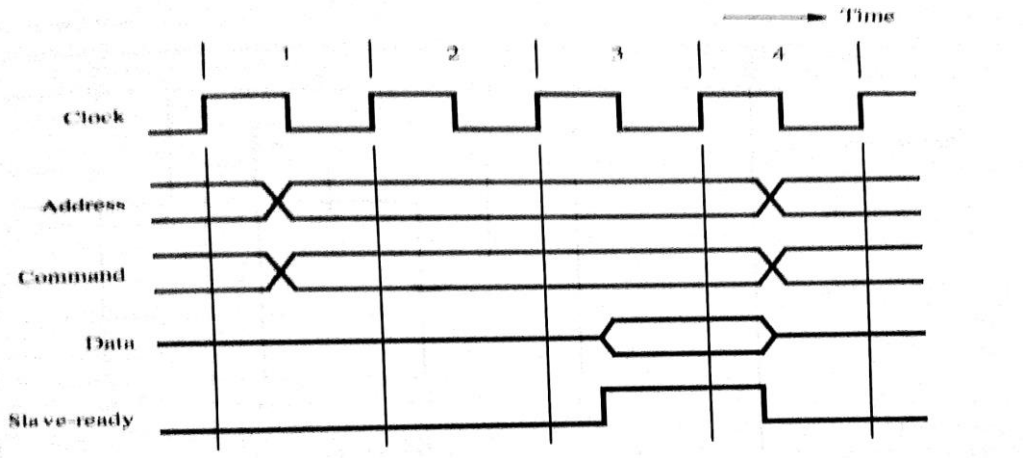
A bus requires a set of rules, often called a bus protocol. The bus protocol determines when a device may place information on the bus, when it may load the data on the bus into one of its registers These rules are implemented by control signals. One control line, usually labeled R/W, specifies whether a Read or a Write operation is to be performed. One device plays the role of a master. This is the device that initiates data transfers by issuing Read or Write commands on the bus. The device addressed by the master is referred to as a slave.

SYNCHRONOUS BUS:

On a synchronous bus, all devices derive timing information from a control line called the bus clock The signal on this line has two phases: a high level followed by a low level. The two phases constitute a clock cycle. The first half of the cycle between the low-to-high and high-to-low transitions is often referred to as a clock pulse.

