How can we define computer architecture?

Old defn: Computer Architecture ≈ Instruction set design.
(flawed, rejected)

Implementation of a computer

Two Components

Organisation (or Microarchitecture)

Hardware

Organisation includes the high level aspects of a computer design e.g. the memory system, the memory interconnection and the design of the internal processor or CPU.

Eg: AMD opteron & Intel core i7 } Same ISA (x86) but different pipeline and cache organisation.

Term "CPU" — gradually, this term is fading away.

New term - 'multicore' (implying multiprocessor microprocessor)

② Hardware: refers to the specifics of a computer, including the detailed logic design and the package technology of the computer.

New Definition: Architecture covers all three aspects of computer design: Instruction set Architecture, organisation or microarchitecture, and hardware.

Computer Architecture has to consider:

① Functional Requirements ② Price ③ Performance ④ Availability.

Some of the most important functional requirements:

1) Application area (Mobile / Desktop / Server / Clusters / Embedded)
2) Level of software compatibility (determines the amount of existing software for the computer).
3) Operating System Requirement (e.g. size of address space, memory management)
4) Standards (e.g. floating point (IEEE 754 standard), I/O interfaces (e.g. serial ATA, serial attached SCSI)), operating system (Windows/Linux), networks (ethernet), programming languages)

Trends in Technology

To plan for the ~~evolution~~ evolution of computer, the designer must be aware of rapid changes in implementation technology.

Five Implementation ~~techniques~~ (technologies), which change at dramatic pace, are critical to modern implementations.

1) Integrated Circuit Tech: Transistor density increases by about 35% per year.
2) Semiconductor DRAM - Capacity per DRAM chip doubled every 2 to 3 years around 2011. This growth rate may not continue due to the increasing difficulty of manufacturing even smaller DRAM cells.
3) Semiconductor Flash (EEPROM) - Capacity doubles roughly every two years. It is 15-20 times cheaper than DRAM.
4) Magnetic Disk Technology: Capacity doubles every 3 years. Disks are 15-20 times cheaper than flash. Disks are 300-500 times cheaper than DRAM. Disk technology is

central to server and warehouse scale technology.

(v) Network Technology - Performance depends on performance of switches and on the performance of transmission systems.

The designer must plan further changes.

Trends in Power and Energy in Integrated Circuits

Power is the biggest challenge facing the computer designer.

(i) Power must be brought in and distributed around the chip

(ii) Power is dissipated as heat and must be removed.

Power and Energy – A Systems Perspective

There are 3 primary concerns:

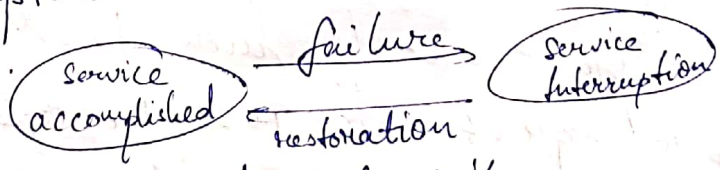
(i) What is the maximum power a processor ever requires?
If a processor attempts to draw more power than a power supply system can provide, the result is typically a voltage drop which can cause the device to malfunction.

(ii) What is sustained power consumption?
This metric is widely called the Thermal Power Design (TDP) since it determines the cooling requirement.

(iii) Energy and energy efficiency – If we want to know which of the two processes is more efficient for a given task, we should compare energy consumption (not power) for executing the task.

Dependability:

The system alternates between two states:



MTTF → Mean time to failure

MTTR → Mean time to Repair

MTBF → Mean time between failures.

= MTTF + MTTR

$$\text{Module Availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$

Quantitative Principles of Computer Design

(i) Take advantage of parallelism

- One of the most important methods for improving performance
- Parallelism at system level → multiple processors and multiple disks in a server
- Parallelism at the level of an individual processor - pipelining

Key Insight Instructions often do not depend on its immediate predecessors.

- Parallelism at the level of detailed digital design:

Ex: (a) Set associative caches use multiple banks of memory that are typically searched in parallel to find a desired item.

(b) Modern ALUs use carry look-ahead.

(ii) Principle of locality: ~~Programs~~ Programs tend to reuse data and instructions they have used recently.

Rule of thumb: A program spends 90% of its execution time in only 10% of the code

Implication: We can predict with reasonable accuracy what instructions and data a program will use in the near future based on its accesses in the recent past.

(iii) Focus on the common case (most imp. and pervassive principle of computer design):

In making a design trade-off, favour the frequent case over the infrequent case. This principle applies when determining how to spend resources since the impact of the improvement is higher if the occurence is frequent.

AMDAHL'S Law can be used to quantify this principle:

The performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.

It defines the speedup that can be gained by using a particular feature.

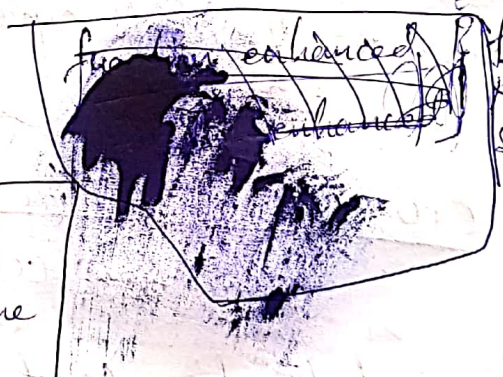
SPEED UP

Suppose that we can make an enhancement to a computer that will improve performance when it is used. Then we define:

$$\text{Speed Up} = \frac{\text{Performance for entire task using the enhancement when possible}}{\text{Performance for entire task WITHOUT using the enhancement.}}$$

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

$$\text{Execution time}_{new} = \text{Execution time}_{old} \times \left\{ \left(1 - \text{fraction}_{enhanced}\right) + \frac{\text{fraction enhanced}}{\text{speedup enhanced}} \right\}$$



The overall speedup is the ratio of the execution times.

$$\text{Speedup}_{\text{overall}} = \frac{\text{execution time old}}{\text{execution time new}} = \frac{1}{\left(1 - \text{fraction}_{enhanced}\right) + \frac{\text{fraction enhanced}}{\text{fraction speedup enhanced}}}$$

Example: Suppose we want to enhance the processor.
New processor is 10 times faster.
Computation time = 40%
I/O time = 60%

$$\text{Fraction}_{\text{enhanced}} = 0.4$$

$$\text{Speedup}_{\text{enhanced}} = 10$$

$$\text{Speed}_{\text{overall}} = \frac{1}{0.6 + \frac{0.4}{10}} = \frac{5}{3.2} \simeq 1.56$$

The processor performance ~~equal~~ equation

CPU time for a program is expressed in two ways.

CPU time = CPU clock cycles for a program × Clock cycle time

OR

$$= \frac{\text{CPU Clock cycles for a prog.}}{\text{Clock rate}}$$

Defn: IC (instruction count or instruction path length count)
→ It is the no. of instructions executed.

CPI → Average no. of clock cycles per instruction.

$$\text{CPI} = \frac{\text{CPU clock cycles for a program}}{\text{Instruction count}}$$

CPU time = Instruction count × cycles per instruction × clock cycle time.

Considering units of measurement:

$$\text{CPU time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock Cycle}} = \frac{\text{Seconds}}{\text{Program}}$$

Classes of parallelism and Parallel Architecture:

There are basically 2 levels of parallelism in applications:

1) Data level parallelism (DLP) arises because there are many data items that can be operated on at the same time.

2) Task level parallelism (TLP): arises because tasks of work are created that can operate independently and largely in parallel.

Computer hardware in turn can ~~operate~~ exploit these two levels of application parallelism in four major ways.

(i) Instruction level parallelism: Exploits data-level parallelism at modest levels with compiler help using ideas like pipelining.

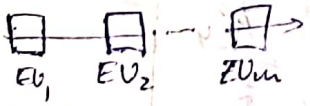
(ii) Vector architectures and Graphic Processing Units (GPUs): exploit data level parallelism by applying a single instruction to a collection of data in ||el.

(iii) Thread Level Parallelism: Exploits either data level parallelism or task level parallelism in a tightly coupled hardware model that allows for interaction among parallel threads.

(iv) Request level Parallelism: exploits parallelism among largely decoupled tasks specified by the programmer or the OS.

Internal Principle of ILP Processes

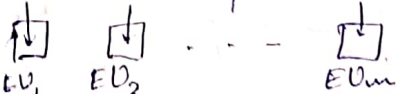
Pipelined Operation



Pipelined processor

Pipelined processor works like an assembler.

Parallel Operation.



VLIW & Superscaler Processor

(very long instruction word)

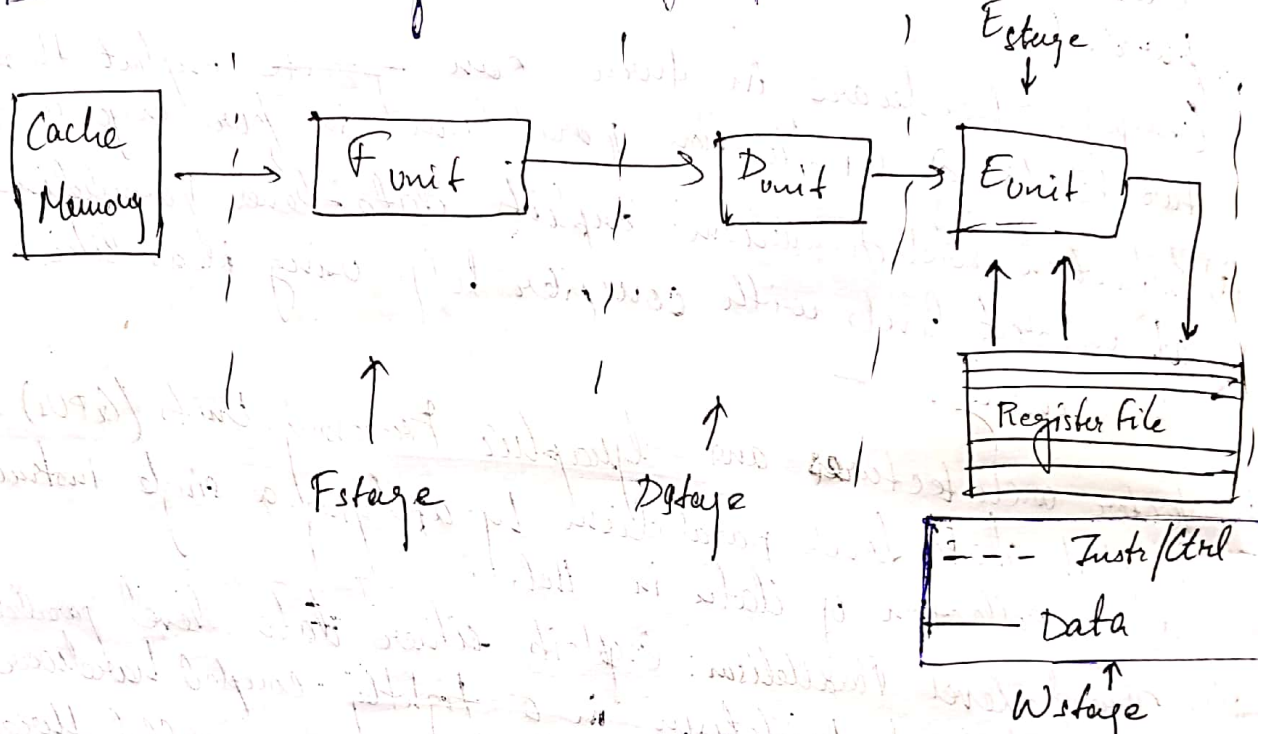
VLIW & Superscaler processors operate basically in parallel, making use of a number of concurrently working EUs.

Principle of Operation of pipelined processors

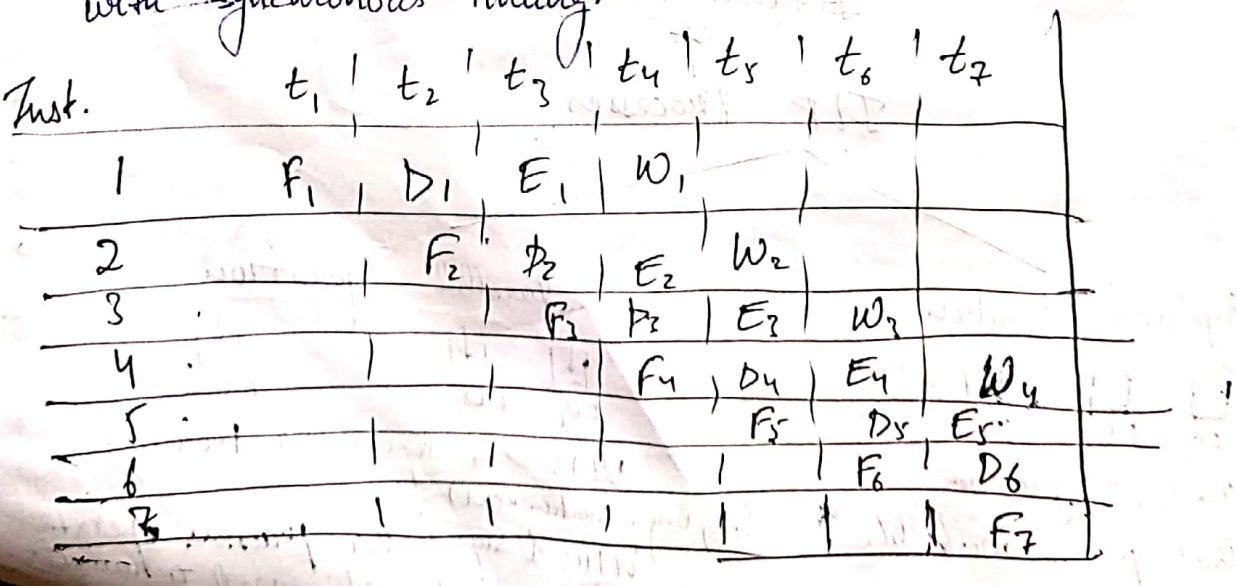
Fetch	Decode	Execution	Write

Subtasks of instruction execution for integer register-register operation.

Basic structure of an integer pipeline.



- Instruction processing is subdivided into a number of successive subtasks
- Each subtask is performed by an associated pipeline stage.
- All the pipeline stages operate like an assembly line, with synchronous timing.



A significant feature of pipelined execution:

→ in each cycle a new instruction can enter the pipeline

Dependencies between instructions

(i) Data dependency
(ii) Control dependency
(iii) Resource dependency

(i) DATA DEPENDENCY

RAW Dependencies (Read After Write)

i1: load r1, a
i2: add r2, r1, r1

i2 is RAW dependent on i1.

(If add executes faster r1 store previous value, so, r2 store wrong results)

[i2 uses r1 as a source, i1 loads r1]

WAR dependencies (Write After Read)

i1: mul r1, r2, r3.
add r2, r4, r5.

(If add executes faster mul (r1) stores wrong result.)