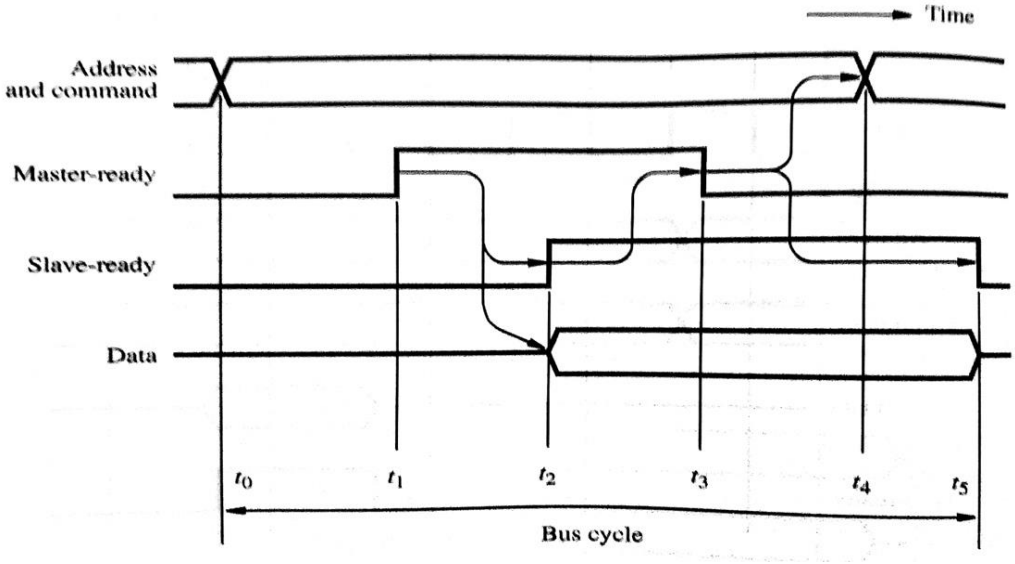


During clock cycle 1, the master sends address and command information on the bus, requesting a Read operation. The slave receives this information and decodes it. It begins to access the requested data on the active edge of the clock at the beginning of clock cycle 2. The data become ready and are placed on the bus during clock cycle 3. The slave asserts a control signal called Slave-ready at the same time. The master, which has been waiting for this signal, loads the data into its register at the end of the clock cycle. The slave removes its data signals from the bus and returns its Slave-ready signal to the low level at the end of cycle 3. The bus transfer operation is now complete, and the master may send new address and command signals to start a new transfer in clock cycle 4.



ASYNCHRONOUS BUS:

An alternative scheme for controlling data transfers on a bus is based on the use of a handshake protocol between the master and the slave. A handshake is an exchange of command and response signals between the master and the slave. A control line called Master-ready is asserted by the master to indicate that it is ready to start a data transfer. The Slave responds by asserting Slave-ready.



t0—The master places the address and command information on the bus, and all devices on the bus decode this information.

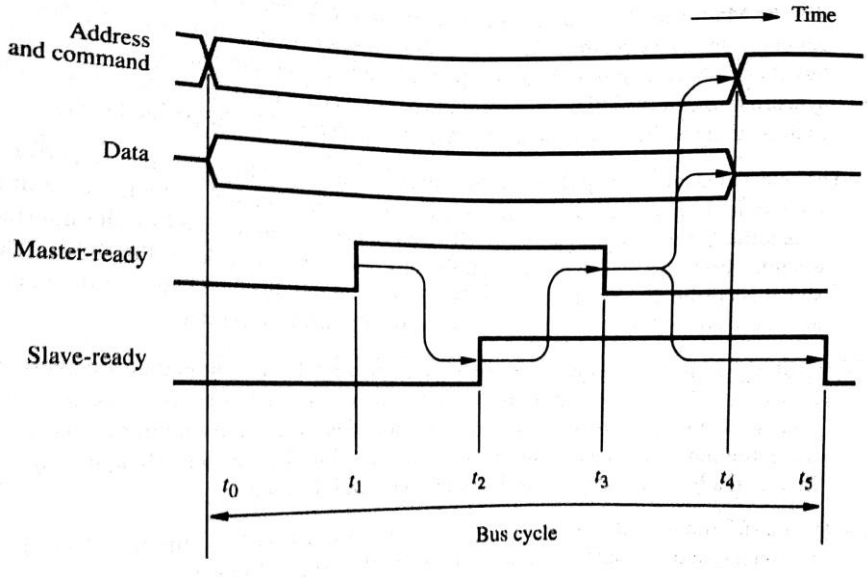
t1—The master sets the Master-ready line to 1 to inform the devices that the address and command information is ready. The delay t1 − t0 is intended to allow for any skew that may occur on the bus. Skew occurs when two signals transmitted simultaneously from one source arrive at the destination at different times.

t2—The selected slave, having decoded the address and command information, performs the required input operation by placing its data on the data lines. At the same time, it sets the Slave-ready signal to 1.

t3—The Slave-ready signal arrives at the master, indicating that the input data are available on the bus.

t4—The master removes the address and command information from the bus.

t5—When the device interface receives the 1-to-0 transition of the Master-ready signal, it removes the data and the Slave-ready signal from the bus. This completes the input transfer.



BUS ARBITRATION

It is the process by which the next device to become the bus master is selected and the bus mastership is transferred to it.

Types:

There are 2 approaches to bus arbitration. They are

i)Centralized arbitration (A single bus arbiter performs arbitration)

ii)Distributed arbitration (all devices participate in the selection of next bus¬ master).

Centralized Arbitration:

Here the processor is the bus master and it may grants bus mastership to one of its DMA controller. A DMA controller indicates that it needs to become the bus master by activating the Bus Request line (BR) which is an open drain line. The signal on BR is the logical OR of the bus request from all devices connected to it. When BR is activated the processor activates the Bus Grant Signal (BGI) and indicated the DMA controller that they may use the bus when it becomes free. This signal is connected to all devices using a daisy chain arrangement. If DMA requests the bus, it blocks the propagation of Grant Signal to other devices and it indicates to all devices that it is using the bus by activating open collector line, Bus Busy (BBSY).