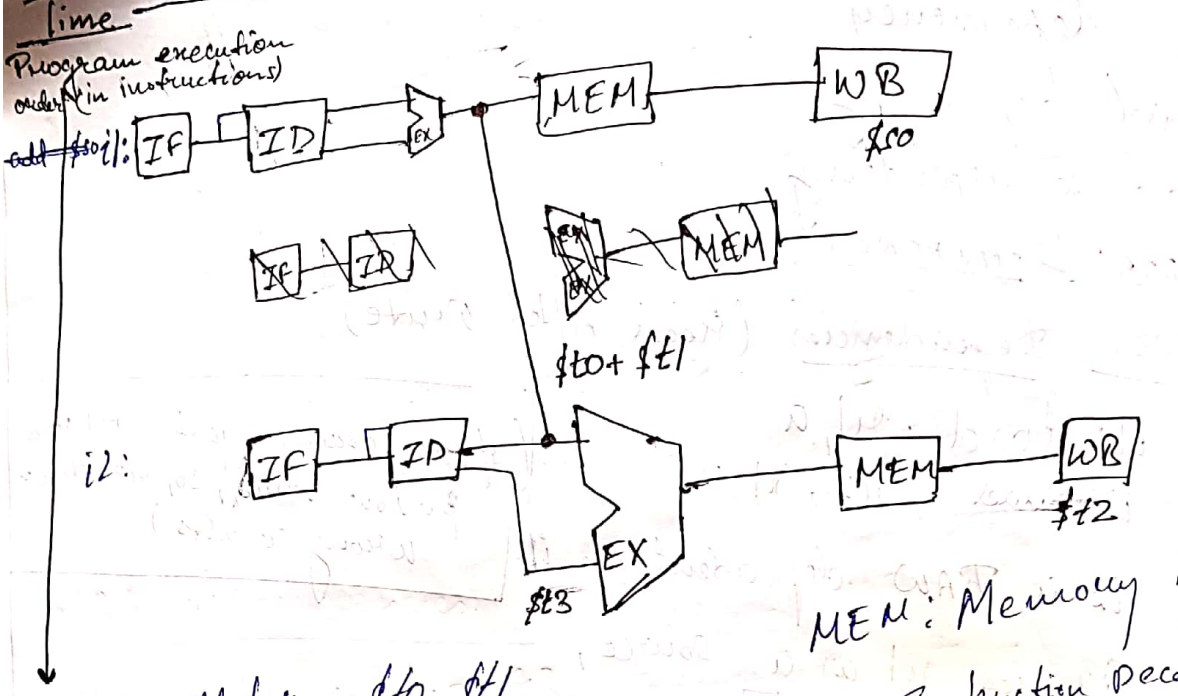
If i2 executes before i1, the original content of r2 would be rewritten earlier than it is read by i1.

Solution to Data Hazards

add $s0, $t0, $t1 (RAW hazard)
subs $t2, $s0, $t3

Soln: We don't need to wait for the instruction to complete before trying to ~~execute~~ resolve the data hazard. As soon as the ALU creates the sum for the add, we can supply it as input for the subtract. Adding extra hardware to retrieve the missing item early from the internal resource is called forwarding or bypassing.

Graphical representation of forwarding



i1: add $s0, $t0, $t1

i2: sub $t2, $s0, $t3

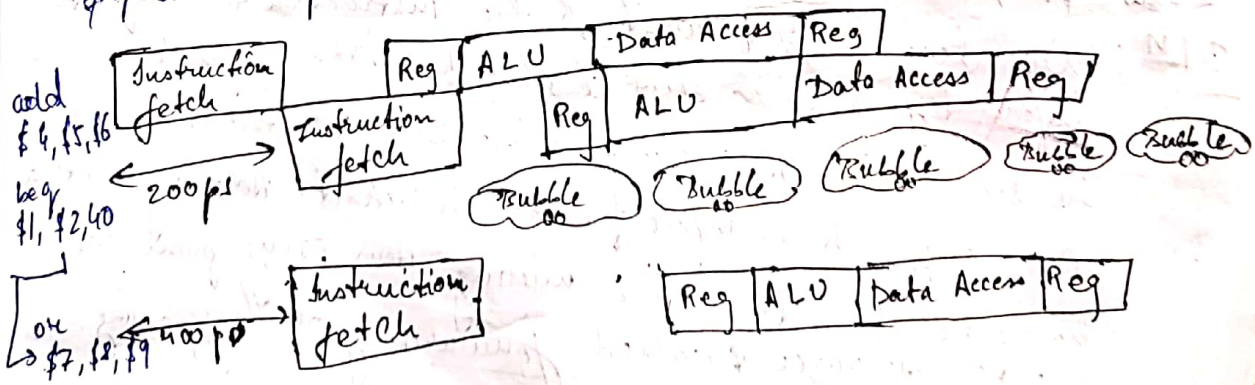
MEM: Memory Access stage

ID: Instruction Decode / Register File Read

The connection shows the forwarding path from the output of the EX stage of add to the input of EX stage for sub, replacing the value from register $s0 read in the second stage of sub.

Control hazard (also called branch hazard):→

An occurrence in which the proper instruction cannot execute in the proper clock cycle because the instruction that was fetched is NOT the one that is needed; i.e, the flow of instruction addresses is not what the pipeline expected



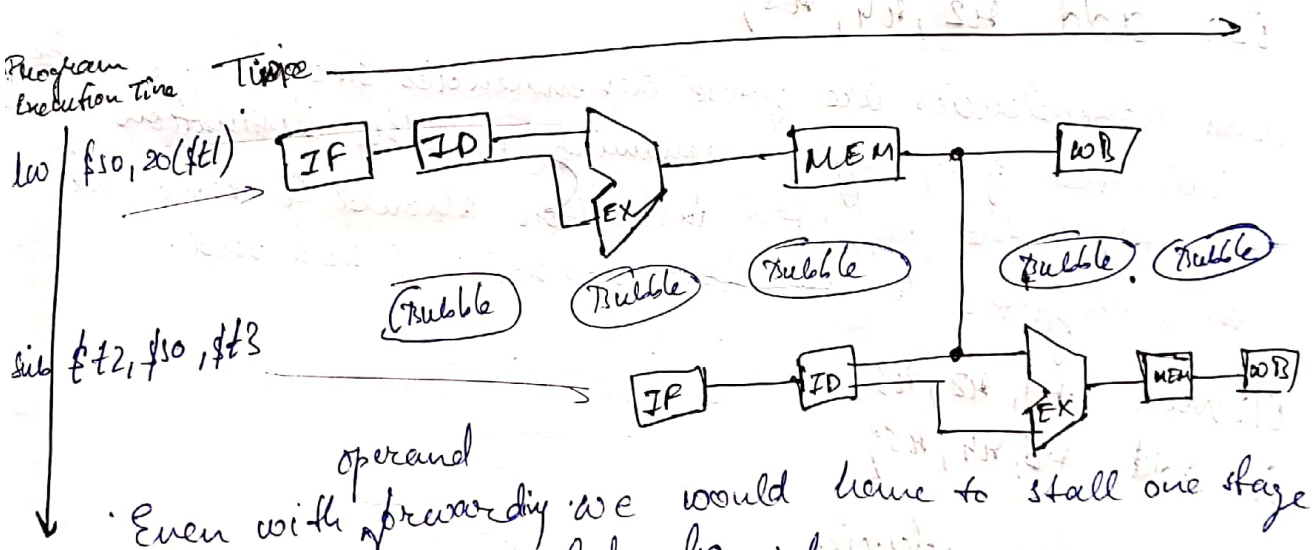
Pipeline showing stalling on every conditional branch as a solution to control hazard.

There is a one-stage STALL or BUBBLE after the branch.

Assumption: We can test registers, calculate the branch address, and update the PC during the SECOND stage.

Load-Use data hazard → A specific form of data hazard in which the data requested by a load instruction has NOT yet become available when it is requested.

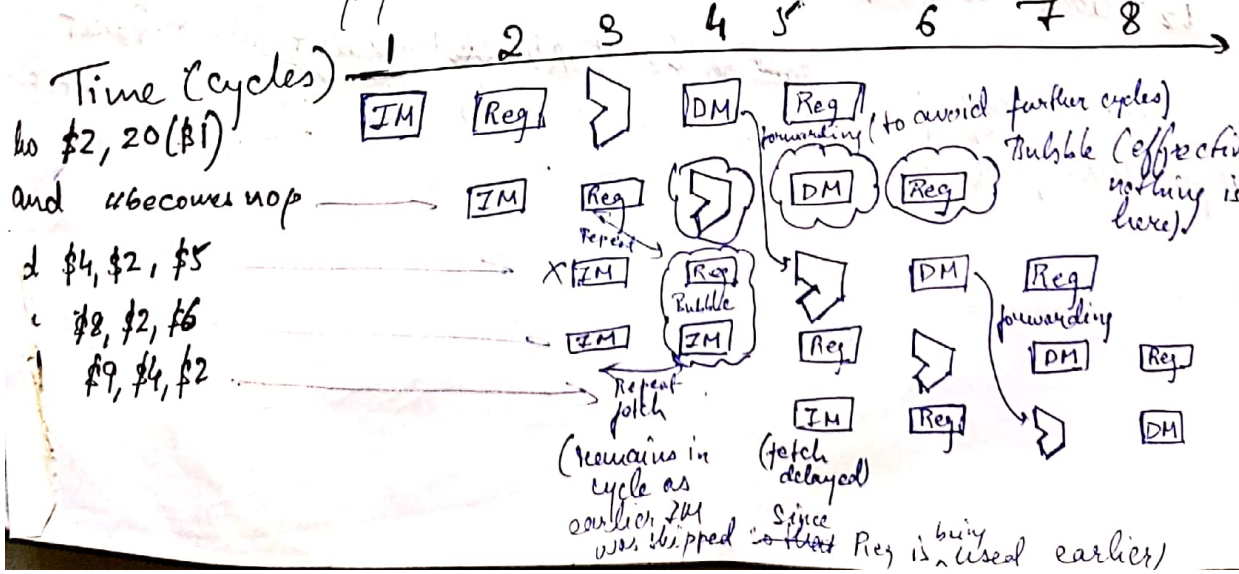
```
lw  $s0, 20($t1)
sub $t2, $s0, $t3
```



operand

Even with forwarding we would have to stall one stage for a load-use data hazard.

Data hazards: Low stalls are actually inserted into the pipeline.



A bubble is inserted, beginning in clock cycle 4, by changing the and instruction into a nop.

Note that the and instruction is really fetched and decoded in clock cycles 2 and 3, but its EX stage is delayed until clock cycle 5. (would have been done if in cycle 4 if stall was absent)

The hazard forces the and and or instructions to repeat in clock cycles 4 what they did in clock cycle 3.

WAR dependencies.

i1: mul r1, r2, r3;

i2: add r2, r4, r5;

WAR dependencies are false dependencies since they can be eliminated by register renaming i.e, the destination register of the affected instruction should be renamed to a register name that has not yet been used.

i1: mul r1, r2, r3;

i2: add r6, r4, r5;

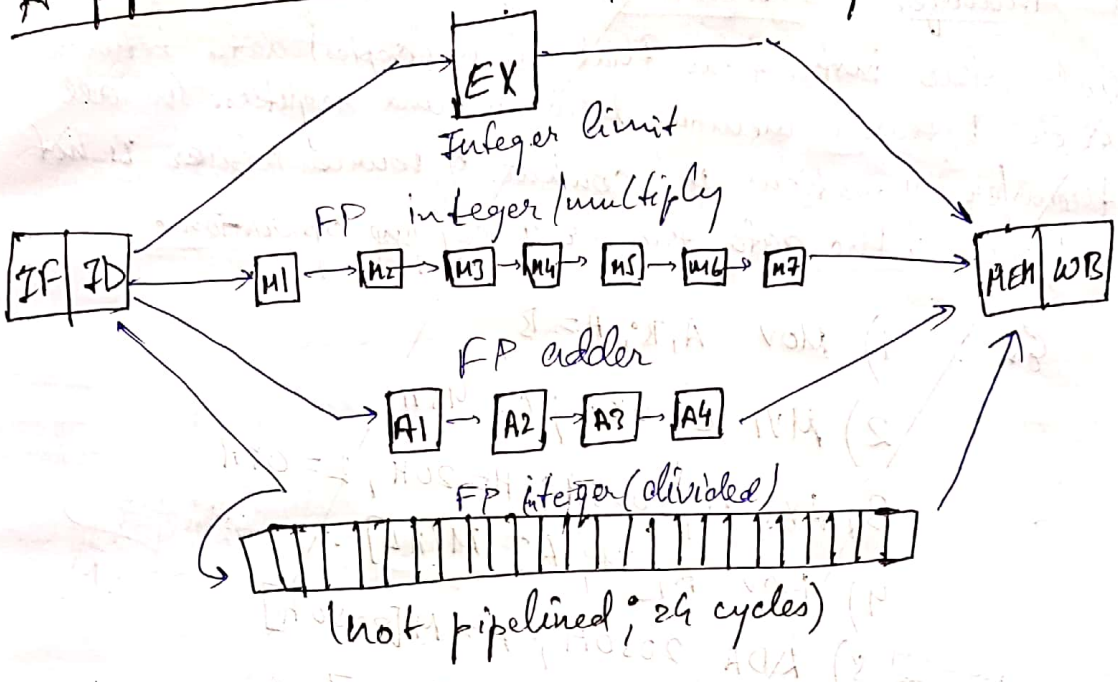
WAW dependencies

i1: mul r1, r2, r3;

i2: add r1, r4, r5;

Two instructions are said to be WAW dependent (or output dependent) if they both write into the same destination.

A pipeline with floating point operation:



Latencies and initiation intervals

Functional Unit	Latency	Initiation Interval
Integer ALU	0	1
Data memory (integer and FP loads)	1	1
FP add	3	1
FP multiply (also integer multiply)	6	1
FP divide	24	25

Latency → the number of intervening cycles between an instruction that produces a result and an instruction that uses the result.

Initiation interval: The number of cycles that must elapse between issuing two operations of a given type.

1. Data Transfer Instructions

It includes the instructions that move (copies) data between registers or between memory locations and registers. In all data transfer operations the content of source register is not altered. Hence the data transfer is copying operation.

Eg:
1) MOV A, B; A = B
2) MVI C, 45H; C = 45H
3) LXI H, 2005H; H = 20H, L = 05H
4) MOV A, M; A = M[HL]
5) LDA 2050H; A = M[2050H]
6) STA 2010H; M[2010H] = A

Only instructions, Direct memory address is specified

Pipeline Scheduling

Consider the loop:

```
for (i = 1000; i > 0; i = i - 1)
    x[i] = x[i] + s;
```

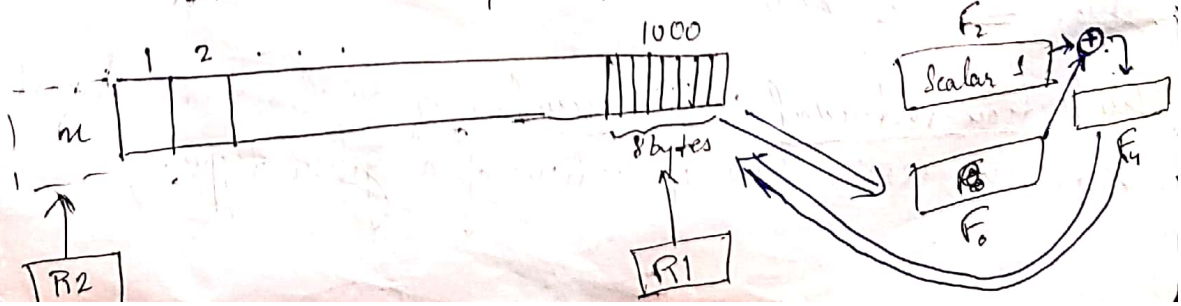
Word = 32 bits
FP (Floating Point)
32-bit single precision

Straight forward translation

```
Loop: L.D    F0, 0(R1)     ; F0 = array element
      ADD D  F4, F0, F2    ; add scalar in F2
      S.D    F4, 0(R1)     ; store result
      DADDUI R1, R1, #-8   ; decrement pointer
                           ; 8 bytes (per DW)
      BNE    R1, R2, Loop  ; branch R1 != R2
```

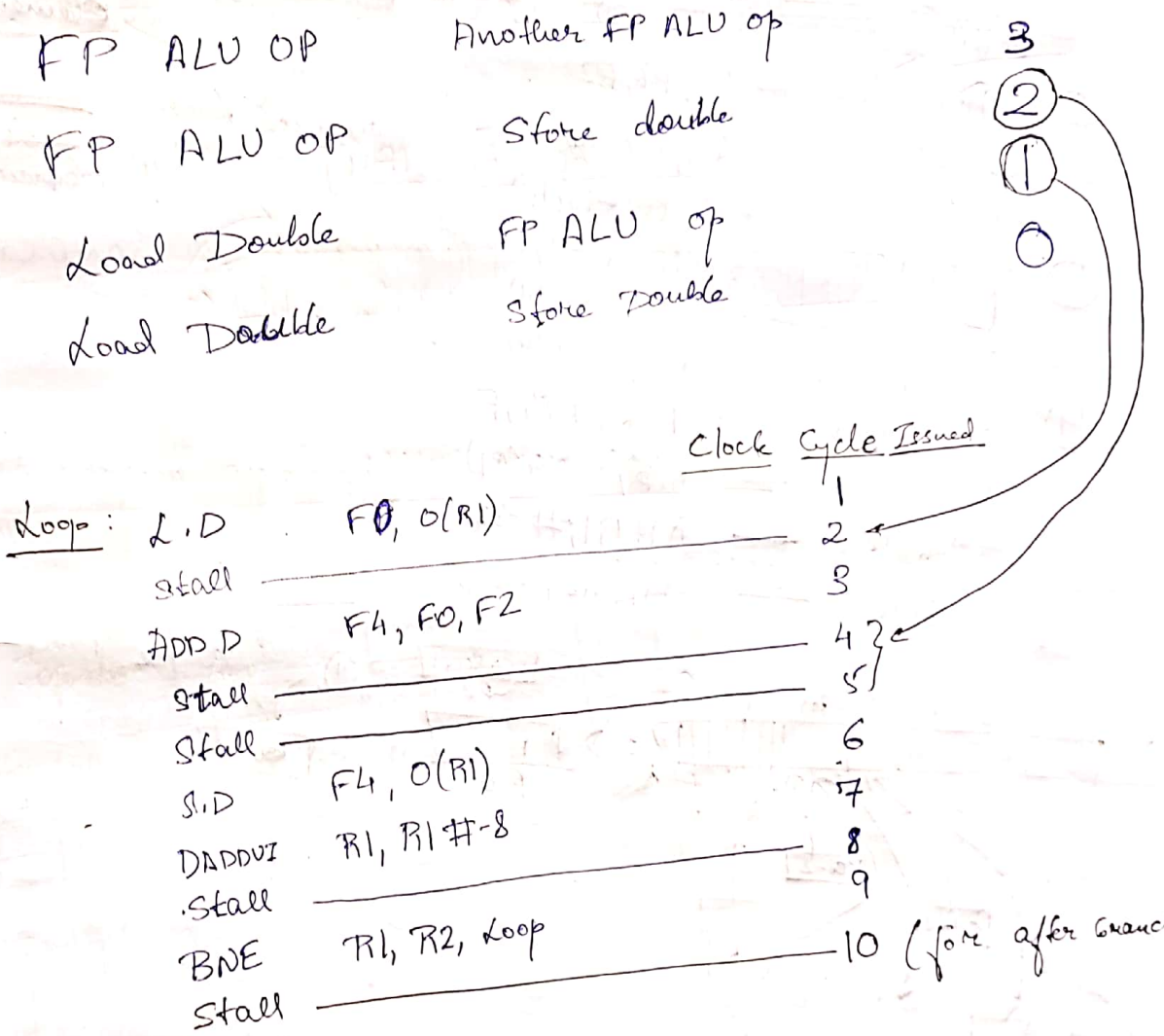
Add Immediate

Dependency



Latencies of FP Operations

Instruction Producing Result	Instruction Using Result	Latency in Clock Cycles
FP ALU OP	Another FP ALU op	3
FP ALU OP	Store double	2
Load Double	FP ALU op	1
Load Double	Store Double	0



WITH SCHEDULING

```
Loop: L.D     F0, 0(R1)
      DADDUI  R1, R1, #-8      (F0 is not affected so stall avoided)
      ADD D   F4, F0, F2
      Stall
      BNE     R1, R2, Loop     ; delayed branch
      S.D     F4, 8(R1)        ; Altered and interchanged with DADDUI
```

2 cycle Latency

To sum up:

With scheduling ⟶ 6 cycles per iteration (only ONE stall!)

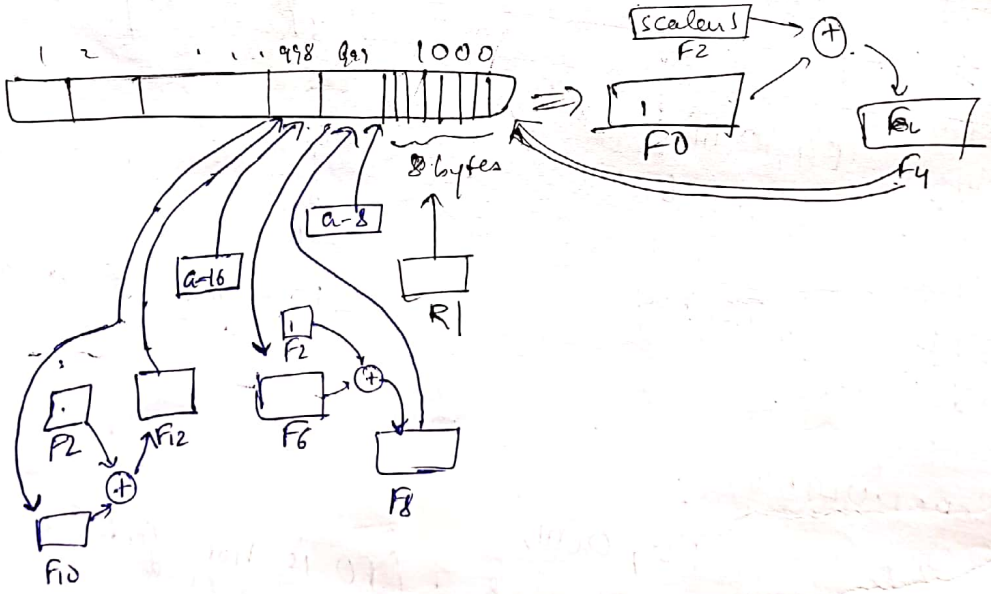
Without " ⟶ 10 cycles per iteration (5 stalls!!)

Loop Unrolling

```
Loop:  ① { L.D    ——  F0, 0(R1)      ← stall
           ADD.D  ——  F4, F0, F2     ← 2 stalls
           S.D    ——  F4, 0(R1)
       ② { L.D    ——  F6, -8(R1)
           ADD D  ——  F8, F6, F2
           S.D    ——  F8, -8(R1)
       ③ { L.D    ——  F10, -16(R1)
           ADD D  ——  F12, F10, F2
           S.D    ——  F12, -16(R1)
       ④ { L.D    ——  F14, -24(R1)
           ADD.D  ——  F16, F14, F2
           S.D    ——  F16, -24(R1)
           DADDUI ——  R1, R1, #-32
           BNE    ——  R1, R2, Loop.
```

14 issue cycles
+14 stalls
= 28 cycles per iteration
= 7 cycles per element

Original Loop: 6 cycles per element.



Scheduling the unrolled loop

```
Loop:  L.D    ——  F0, 0(R1)
       L.D    ——  F6, -8(R1)
       L.D    ——  F10, -16(R1)
       L.D    ——  F14, -24(R1)
       ADD D  ——  F4, F0, F2
       ADD D  ——  F8, F6, F2
       ADD D  ——  F12, F10, F2
       ADD D  ——  F16, F14, F2
       S.D    ——  F4, 0(R1)
       S.D    ——  F8, -8(R1)
```

cycles
s
Cper iteration
s per element
oop: 6 cycles per element

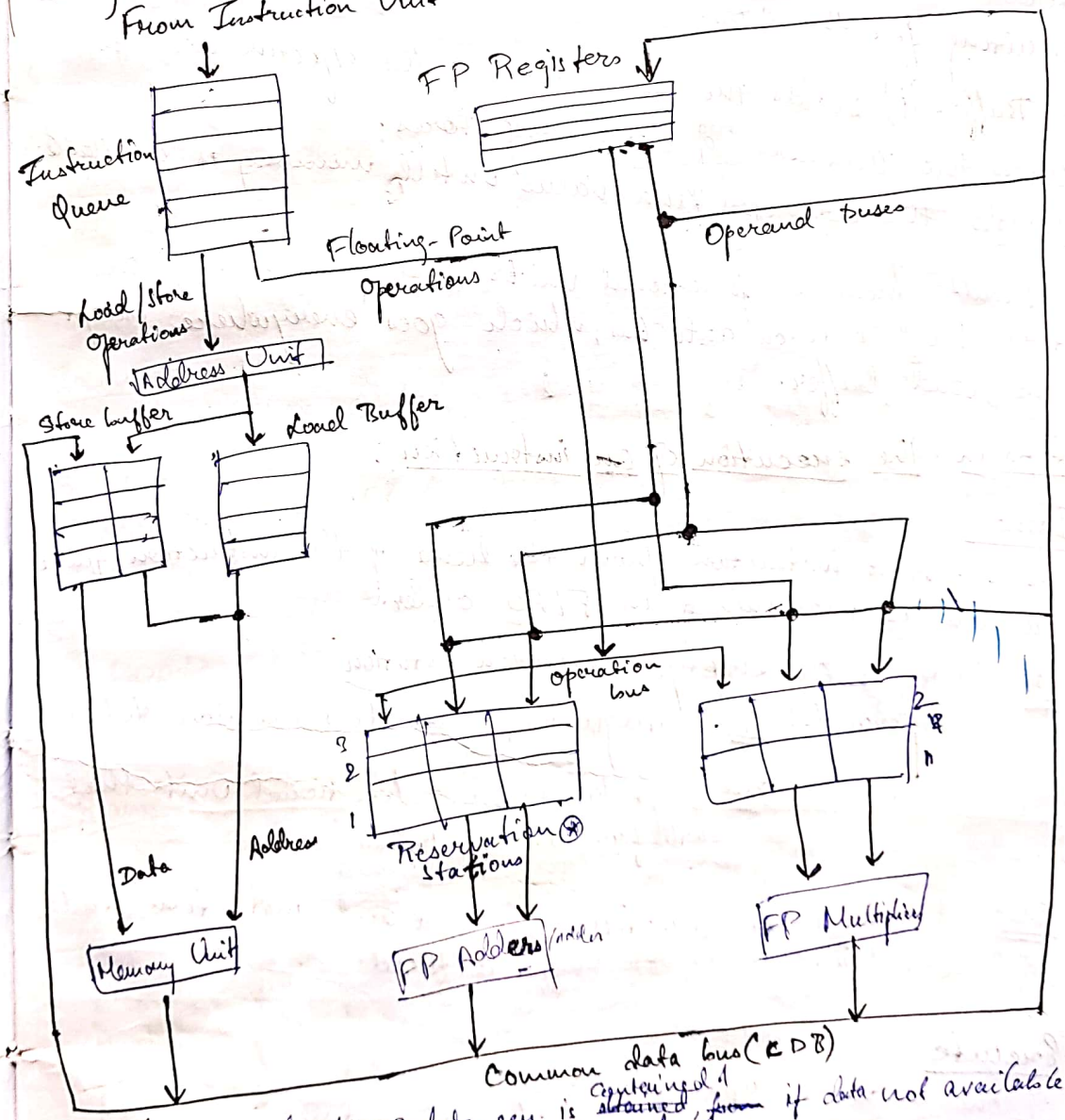
DADDUI	R1, R1, #-32
S.D	F12, -16(R1)
BNE	R1, R2, Loop
S.D	F4, 8(R1) ; 8-32 = -24

Execution Time

= 14 cycles per iteration

i.e. 14/4 = 3.5 cycles per element of the array.

Dynamic Hardware Scheduling (MIPS floating Point Unit)



* Address location of data gen. is contained / stored, from if data not available used.

Each reservation station holds the ~~inform~~ instruction that has been issued and is awaiting at a functional unit. It also contains the operand values for the instruction(if they have already been completed) or the names of the reservation station that will provide the operand values.

Load Buffers (i) holds the components of the effective address until it is computed.
(ii) Track outstanding loads that are waiting in the memory.
(iii) holds the results of the completed loads that are waiting for the CDB.

Store Buffers (i) holds the components of the effective address
(ii) holds the addresses of outstanding stores.
(iii) holds the addresses and value until memory is available.

All results from the functional units and from memory are sent on the common data bus, which goes everywhere except to the load buffer.

Steps in the execution of an instruction:

1. Issue

Get the next instruction from the head of the instruction queue, which is maintained in FIFO order.

If there is an empty reservation station
Operands: If in registers, issue to reservation station.
Else keeps the track of functional units that will produce the operands.

Else there is a structural hazard and instruction stalls until a station or buffer is freed.

2. Execute

If one or more operands is not available.
Monitor the common data bus while waiting for it to be computed. When an operand becomes available. If it is placed into the corresponding reservation station.

else (* all operands available *)

Execute operation of the corresponding functional unit.

LOADS & STORES require a TWO-STEP execution process.

If first step computes the effective address when the base register is available, and the effective address is then placed in the load or store buffers. LOADS in the load buffer execute as soon as the memory unit is available.

STORES in the store buffer, wait for the value to ~~be~~ be stored before being sent to the memory unit.

3. Write Result: When the result is available, write it on the CDB and from there into the registers and into any reservation stations (including store buffers) waiting for this result. STORES also write data to memory during this step: When both the address and data are available, they are sent to the memory unit and the store completes.

Example:

1. L.D	F6, 34(R2)	→ Completed; result on CDB
2. L.D	F2, 45(R3)	} Issued
3. MUL.D	F0, F2, F4	
4. SUB.D	F8, F2, F6	
5. DIV.D	F10, F0, F6	
6. ADD.D	F6, F8, F2	

Reservation Stations

Name	Busy	Op	Vj	Vk	Qj	Qk	A
							45+Reg[R3]
Load1	no						
Load2	yes	Load		Load1 → F6 Mem[34+Reg[R2]]			
Add1	yes	SUB			F2 Load1		F2 Load2
Add3	no				F8 Add1		
Mult1	yes	MUL		Reg[F4]	Load2 (F2)		
Mult2	yes	DIV		Load→F6 Mem[34+Reg[R2]]	Mult1	F0	

Register Status							...	
Field	F0	F2	F4	F6	F8	F10	F12 ...	F30
Q_i	Mult1	Load2		Add2	Add1	Mult2		

~~Q_i mult1 load2~~

Each reservation station has 7 fields:

V_j, V_k — The value of the source operands.

Q_j, Q_k — The reservation stations that will produce the source operands.

A — Used to hold information for the memory address calculation, for a load or store. Initially the immediate field of the instruction is stored here; after the address calculation, the effective address is stored here.

Busy — Indicates that this reservation station and its accompanying functional unit are occupied.

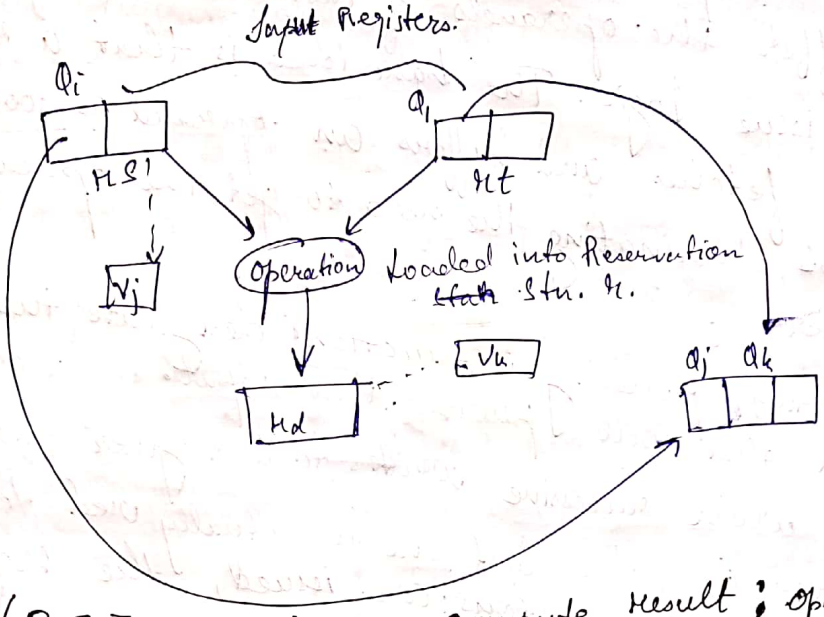
Op — the operation to perform on source operands.