A simple arrangement for bus arbitration using a daisy chain

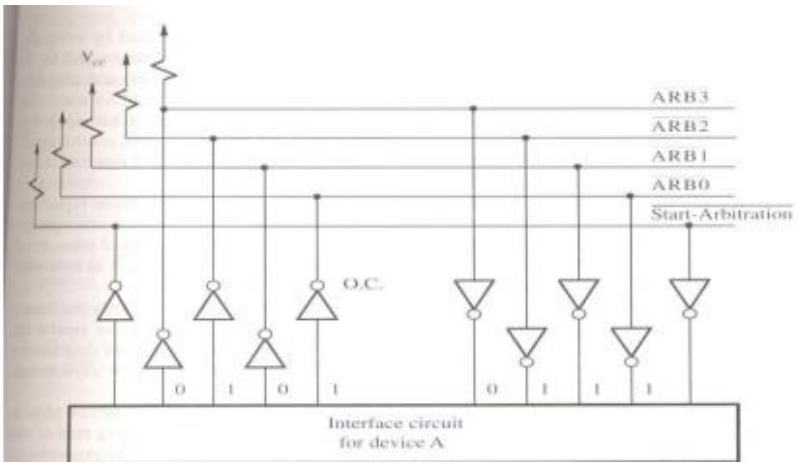
Sequence of signals during transfer of bus mastership for the devices

The timing diagram shows the sequence of events for the devices connected to the processor is shown. DMA controller 2 requests and acquires bus mastership and later releases the bus. During its tenture as bus master, it may perform one or more data transfer. After it releases the bus, the processor resources bus mastership.

Distributed Arbitration:

It means that all devices waiting to use the bus have equal responsibility in carrying out the arbitration process.

Fig: A distributed arbitration Scheme



Each device on the bus is assigned a 4 bit id. When one or more devices request the bus, they assert the Start-Arbitration signal & place their 4 bit ID number on four open collector lines, ARB0 to ARB3. A winner is selected as a result of the interaction among the signals transmitted over these lines. The net outcome is that the code on the four lines represents the request that has the highest ID number. The drivers are of open collector type. Hence, if the i/p to one driver is equal to 1, the i/p to another driver connected to the same bus line is equal to 0 (ie) bus the is in low-voltage state Eg: Assume two devices A & B have their ID 5 (0101), 6(0110) and their code is 0111.

Each device compares the pattern on the arbitration line to its own ID starting from MSB.

If it detects a difference at any bit position, it disables the drivers at that bit position. It does this by placing „0" at the i/p of these drivers. In our eg., A detects a difference in line ARB1, hence it disables the drivers on lines ARB1 & ARB0. This causes the pattern on the arbitration line to change to 0110 which means that B has won the contention.

INTERFACE CIRCUITS

The I/O interface of a device consists of the circuitry needed to connect that device to the bus. On one side of the interface are the bus lines for address, data, and control. On the other side are the connections needed to transfer data between the interface and the I/O device. This side is called a port, and it can be either a parallel or a serial port. A parallel port transfers multiple bits of data simultaneously to or from the device. A serial port sends and receives data one bit at a time. An I/O interface does the following:

1. Provides a register for temporary storage of data
2. Includes a status register containing status information that can be accessed by the processor
3. Includes a control register that holds the information governing the behavior of the interface
4. Contains address-decoding circuitry to determine when it is being addressed by the processor
5. Generates the required timing signals
6. Performs any format conversion that may be necessary to transfer data between the processor and the I/O device, such as parallel-to-serial conversion in the case of a serial port

PARALLEL INTERFACE

An interface circuit for an 8-bit input port that can be used for connecting a simple input device, such as a keyboard. An interface circuit for an 8-bit output port, which can be used with an output device such as a display. Interface circuits are connected to a 32-bit processor that uses memory-mapped I/O and the asynchronous bus protocol.

Input Interface

There are only two registers: a data register, KBD_DATA, and a status register, KBD_STATUS. The latter contains the keyboard status flag, KIN. A typical keyboard consists of mechanical switches that are normally open. When a key is pressed, its switch closes and establishes a path for an electrical signal. A difficulty with such mechanical pushbutton switches is that the contacts bounce when a key is pressed, resulting in the electrical connection being made then broken several times before the switch settles in the closed position. The software detects that a key has been pressed when it observes that the keyboard status flag, KIN, has been set to 1. The I/O routine can then introduce sufficient delay before reading the contents of the input buffer, KBD_DATA, to ensure that bouncing has subsided. When debouncing is implemented in hardware, the I/O routine can read the input character as soon as it detects that KIN is equal to 1. It consists of one byte of data representing the encoded character and one control signal called Valid. When a key is pressed, the Valid signal changes from 0 to 1. The status flag is cleared to 0 when the processor reads the contents of the KBD_DATA register. The interface circuit is connected to an asynchronous bus on which transfers are controlled by the handshake signals Master-ready and Slave-ready. The bus has one other control line, R/W, which indicates a Read operation when equal to 1.

Output Interface

It can be used to connect an output device such as a display. The display uses two handshake signals, New-data and Ready, in a manner similar to the handshake between the bus signals Master-ready and Slave-ready. When the display is ready to accept a character, it asserts its Ready signal, which causes the DOUT flag in the DISP_STATUS register to be set to 1. When the I/O routine checks DOUT and finds it equal to 1, it sends a character to DISP_DATA. This clears the DOUT flag to 0 and sets the New-data signal to 1. In response, the display returns Ready to 0 and accepts and displays the character in DISP_DATA

SERIAL INTERFACE

A serial interface is used to connect the processor to I/O devices that transmit data one bit at a time. Data are transferred in a bit-serial fashion on the device side and in a bit-parallel fashion on the processor side. The transformation between the parallel and serial formats is achieved with shift registers that have parallel access capability. The input shift register accepts bit-serial input from the I/O device. When all 8 bits of data have been received, the contents of this shift register are loaded in parallel into the

DATAIN register. Output data in the DATAOUT register are transferred to the output shift register, from which the bits are shifted out and sent to the I/O device. The part of the interface that deals with the bus is the same as in the parallel interface described earlier. Two status flags, which we will refer to as SIN and SOUT, are maintained by the Status and control block. The SIN flag is set to 1 when new data are loaded into DATAIN from the shift register, and cleared to 0 when these data are read by the processor. The SOUT flag indicates whether the DATAOUT register is available. It is cleared to 0 when the processor writes new data into DATAOUT and set to 1 when data are transferred from DATAOUT to the output shift register. The double buffering used in the input and output paths in Figure 7.15 is important. It is possible to implement DATAIN and DATAOUT themselves as shift registers, thus obviating the need for separate shift registers

USB

The Universal Serial Bus (USB) is the most widely used interconnection standard. A large variety of devices are available with a USB connector, including mice, memory keys, disk drives, printers, and cameras. The commercial success of the USB is due to its simplicity and low cost. The original USB specification supports two speeds of operation, called low-speed (1.5 Megabits/s) and full-speed (12 Megabits/s). It supports data transfer rates up to 5 Gigabits/s.

The USB has been designed to meet several key objectives:

- Provide a simple, low-cost, and easy to use interconnection system
- Accommodate a wide range of I/O devices and bit rates, including Internet connections, and audio and video applications
- Enhance user convenience through a "plug-and-play" mode of operation

DEVICE CHARACTERISTICS:

The kinds of devices that may be connected to a computer cover a wide range of functionality. The speed, volume, and timing constraints associated with data transfers to and from these devices vary significantly. One byte of data is generated every time a key is pressed, which may happen at any time.

These data should be transferred to the computer promptly. The event of pressing a key is not synchronized to any other event in a computer system, the data generated by the keyboard are called asynchronous. The sampling process yields a continuous stream of digitized samples that arrive at regular intervals, synchronized with the sampling clock. Such a data stream is called isochronous, meaning that successive events are separated by equal periods of time. A signal must be sampled quickly enough to track its highest-frequency components.