The register file has a ~~field~~ field Q_i:

Q_i — the number of the reservation station that contains the operation whose result should be stored into this register.

TOMASULO'S ALGORITHM

Instruction State	Wait Until	Action or Bookkeeping
Issue	Station-r empty	if (RegisterStat[rs].$Q_i \neq 0$) {RS[r].$Q_j \leftarrow$ RegisterStat[rs].Q_i} else { RS[r].$V_j \leftarrow$ Regs[rs]; RS[r].$Q_j \leftarrow 0$}; if (RegisterStat[rt].$Q_i \neq 0$) {RS[r].$Q_k \leftarrow$ RegisterStat[rt].Q_i} else { RS[r].$V_k \leftarrow$ Regs[rt]; RS[r].$Q_k \leftarrow 0$}; RS[r].Busy $\leftarrow$ Yes; RegisterStat[rd].$Q_i = r$;



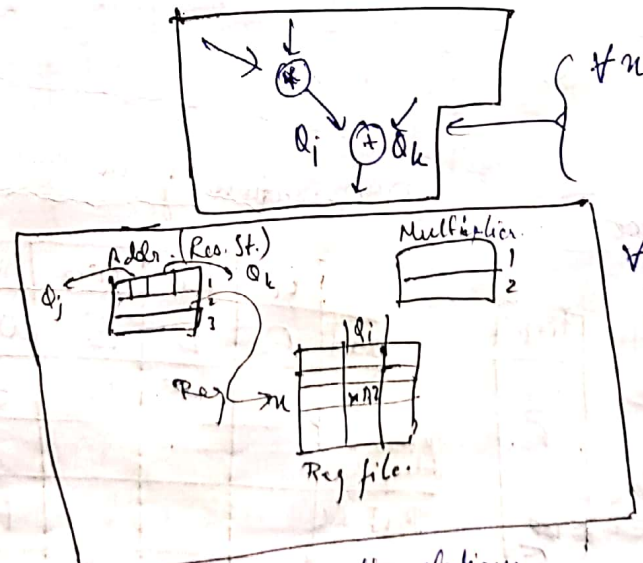
Execute (RS[r]. $Q_j = 0$) and compute result; operands are ($RS[r].Q_k = 0$) in V_j and V_k.

13/02/19

Instruction State	Wait Until	Action or Book-keeping

Write Result

Execution complete at r & CDB available



$\forall r$ (if RegisterStat[r].Q_i = r)
{ Regs[r] ← result;
RegisterStat[r].Q_i = 0 });

$\forall r$ (if RS[r].Q_j = r
{ RS[r].V_j ← result;
RS[r].Q_j ← 0 });

$\forall$ (if RS[r].Q_k = r)
{ RS[r].V_k ← result;
RS[r].Q_k ← 0 });
RS[r].Busy ← no;

Q_j, Q_k → The reservation stations which will provide input data for the operation.

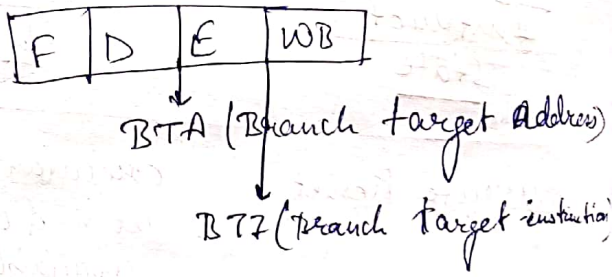
Register renaming is provided by the reservation stations, which buffer the operands of instructions waiting to issue, and by the issue logic. The basic idea is that a reservation station fetches and buffers an operand as soon as it is available, eliminating the need to get the operand from a register.

In addition, pending instructions designate the reservation station that will provide their input.

Finally when successive writes to a register overlap in execution, only the last one is actually used to update the register. As instructions are issued, the register specifiers for pending ~~operations~~ operands are renamed to the names of the reservation station, which provides register renaming.

Delayed Branching

```
add     r1, r2, r3
(branch) b
load
.
.
.
anywhere: sub
.
.
.
```



(a) Instruction Sequence

	t_i	t_{i+1}	t_{i+2}	t_{i+3}
add	F	D	E	WB
b		F	D	E
load			~~X~~	
BTI: sub				F

Wasted instruction slot

BTA

(b) Execution of the unconditional branch

	t_i	t_{i+1}	t_{i+2}	t_{i+3}	t_{i+4}
add	F	D	E	WB	
b		F	D	E	WB
Branch delay slot					
BTI: sub				F	D
			BTA	F	

(c) Execution without delayed branching

(d) The motion of a branch delayed slot.

We assume that:

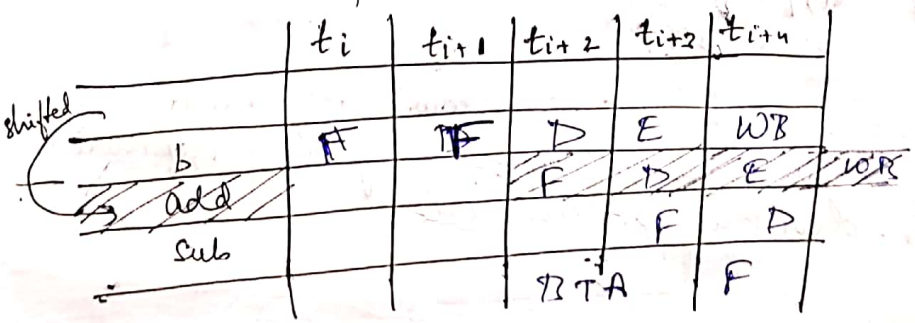
- the branch target address (BTA) becomes available at the end of the D-cycle.
- the referenced branch target instruction (BTI) can be fetched in ~~this~~ a single cycle (E cycle) from the cache.

Both assumptions indicate a fairly advanced implementation. In a straightforward case, the BTA would be calculated during the E-cycle and a cache access could take more than one cycle.

Instruction slots following branches are calculated called: branch delayed slots.

Branch delay slots are wasted during traditional execution. However, when delayed branching is employed, these slots can be utilised at least partially.

With delayed branching, the instruction that follows the branch is executed in the delay slot. However, the branch only will be effective later, that is, delayed by one cycle.



(effectively branch executes after add)

→ The add instruction that originally preceded the branch is moved into the branch delay slot. With delayed branching, the processor executes the add instruction first, but the branch will only be effective later.

→ Delayed branching reverses the execution sequence of the branch instruction and the instruction placed in delayed slot.

delayed ~~the~~ branching requires an architectural redefinition of the execution sequence of instructions compared with traditional von Neumann architecture.

Performance gain is achieved by filling the delayed slots as far as possible with appropriate instructions and executing them in the slot, instead of leaving the delay slots empty.
However, not every instruction is suitable for the delay slot. The only suitable instructions are those which are not dependent on preceeding ones and can be executed in a single pipeline cycle, like simple integer or boolean instructions.
Compilers can usually fill 60-70% of delayed slots with useful ~~information~~ instructions. Obviously the remaining delay slots are filled with NOPs.

Resource Dependencies

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.
If, for instance, only a single non-pipelined division unit is available, as is usual in ILP-processor, then in the code sequence:- (Instruction Level Parallelism)

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

the second division instruction is resource dependent on the first one and cannot be executed in parallel, if there is only a single-division unit available.

Control Dependencies

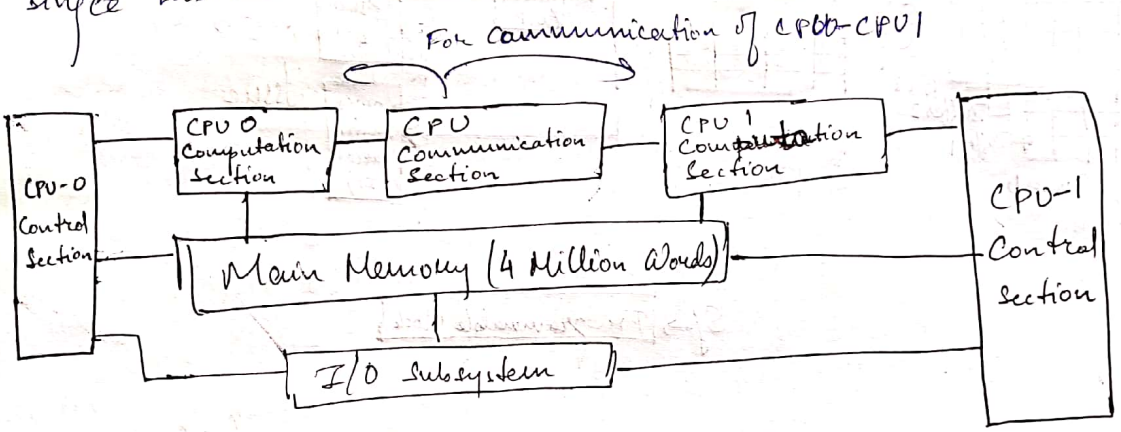
```
Mul  r1, r2, r3;
jz   zproc
Sub  r4, r1, r1;
...
zproc: Load r1, n;
```

The actual path of execution, depends on the outcome of the multiplication.
All conditional control instructions, such as conditional branches

calls, and so on, impose dependencies on the logically subsequent instructions.
These dependencies are referred to as control dependencies.

VECTOR PROCESSORS

By a vector processor we mean a processor that can perform an operation in a one-dimensional array of values in a single instruction.



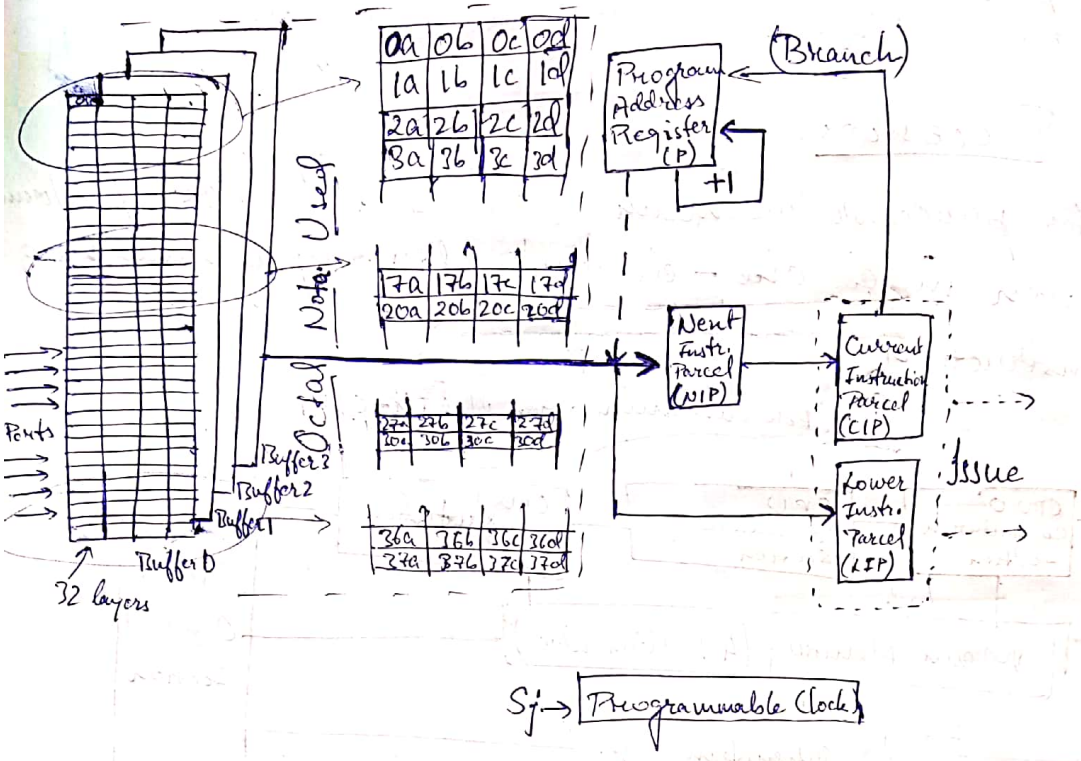
The CRAY X-MP/Model 24 Architecture.

Features:

→ Two identical CPUs, each with a 9.5ns cycle time.
→ The processors share a common main memory and I/O section.
→ The control section for each processor manages the instruction buffers, issues instructions, and controls the flow of information within other sections.
→ Each CPU has its own computation section consisting of registers and functional units. The registers include address registers, scalar registers, and vector registers.
There are 13 pipelined functional units dedicated to performing specific integer and floating point operations.

The Control Section of Cray - X-MP:

The Control Section of the Cray X-MP



→ The Cray X-MP has a load and store architecture.

→ Functional Units require register operands.

→ Most instructions DO NOT experience a delay due to operand fetches from memory.

→ Instructions are either 16 bits (1 parcel) or 32 bits (2 parcel) long.

Instruction Issue Phase

1-Parcel Instructions

During the instruction fetch, the first parcel of an instruction is transferred from the instruction buffer to the NIP (Next Instruction Parcel Register). This process takes one clock cycle. On the next clock cycle the parcel is moved to the CIP (Current Instruction Parcel), where it is decoded. The instruction is held in the CIP until it can be issued, that is until the processor is ready to execute it.

A one-parcel instruction will need only one clock cycle in the CIP unless there is a conflict because a previously issued instruction is using resources which will be required

during the execution of the current instruction. This situation is called a hold condition.

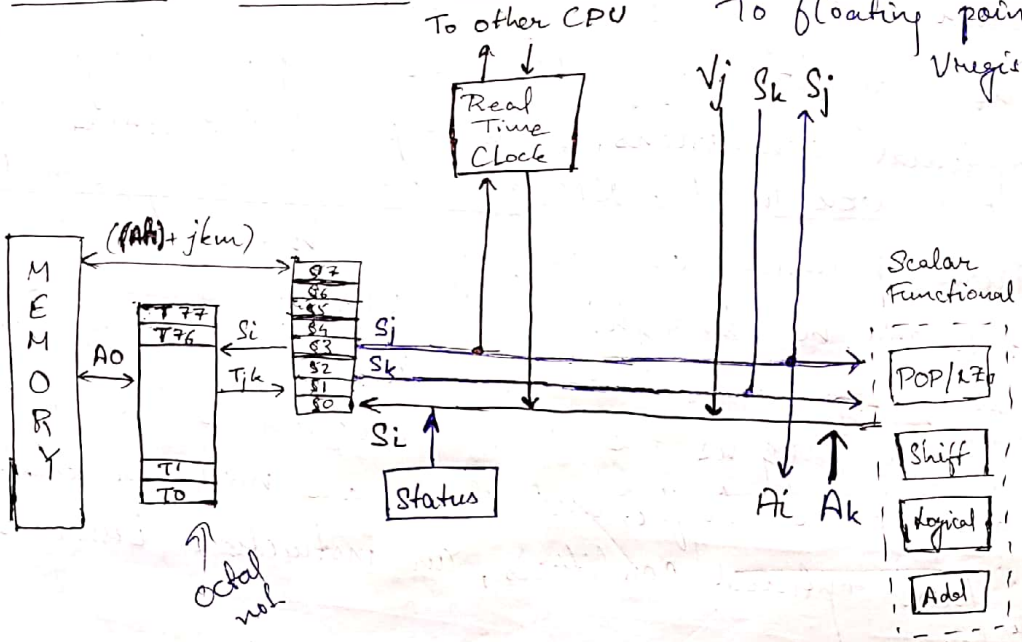
Most of the Cray X-MP instructions are one parcel long. Under optimal conditions, each 1-parcel instructions will spend one clock in the NIP, and one clock cycle in the CIP. These operations can overlap, so that the next instruction can be brought to the NIP, on the same clock cycle as the previous instruction, is brought to the CIP. This works as long as the next instruction is in the current instruction buffer, and there are no branches. Under these optimal conditions, an instruction issues on each clock cycle.

Branches: When a branch instructions reaches the CIP, it stays there until the instruction completes. No succeeding instructions can be issued until the branch is determined. Thus branch instructions spend their entire instruction cycle in the issue phase and no time in the execution phase.

2-Parcel Instructions

2-parcel instructions spend a minimum of 2 clock cycles in the CIP. In the cycle in which the first parcel moves from the NIP to the CIP, the second parcel is transferred from the instruction buffer to the NIP. The NIP is then filled with a zero (No operation) so that the processor does not issue an instruction on the next cycle.

Scalar Section Of The Cray X-MP



N.B: The scalar floating point are performed in the vector functional unit.

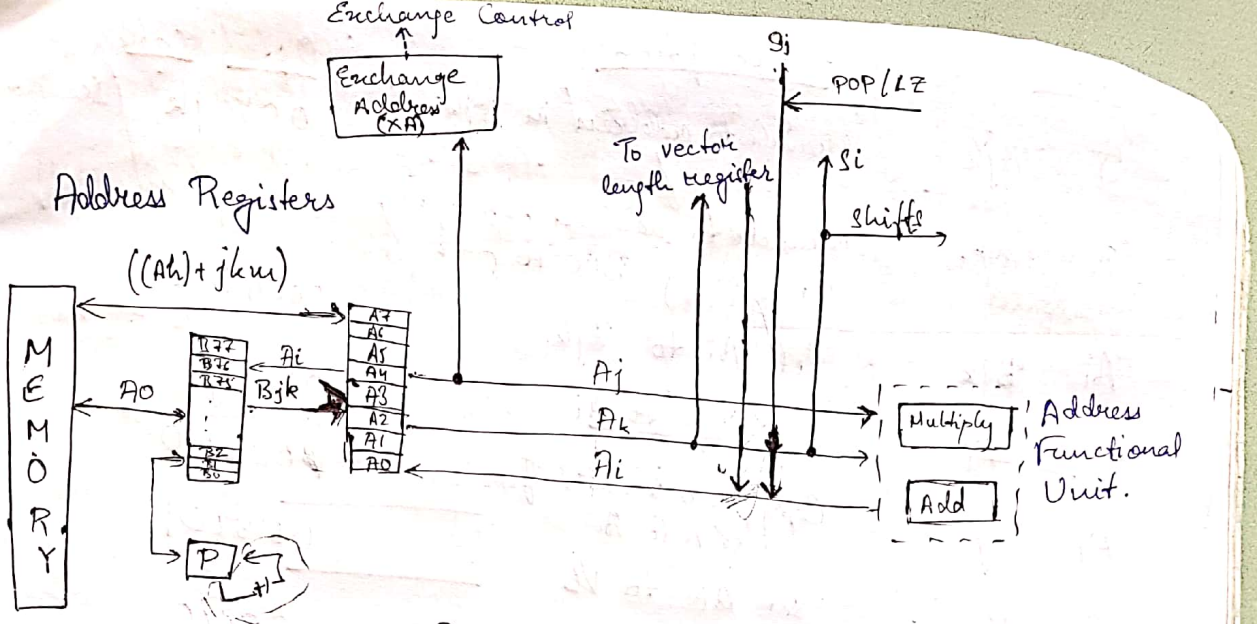
Scalar functional Unit	No. of stages
SCALAR ADD (64-bit integer)	3
" SHIFT (64-bit logical)	2
" SHIFT (128-bit logical)	3
" LOGICAL (64-bit logical)	1
" POP/PARITY (Population or parity)	4
" POP/PARITY (Leading zero Count)	3

SCALAR Output Operands are reserved

Line	Instruction	
1	S1	S2 + S3
2	S1	S2 + S3
3	S2	S1 + S7
4	S1	S4 + S1'

Instruction	1	2	3	4	5	6	7	8	9	10	11
1	I	E	E	E							
2				I	E	E	E				
3							I	E	E	E	
4								I	E	E	E

No dependency b/w 3 & 4



Address Functional Unit — No. of stages

ADDRESS ADD (24-bit integer) ——— 2

" MULTIPLY (") ——— 4

Address Register (A0 ... A7) → 8 24 bit registers. They hold addresses referenced in memory operations. Also used as index registers and for short integer computation.

B-Registers (B0 ... B77) → 64 24 bit registers. They are NOT connected to the functional units. They are used for saving registers during subroutine linkage or for temporary storage of addresses.

Addresses of the CRAY X-MP

→ All memory data is stored in words of 64 bits, and word addresses are specified by 22 bits.

→ Instructions take up either 16 bits (one parcel) or 32 bits (two parcels), and parcel addresses are 24-bit wide.

→ A and B register addresses are either interpreted as 22-bit word addresses or as 24-bit parcel addresses, depending on the instruction in which they occur.

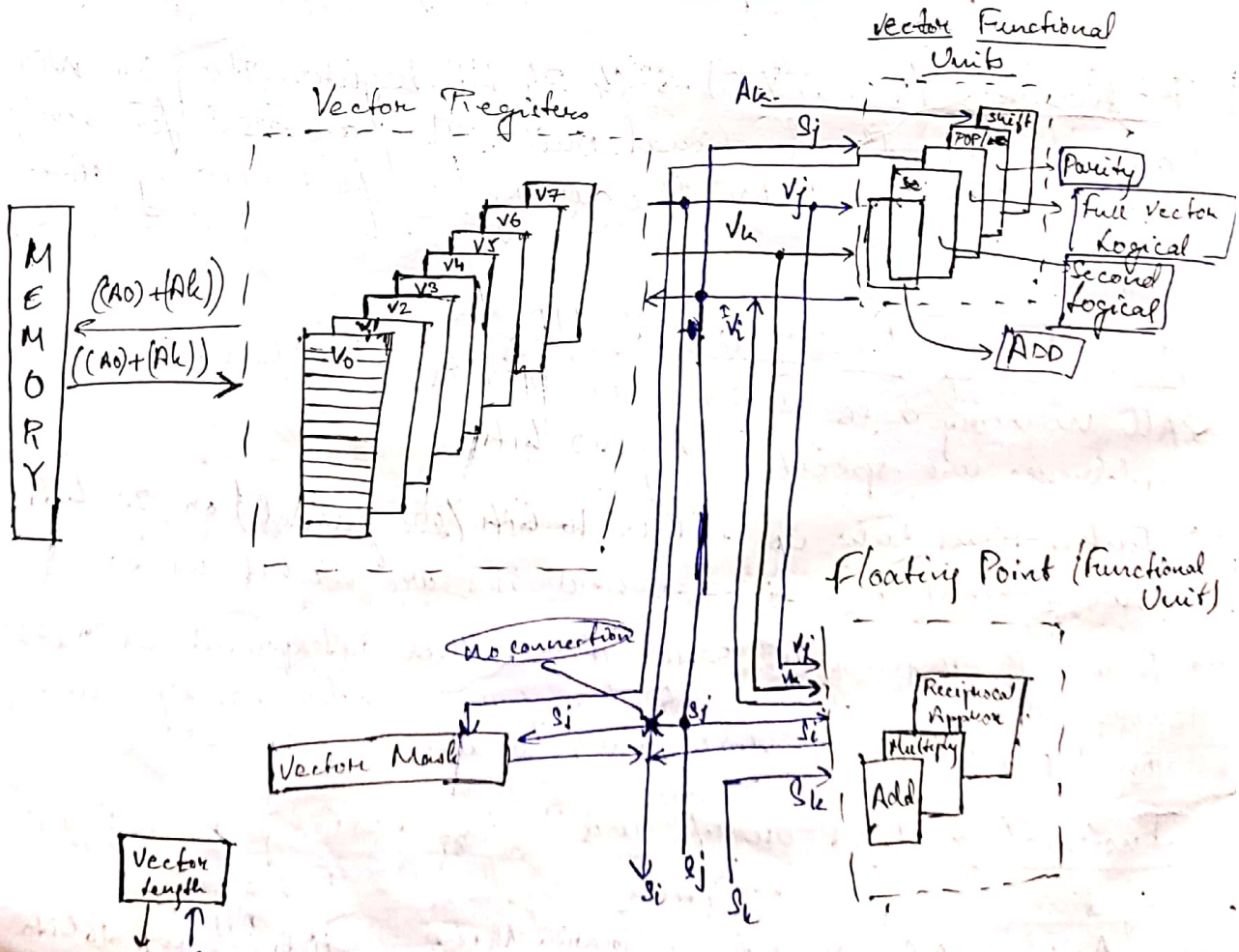
Instructions that use A and B registers: A few examples:

Syntax	Description	Octal Code
Ai exp A,k	Read Ai from location Ak+exp (exp is an immediate value: jkm)	10 hijkm | 16 bits
exp A,k Ai	Store Ai at location Ak+exp	11 hijkm

Syntax		Description	Octal code
J	Bjk	Branch to address in Bjk	0050jk
R	exp	Branch to address exp=ijkm after setting B00 to p+2	0071jkm
Ai	Bjk	Set Ai to Bjk	0024ijk
Bjk	Ai	Set Bjk to Ai	0025ijk
Ai	Sj	Copy loco 24-bits of Sj into Ai	0023ij0
Ai	VL	Set Ai to VL	023i01
Ai	LZSj	Set Ai to leading zero count of Sj. Ai = 64 if j=0	027ij0
Ai	Aj+Ak	Set Ai to Aj+Ak	030ijk

(See add. size)

The VECTOR Section of the CRAY X-MP



A vector is defined as a collection of values which are stored in memory locations spaced a fixed distance apart. The spacing of the elements is called a stride of the vector.

Parallelism and computational efficiency are achieved in a vector processor when a significant portion of a program's data can be mapped into vectors.

The same operation can be performed on all of the elements of a vector with results becoming available in consecutive clock cycles.

Basic Operation of the vector Section

→ Each processor of the Cray-X8 has 8 vector processors, $V_0 \dots V_7$.

Each register is 64 words long.

→ Since there are no direct connections from main memory to the vector functional units, all vector arithmetic operations are performed through the vector registers.

→ A vector is processed by reading its components into the consecutive elements of the vector register starting with element 0 and going through element VL-1. VL is the value of the vector length register when the instruction is issued.

→ The source and the destination vector registers and the functional units are reserved during vector operations.

→ All of the vector functional units are pipelined. The units are designed to perform the same operation on each element of a vector.

→ Once the pipeline of a functional unit is filled, the unit will produce a result on every clock cycle.

→ There are three phases of the pipelined operation.

→ The first is called the setup phase. During this phase, the functional unit is set to perform the appropriate operation, and the source and the destination routes to the vector register are established. The setup time for all of the vector functional units is 2 clock cycles.

→ The second phase is called the execution phase. During each clock in this phase, one pair of source elements enters the first stage of the pipeline and the result computed for the previous pair is sent to the next stage.

Vector Functional Unit	No. of Stages
VECTOR ADD (64 bit integer)	3
VECTOR SHIFT (64 bit logical)	3
VECTOR SHIFT (128 bit logical)	4
FULL VECTOR LOGICAL (64 bit logical)	2
SECOND VECTOR LOGICAL (64 bit logical)	4
VECTOR POP/PARITY	5
FLOATING ADD	6
FLOATING MULTIPLICATION	7
RECIPROCAL APPROXIMATION	14

One cycle after the last pair of input elements has entered the pipeline, the functional unit can be used for another operation. Thus the functional unit will be available ~~for~~ in:

$$3 + VL + 1 = VL + 4 \text{ clock cycles.}$$

Here VL is the value of the vector length register when the instruction issues.

→ The source operands become available immediately after the last pair of input elements has entered the pipeline. Vector source registers are therefore reserved for VL + 3 exit clock cycles.

→ The third phase of a pipelined operation is the shutdown phase. The shutdown time is the difference between the time when the last individual result emerges from the pipeline and the time when the destination vector register becomes available for another operation.

The shutdown time is 3 clock cycles, so the destination register becomes available in $3+n+(VL-1)+3 = n+VL+5$ clock cycles where n is the number of stages in the pipeline and VL is the number of component operations to be performed (chaining is an exception to this rule).

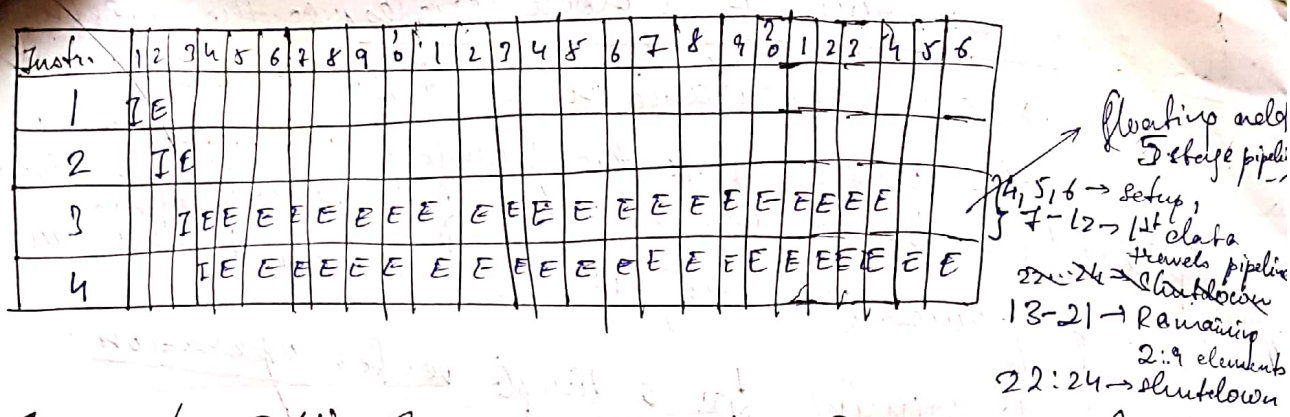
Example 1: Timing for a simple vector operation:

Line	Instruction		Description
1	A1	S3	Set register A1 to S3
2	VL	A1	set the length of vector to S3.
3	V4	V2 + V3	Perform the addition.

Setup time ——— 3 cycles ——— 3 cycles
(Vector integer adder is a 3 stage pipeline)
Time required to compute first result ——— 3 cycles
First result emerges after 3+3 = ——— 6 cycles
Time required to compute remaining 52 results ——— 52 cycles
Shutdown time ——— 3 cycles
Destination register V4 will be available after 6+52+3 = ... 61 cycles
The functional unit i.e adder can be released after VL+4 ... = 57 cycles
The vector source registers V2 and V3 are reserved for VL+3 = 56 cycles

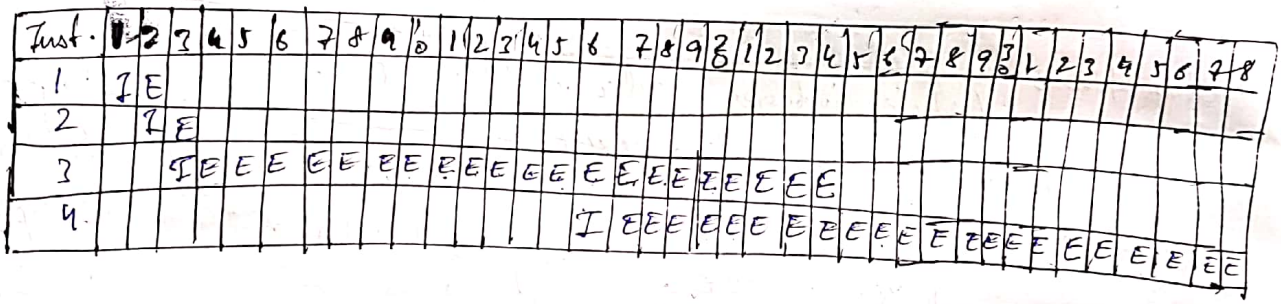
Example-2(a): No source reservation conflict.

Line	Instruction		Description
1	A1	10	Set A1 to 10
2	VL	A1	Set the vector length to 10
3	V4	V3+FV2	Set V4 to floating sum of V3 and V2.
4	V6	V5*FV7	Set V6 to floating product of V5 and V7.