Plug-and-Play

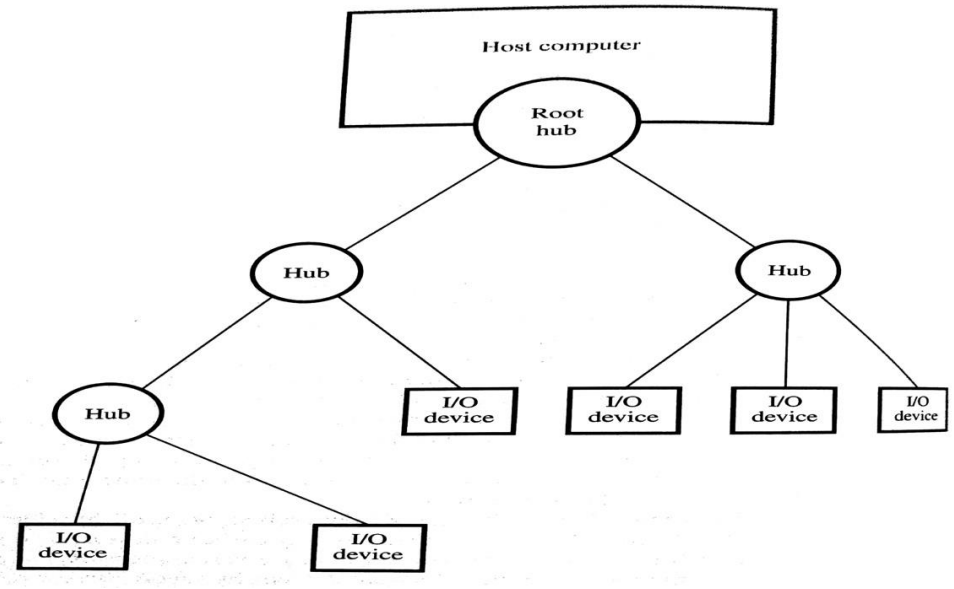
When an I/O device is connected to a computer, the operating system needs some information about it. It needs to know what type of device it is so that it can use the appropriate device driver.

It also needs to know the addresses of the registers in the device's interface to be able to communicate with it. The USB standard defines both the USB hardware and the software that communicates with it. Its plug-and-play feature means that when a new device is connected, the system detects its existence automatically.

USB Architecture

The USB uses point-to-point connections and a serial transmission format. When multiple devices are connected, they are arranged in a tree structure. Each node of the tree has a device called a hub, which acts as an intermediate transfer point between the host computer and the I/O devices. At the root of the tree, a root hub connects the entire tree to the host computer. The leaves of the tree are the I/O devices: a mouse, a keyboard, a printer, an Internet connection, a camera, or a speaker. The tree structure

makes it possible to connect many devices using simple point-to-point serial links.



The USB uses point-to-point connections and a serial transmission format. When multiple devices are connected, they are arranged in a tree structure. Each node of the tree has a device called a hub, which acts as an intermediate transfer point between the host computer and the I/O devices. At the root of the tree, a root hub connects the entire tree to the host computer. The leaves of the tree are the I/O devices: a mouse, a keyboard, a printer, an Internet connection, a camera, or a speaker. The tree structure makes it possible to connect many devices using simple point-to-point serial links.

Electrical Characteristics

USB connections consist of four wires, of which two carry power, +5 V and Ground, and two carry data. I/O devices that do not have large power requirements can be powered directly from the USB. Two methods are used to send data over a USB cable. When sending data at low speed, a high voltage relative to Ground is transmitted on one of the two data wires to represent a 0 and on the other to represent a 1. The Ground wire carries the return current in both cases. Such a scheme in which a signal is injected on a wire relative to ground is referred to as *single-ended* transmission. The speed at which data can be sent on any cable is limited by the amount of electrical noise present. The term noise refers to any signal that interferes with the desired data signal and hence could cause errors. Single-ended transmission is highly susceptible to noise. The voltage on the ground wire is common to all the devices connected to the computer. Signals sent by one device can cause small variations in the voltage on the ground wire, and hence can interfere with signals sent by another device. Interference can also be caused by one wire picking up noise from nearby wires. The High-Speed USB uses an alternative arrangement known as *differential signaling*. The data signal is injected between two data wires twisted together.