Example 2(b): Source Reservation Results in a Conflict

1	A1	10
2	VL	A1
3	V4	V3 + FV2
4	V6	V3 * FV7

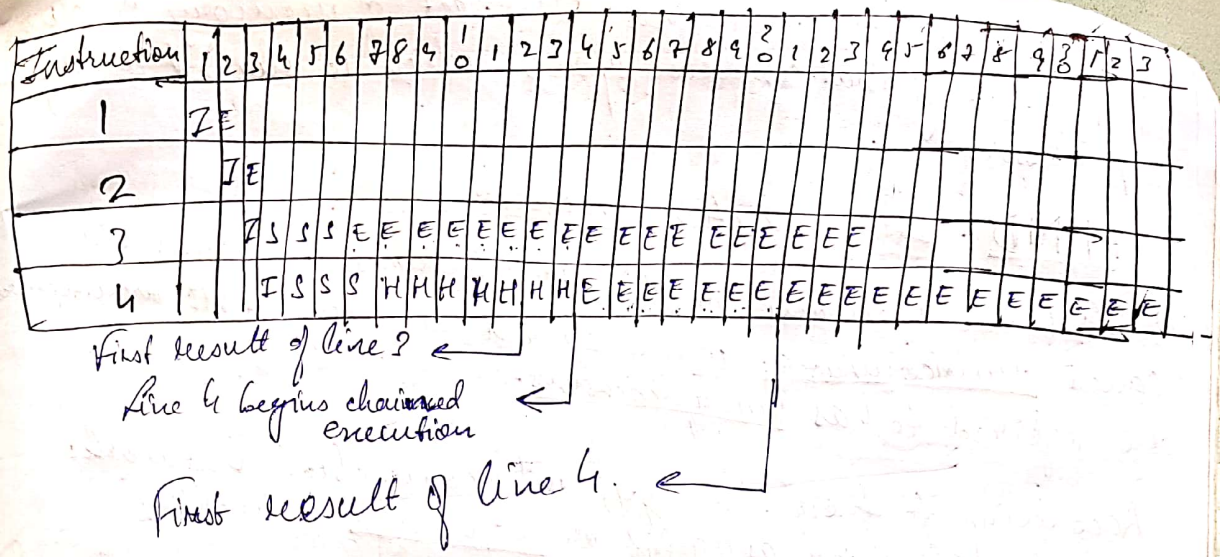


Vector Chaining

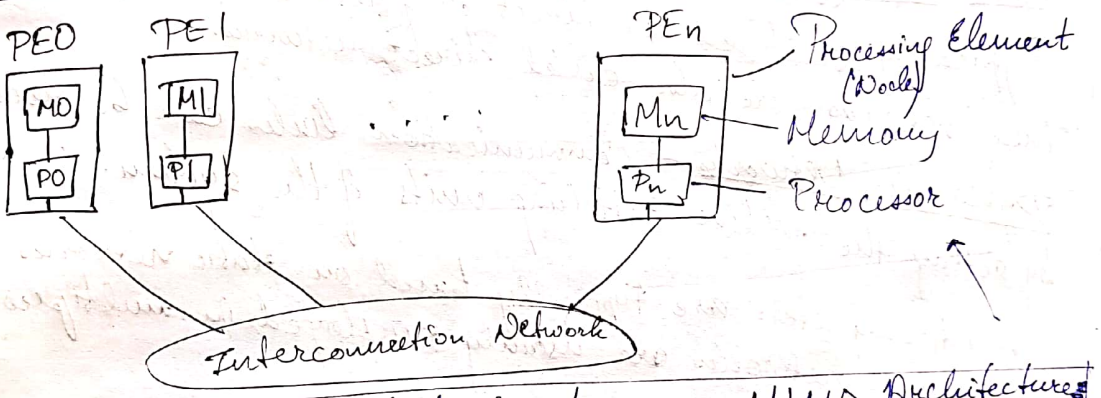
1	A1	10
2	VL	A1
3	V4	V3 + FV2
4	V5	V4 * FV7

In chaining the results produced by one operation can be used as input to a succeeding operation BEFORE the first instruction has completed. That is, the component results from the first instruction are used by the second instruction as they are produced.

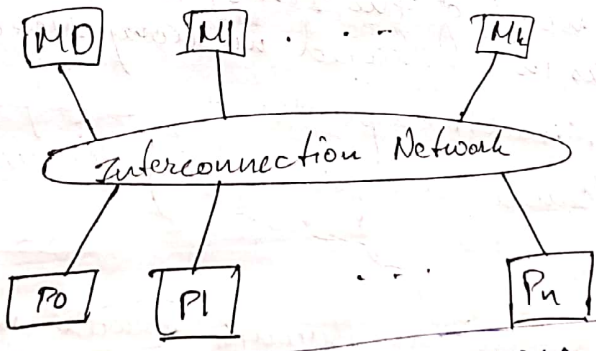
With chaining, a value which emerges from the pipeline may be used by a waiting operation 2 cycles after it is produced. This is in contrast to unchained operations where the destination register cannot be accessed until 7 cycles after the last component computation is completed.



MIMD Architecture



Structure of Distributed Memory MIMD Architectures



Structure of Shared Memory MIMD Architectures

1) Distributed Memory (or message passing)
— Whenever interaction among PEs is necessary, they send messages to each other. None of the PEs can ever access the memory module of another PE directly.

2) Shared Memory.
— Any processor can access any memory module via

interconnection network. The set of memory modules defines a global address space which is shared among the processors.

MIMD Interconnection Network

Distributed memory MIMD architectures are often simply called multicomputers while shared memory MIMD architectures are referred to as multiprocessors.

According to their topology, interconnection networks can be classified as static or dynamic networks.

Static Networks - Connection of switching units is fixed and typically realized as direct or point-to-point connections. These networks are also called direct networks.

Dynamic Networks - Communication links can be reconfigured by setting the active switching units of the system.

Multicomputers are typically based on static networks while dynamic networks are mainly employed in multiprocessor.

Distributed Memory System: ADVANTAGES:

→ Since processors work ~~until~~ with their attached local memory module most of the time, the contention problem is not so severe as in shared memory systems. As a result distributed memory multicomputers are highly SCALABLE and good architectural candidates for building MASSIVELY PARALLEL COMPUTERS.

→ Processors cannot communicate through shared data structures and hence sophisticated synchronization techniques like monitors are NOT NEEDED. Message passing solves not only COMMUNICATION but SYNCHRONIZATION, as well.

DISADVANTAGES:

→ In order to achieve high performance in multicomputers, special attention should be paid to LOAD BALANCING and synchronization.

→ Message passing based communication can lead to deadlock situations.

→ Although there is no architectural bottleneck in multicomputers message passing requires physical copying of data structures between processors. Intensive data copying can result in significant performance degradation.

Shared Memory Systems: ADVANTAGES:

→ There is no need to partition either the code or the data; therefore uniprocessor programming techniques can easily be adapted; in the multiprocessor environment.
Neither new programming languages nor sophisticated compilers are needed to exploit shared memory systems.

→ There is no need to physically move data when two or more processes communicate. The consumer process can access the data from the place where the producer stored it. As a result, COMMUNICATION between processes is VERY EFFICIENT.

Shared Memory Systems: Disadvantages:

→ Synchronized access to shared data structures requires special synchronizing constructs such as semaphores, conditional critical regions, monitors, and so on.
Usually, message-passing synchronization is simpler to understand and apply.

→ The main disadvantage of shared memory systems is LACK OF SCALABILITY, due to the contention problem. When several processors want to access the same memory module they must compete for the right to do so. The winner can access the memory, while the losers must wait.
The larger the number of processors the higher the probability of memory contention. Beyond a certain number processors, this probability is so high that adding a new processor to the system will not increase performance.

Shared Memory MIMD Architectures

The distinguishing feature of shared memory system is that no matter how many memory blocks are used in them and how these memory blocks are connected to the processors, the address spaces of these memory blocks are unified into a global address space which is completely visible to all processors.

Design issues

The three main design issues in increasing the scalability of shared memory systems are:

- Organisation of memory
- Design of interconnection network
- Design of cache coherent protocols.

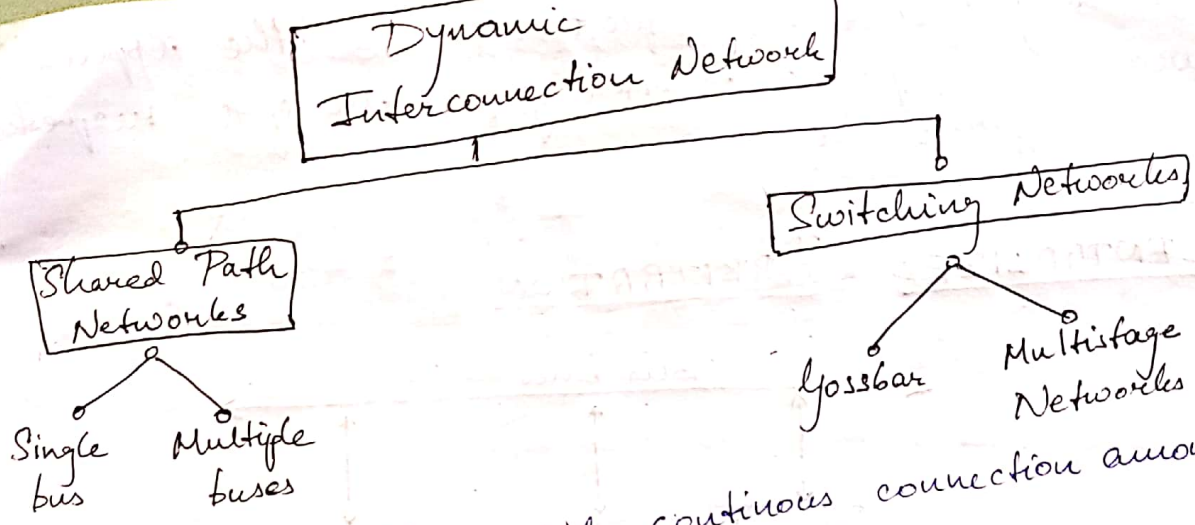
Organisation of Memory

Shared memory systems are basically classified according to their memory organisation, since this is the most fundamental design issue. Accordingly, shared memory systems can be divided into four main classes:

- Uniform - Memory Access (UMA) machines
- Non - Uniform Memory Access (NUMA) machines
- Cache - coherent Non - Uniform Memory Access (cc-NUMA) m/c.
- Cache only Memory Access (COMA) machines.

Shared Memory MIMD: Interconnection Networks

The quality of the interconnection network has a decisive impact on the speed, size and cost of the whole machine. Dynamic interconnection schemes are usually employed in multiprocessors.

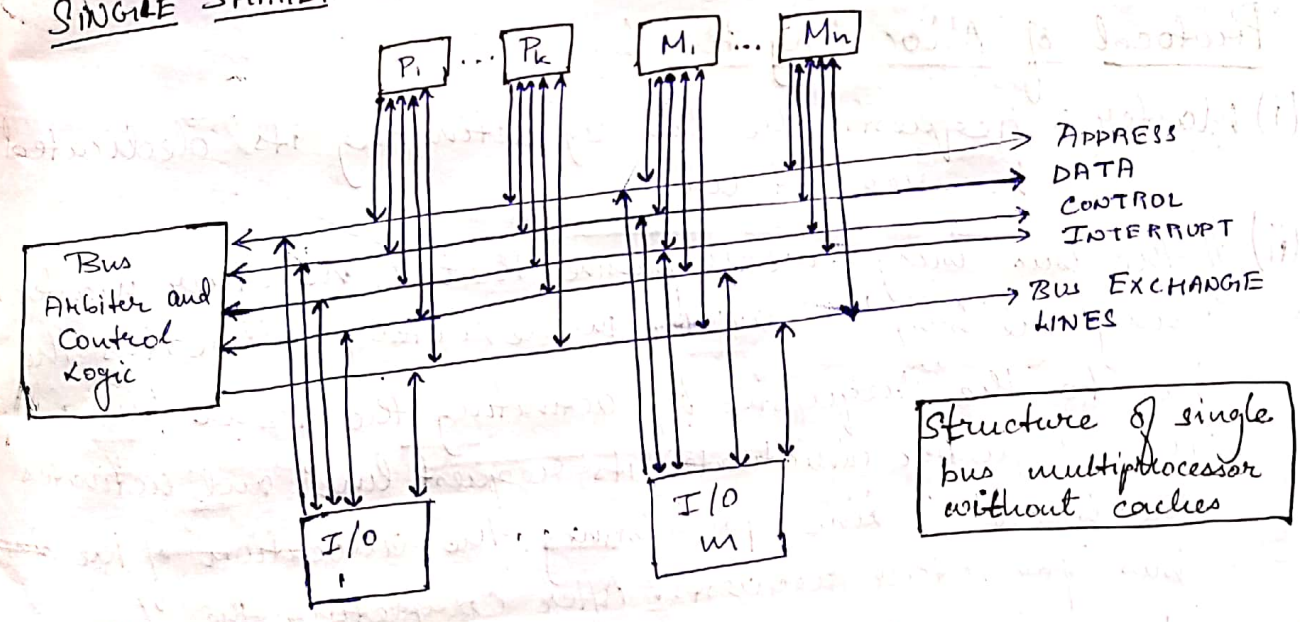


Shared path networks: provide continous connection among the processors and memory blocks.

The continous connection is shared among the processors which have to compete for its use.

Switching networks: do NOT provide a continous connection among the processors and memory blocks. A switching mechanism enables processors to be temporarily connected to memory blocks.

SINGLE SHARED BUS



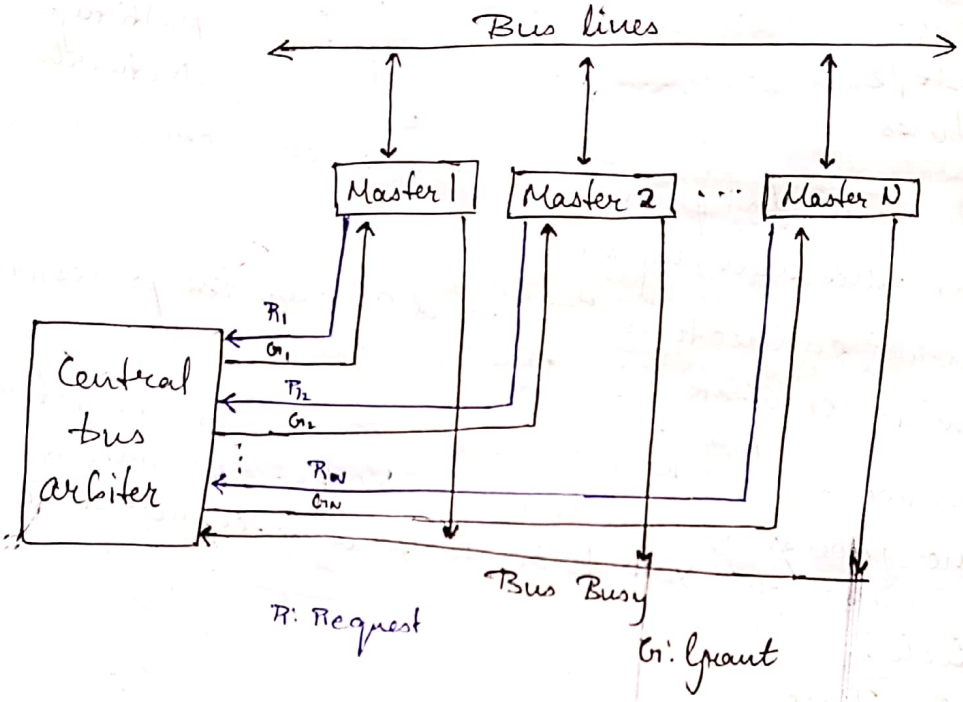
Structure of single bus multiprocessor without caches

Bus- Arbiter Logic –
Allocates the bus in the case of several simultaneous bus requests

Bus- Exchange Lines –
Comprise typically of one or more bus request lines by means of which the processor or other temporary bus masters can request bus allocation.

According to the bus request lines and the applied bus allocation policy, the arbiter grants one of the requesters via the grant lines.

CENTRALIZED ARBITRATION



Protocol of Allocating the bus

(i) Master: Requests the bus by activating its dedicated bus request line.

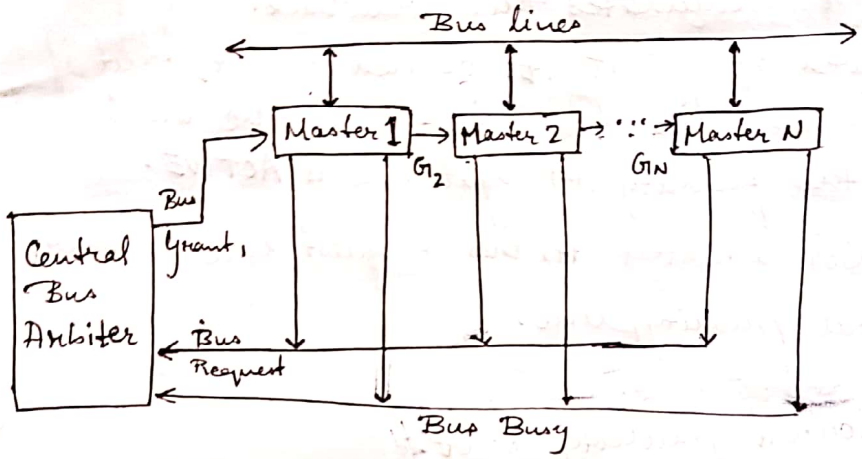
(ii) If the bus busy line is passive, that is, no other master is using the bus, the arbiter immediately allocates the bus to the requester by activating the grant line. The requester deactivates its request line and activates the bus busy line, prohibiting the allocation of the ~~lines~~ bus for other requests. After completing the bus transaction, the requestor deactivates the bus busy line.

(iii) When the bus busy line is active, the arbiter DOES NOT ACCEPT any bus requests.

(iv) When several request lines are active by the time the bus busy line becomes passive, the arbiter can use any of the following bus allocation policies:

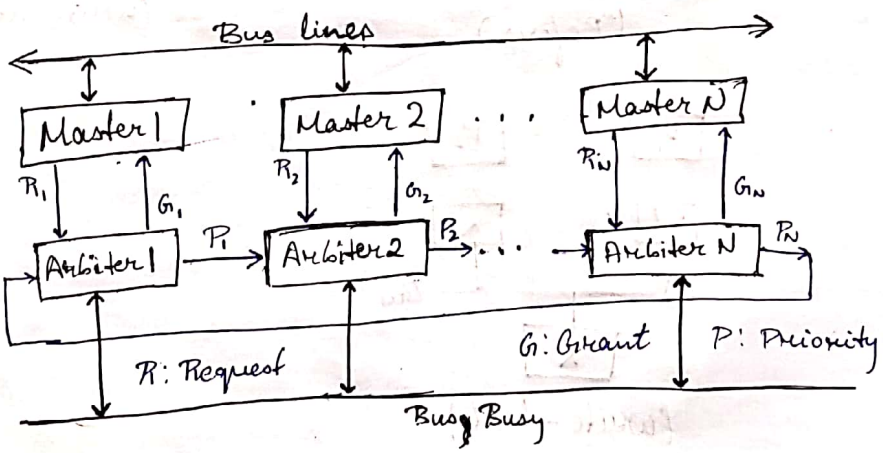
fixed priority, rotating priority, round robin, least recently used, first come first served

DAISY-CHAINED BUS ARBITRATION



- One of the most popular organisations of arbiter logics is daisy-chaining.
- There is only ONE shared bus request line. All the masters use this line to indicate their need to access the shared bus.
- The arbiter passes the bus grant line to the first master and then it is passed from master to master, creating a chain of masters.
- The priority of a master is determined by its position in the grant chain. The closer it is to the arbiter, the higher its priority.
- A master can access the shared bus if the bus busy line is passive and its input grant line is active.
- When the master does not require the bus and receives an active grant line, it activates its output grant line, enabling the next master to use the bus.

DECENTRALIZED ROTATING ARBITER

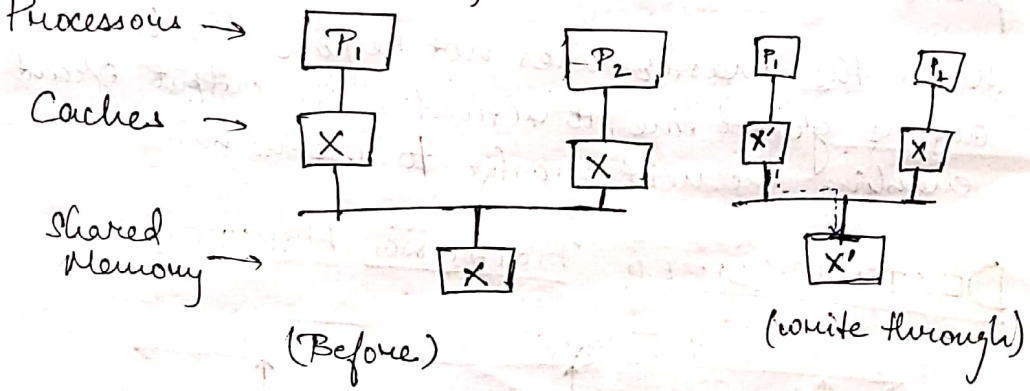


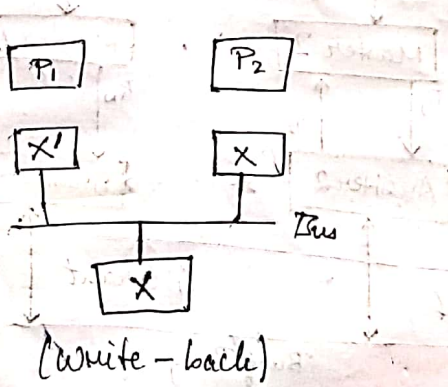
- Daisy chained scheme lacks fairness.
- Rotating arbiter eliminates this drawback.
- Arbiter$_i$ is allowed to grant its coupled master$_i$ unit if master$_i$ has activated its bus request line, the bus busy line is passive, and the priority (i-1) input line is ACTIVE.
- If master$_i$ has NOT activated its bus request line arbiter$_i$ activates its output priority$_i$ line.
- Selecting the lowest priority unit:
  - daisy-chained arbiter → the master farthest away.
  - rotating arbiter → the master that releases the bus.

CACHE COHERENCE

When multiple processors maintain locally cached copies of a unique shared-memory location, any local modification of the location can result in a globally inconsistent view of memory. Cache coherence schemes prevent this problem, by maintaining a uniform state for each cached block of data.

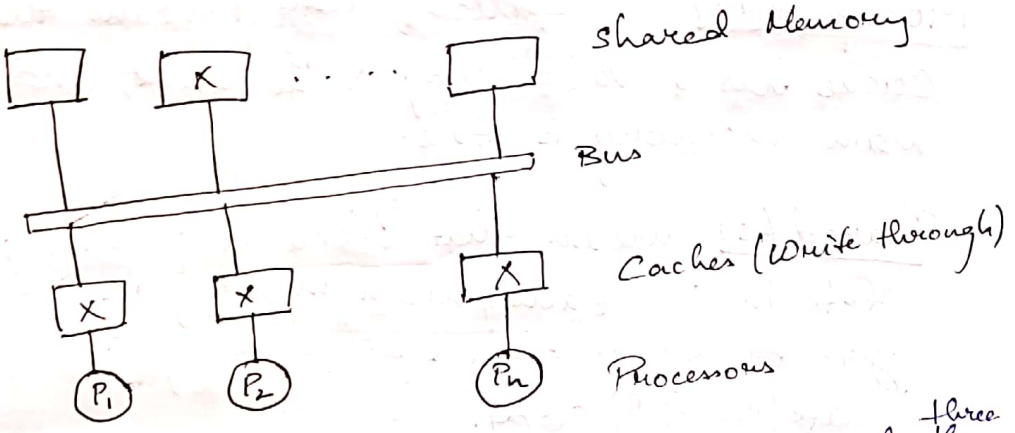
Inconsistency in Data Sharing



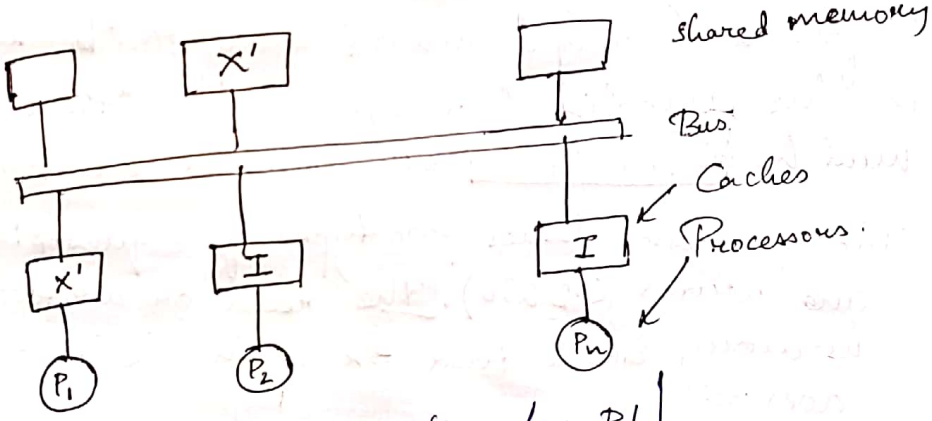


Snoopy Protocols

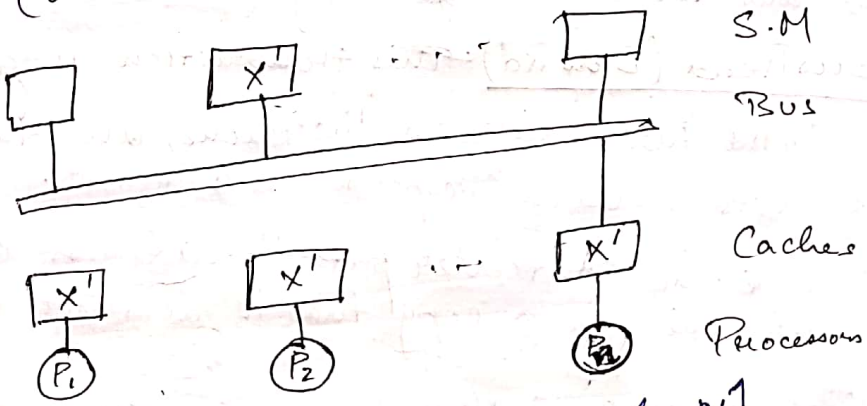
Snoopy protocols achieve data consistency among the caches and shared memory through a bus-watching mechanism.



Consistent copies of block X are in shared memory and three processor caches. (others are not interested)



After a write-invalidate operation by P1 (write invalidate protocol)



After a write-update operation by P1 (write-update protocol)

MSI Write-Back Invalidation Protocol

Three states of a cache:

Modified (M) (also called "dirty") means that only this cache has a valid copy of the block, and the copy in main memory is stale.

Shared (S) means that block is present in an unmodified state in the cache, main memory is up-to-date and zero or more other caches may also have an up-to-date (shared) copy.

Invalid (I) means that the block is not present in cache.

Before a shared or invalid block can be written and placed in the modified state, all the other potential copies must be invalidated via a read-exclusive bus transaction.

The processor issues two types of requests: reads ($P_r Rd$) and writes ($P_r Wr$). The read or write could be to a memory block that exists in the cache or to one that does not.

MSI Protocol: Bus Transactions

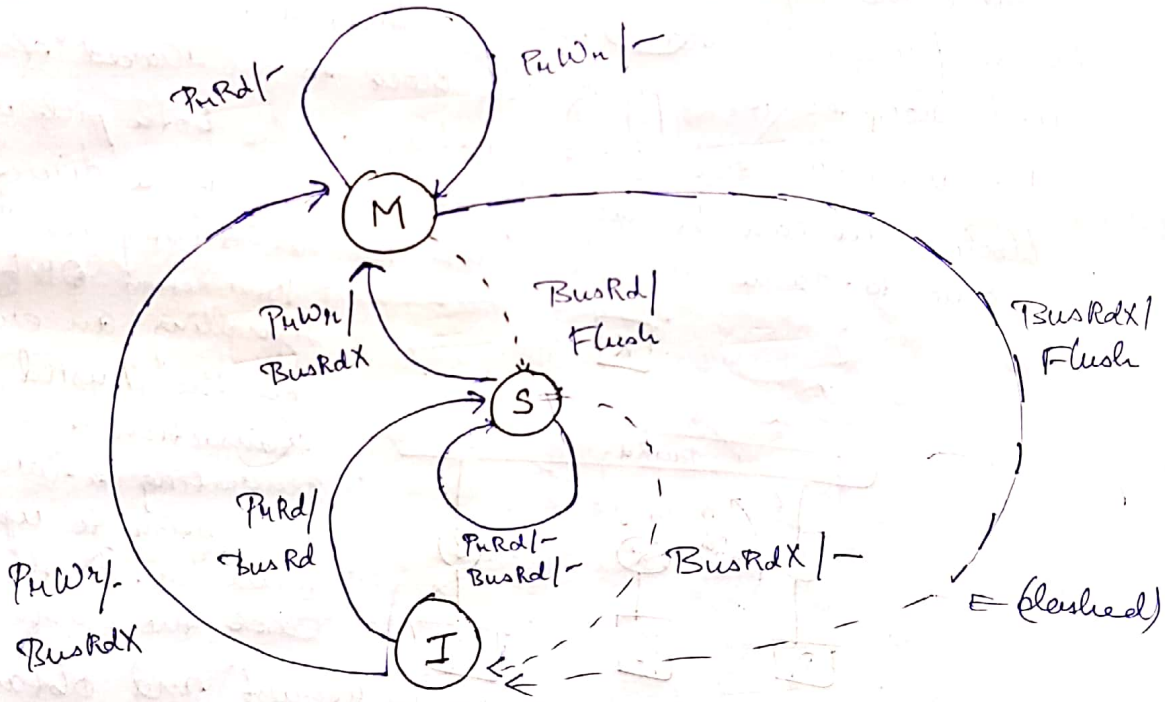
The bus allows the following transactions:

- **BusRead (BusRd)**: This transaction is generated by a $P_r Rd$ that misses in the cache, and the processor expects a data response as a result.
  The cache controller puts the address on the bus and asks for a copy that it does not intend to modify.
  The memory system (possibly another cache) supplies the data.
- **BusReadExclusive (BusRdX)**: This transaction is generated by a $P_r Wr$ to a block that is either not in the cache or is in the cache, but not in the modified state.
  The cache controller puts the address on the bus and asks for an exclusive copy that it intends to modify

The memory system (possibly another cache) supplies the data. All other caches are invalidated. Only this cache obtains the exclusive copy, the write can be performed in the cache.

Bus Write Back (Bus WB): This transaction is generated by a cache controller on a write back; the processor does not know about it and does not expect a response. The cache controller puts the address and the contents for the memory block on the bus. The main memory is updated with the latest contents.

MSI Protocols: State Transitions



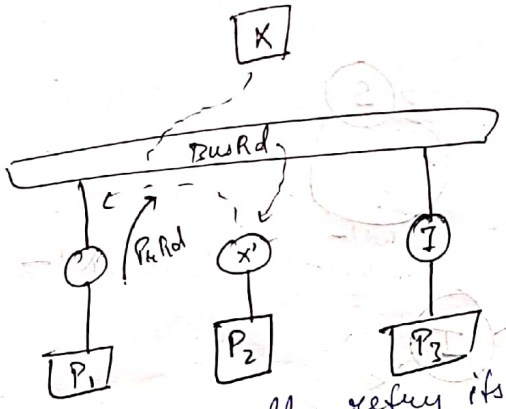
Notation A/B: If the controller observes the event A from the processor side or the bus side, then in addition to the state change, it generates the bus transaction or action B.

"–" means null action.

Transactions due to observed bus transactions are shown in dashed arcs while those due to local processor actions are shown in bold arcs.

(Replacements and the write backs they may cause are not shown in the diagram.)

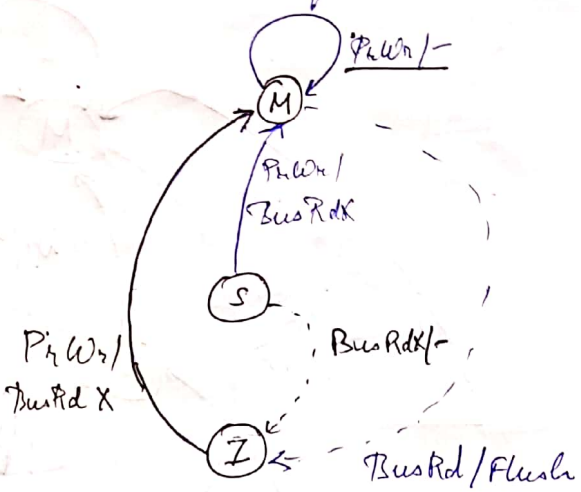
A processor read to a block that is invalid (or not present) causes a BusRd transaction to service the miss. The newly loaded block is promoted, moved up in the state diagram, from invalid to the shared state in the requesting cache, whether or not another cache holds a copy. Any other caches with the block in the shared state observe the BusRd but take no special action, allowing main memory to respond with the data. However if a cache has the block in the modified state (there can be one) and it observes a BusRd transaction on the bus, then it must get involved in the transaction since the copy in main memory is stale. This cache flushes the data onto the bus, in lieu of memory, and demotes its copy of the block to the shared state. The memory and the requesting cache both pick up the block. This can be accomplished either by a direct cache-to-cache transfer across the bus during the BusRd transaction OR by signalling an error on the BusRd transaction and generating a write transaction to update memory. In the latter case, the original cache will eventually retry its request and obtain the block from memory.



MSI State Transitions (Contd.)

Writing into an invalid block is a write miss, which is serviced by first loading the entire block and then modifying the desired bytes within it. The write miss generates a read exclusive bus transaction which causes all other cached copies of the data block to be invalidated, thereby granting the requesting cache exclusive ownership of the block. The block of data returned by the read exclusive is promoted to the modified state and the desired bytes are then written into it. If another cache requests exclusive access, then

in response to its BusRdX transaction this block will be invalidated (demoted to the invalid state) after flushing the exclusive copy to the bus.



MSI Drawback

When a process reads in and modifies a data item, two bus transactions are generated:

(i) A BusRdX that gets the memory block in the S state.

(ii) A BusRdX that converts the block from S to M state. [takes place even if no other processor is sharing this data]

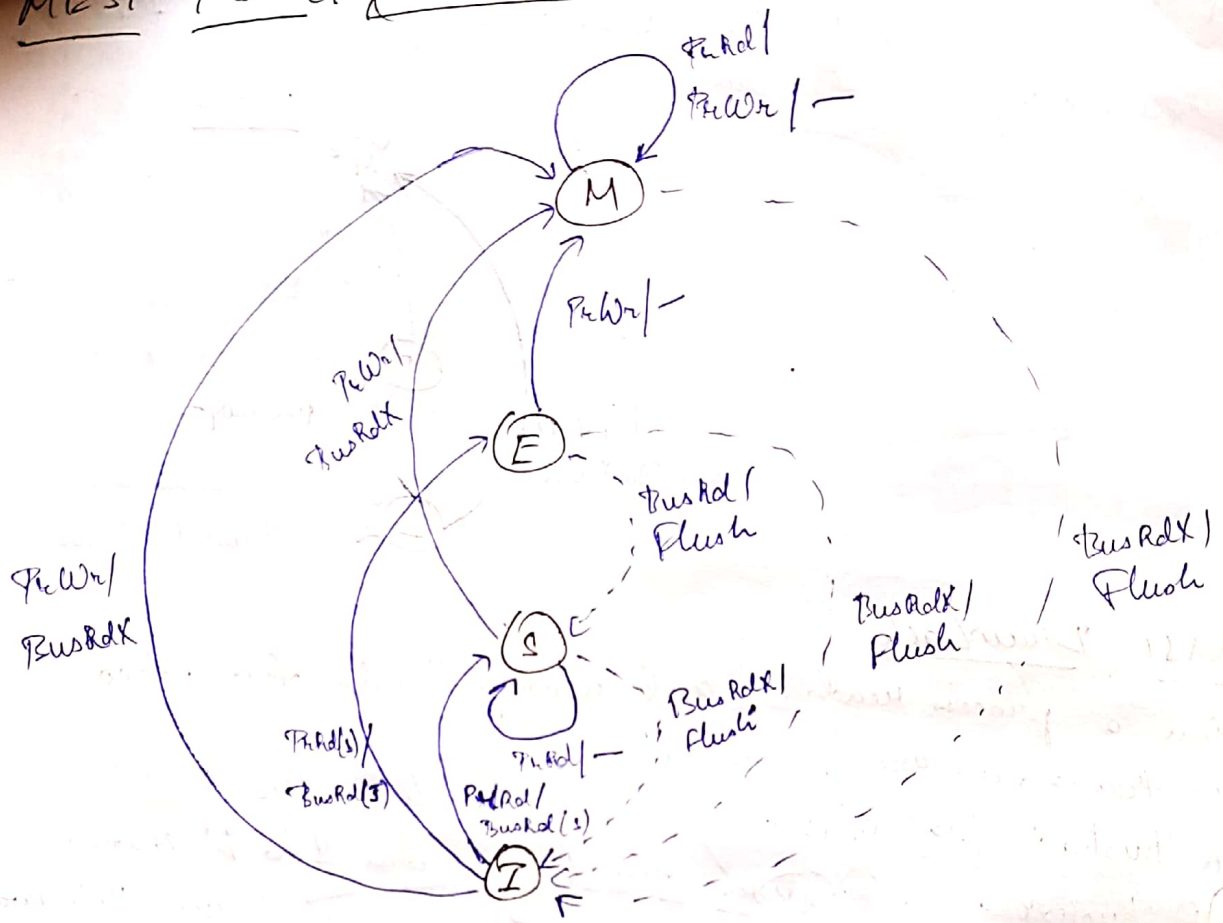
→ It is not known whether other processors are sharing this block; if they do, those copies must be invalidated.

MESI Protocol

By adding a state that indicates that the block is the ONLY (EXCLUSIVE) copy but is not modified, and by loading the block in this state, we can save the second transaction since the state indicates that no other processor is caching the block.

This new state, called exclusive clean or exclusive owned (or even simply exclusive) indicates an intermediate level of binding between shared and modified. It is Exclusive so unlike the shared state, the cache can perform a write and move to the modified state, without further bus transactions, but it does not imply ownership (memory has a valid copy), so unlike the modified state, the cache need not reply upon observing a request for the block.

MESI Protocol (State Transitions)

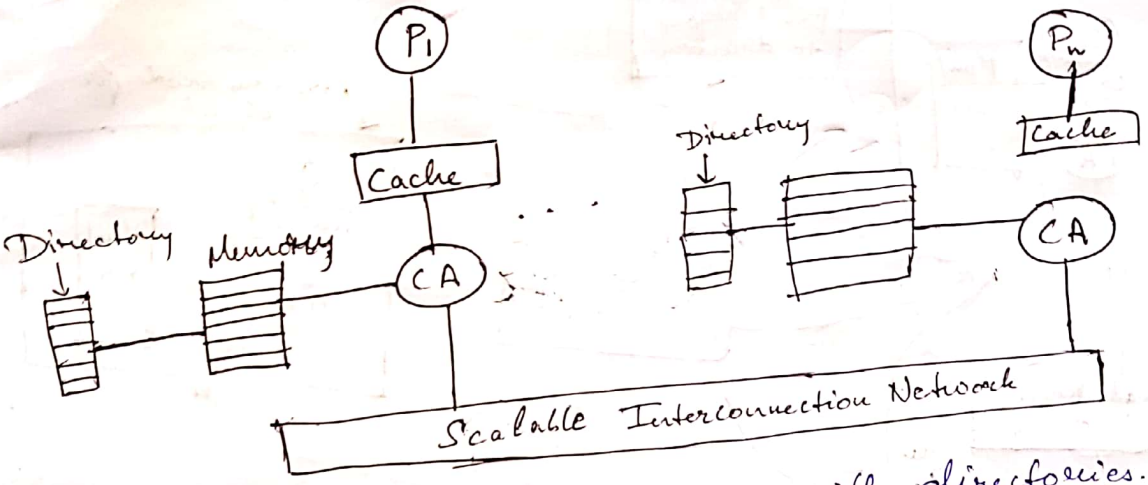


This protocal places a new requirement on the physical interconnect of the bus. An additional signal called the **shared signal(S)** must be available to the controllers in order to determine on a BusRd if any other cache currently holds the data. During the address phase of the bus transaction, all caches determine if they contain the requested block and, if so, assert the shared signal.

Flush': A cache, rather than main memory, supplies data for BusRd and BusRdX. Flush' applies to the cache that supplies the data.

DIRECTORY-based Cache Coherence

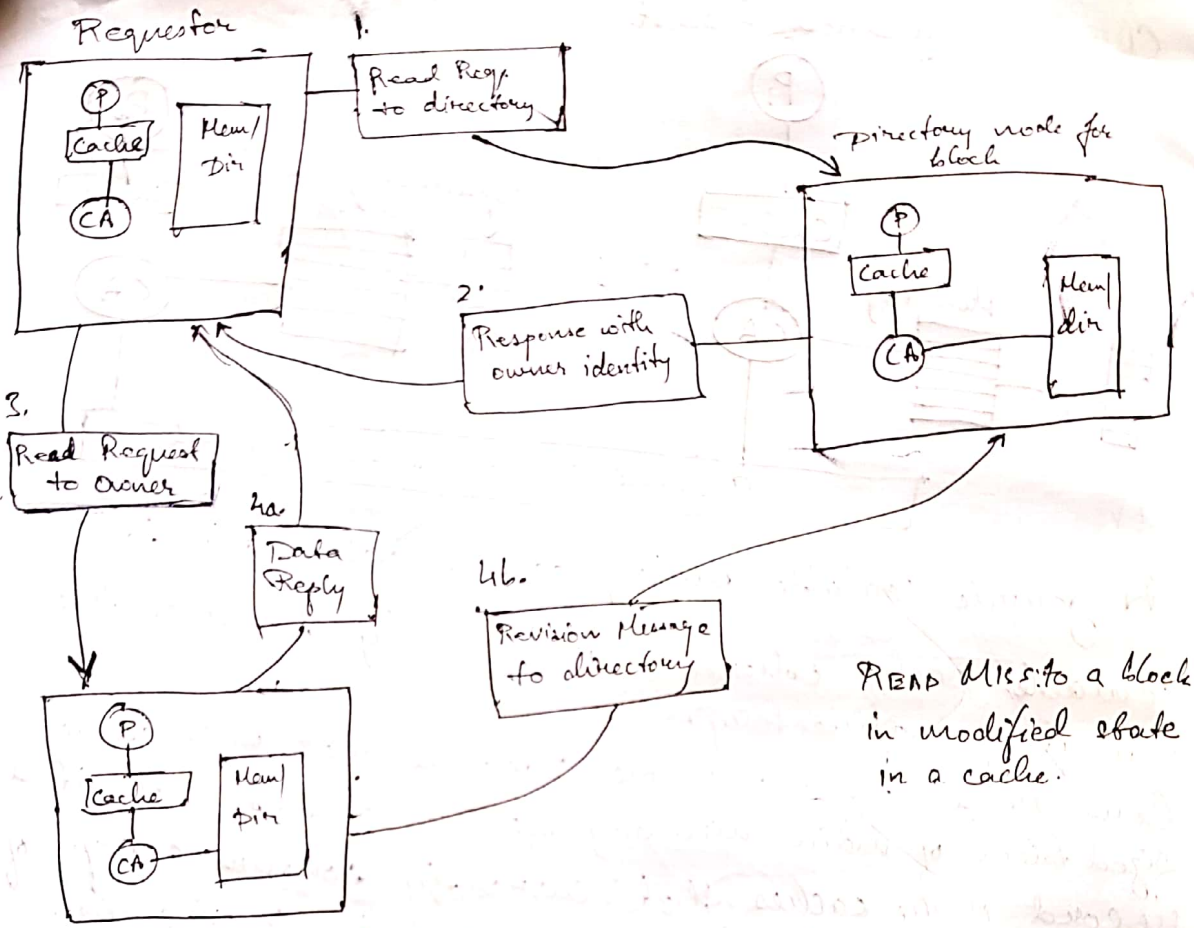
CA = Communication Assist.



A generic scalable multiprocessor with directories. Scalable cache coherence is typically based on the concept of a directory.

Consider a simple example. Imagine that each cache-line-sized block of main memory has associated with it a record of the caches that currently contain a copy of the block and the state of the block in those caches. This record is called the directory entry for that block. When a node incurs a cache miss, it first communicates with the directory entry for that block using point-to-point network transactions. From the directory, the node determines WHERE the valid cached copies are and what further action to take. It then communicates with the cached copies as necessary using additional network transactions. For example, it may obtain a modified block from another node.

Operation of a Simple Directory Scheme



READ MISS to a block in modified state in a cache.

Home Node: node in whose memory the block is allocated.

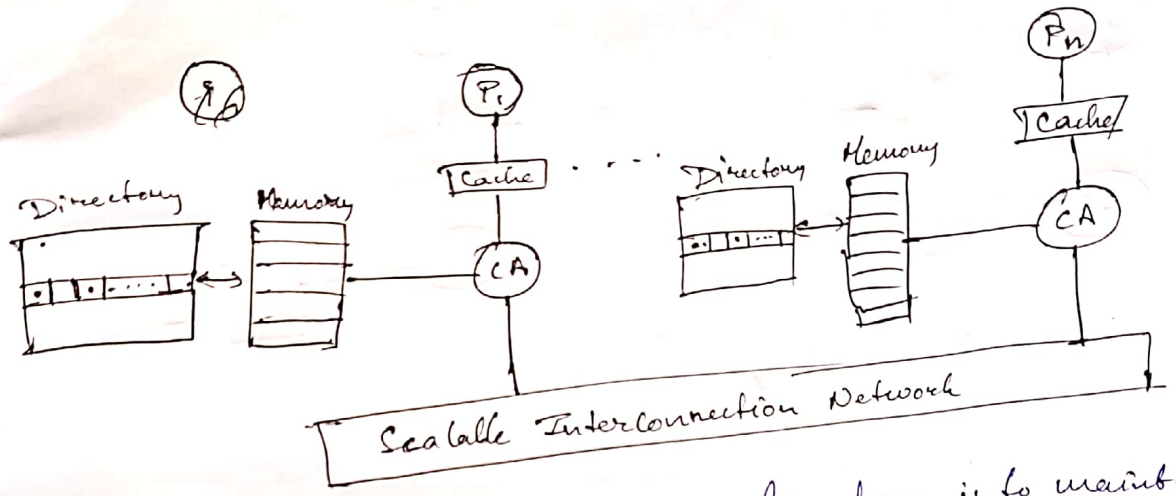
May be the Same

Dirty Node: node that has a copy of the block in its cache in modified ('DIRTY') state.

(Home/ Dirty) **Owner Node:** Node that currently holds the valid copy of a block and must supply the data when needed.

Exclusive Node: Node that has a copy of the ~~data~~ block in its cache in an exclusive state either dirty/or clean/ exclusive as the case may be.

DIRECTORY ORGANISATION



A natural way to organise a directory is to maintain the directory information for a block together with the block in main memory, that is, at the home node for the block. A simple organisation for the directory information for a block is as a bit vector of p-presence bits- which indicates for each of the p- nodes (uniprocessor or multiprocessor) whether that node has a cached copy of the block- together with one or more state bits. Let us assume, for simplicity that there is only one state bit, called the dirty bit, which indicates if the block is dirty in one of the node caches. Of course, if the dirty bit is ON, then only one node (the dirty node) should be caching that block and only that node's presence bit should be ON.