

UNIT III.

INSTRUCTION LEVEL PARALLELISM AND ITS EXPLOITATION

I. ILP Concepts and Challenges:

What is Instruction Level Parallelism (ILP) ?

It is a technique of overlapping the execution of unrelated instructions.

→ This technique increases parallelism by focusing on two factors:

1. Reduce the impact of data and control hazards.
2. Increases processor ability to exploit parallelism.

ILP within a Basic Block:

→ Initial techniques focused on achieving parallelism to a greater extent by exploiting them within a basic block. Before knowing about the exploitation of parallelism within a basic block, let's know **what is a basic block?**

→ It is a straight line code sequence with no branching statements except one entry and one exit point.

Example: for example consider a c code given below:

```
for (i=0; i<n; i++)
```

```
{
```

```
some code
```

```
}
```

Here in this example, some code is a body of the loop which executes iteratively **n** times and each iteration in this condition is considered as one basic block and is executed separately. The major disadvantage of exploiting parallelism within a basic block is that basic block consists of limited number of instructions (3 to 6) and that too there are dependences between those instructions. To overcome this disadvantage technique has come up exploiting parallelism across multiple basic blocks which considers many independent basic blocks as a single basic block and executes the instruction in parallel. One of the well known techniques is **loop- level parallelism**.

Loop Level Parallelism:

- One of the techniques to exploit ILP to a greater extent.
- Exploiting parallelism among the iterations of the loop is called **loop level parallelism**.

As an example consider the following code:

```
for (i=1; i<=1000; i++)
```

```
    x[i]= x[i] + y[i];
```

Here in the above example, each iteration of the loop is independent of the previous iteration and can be executed in parallel. The technique is called as **loop-level parallelism**.

Loop Unrolling

It is a technique of converting loop level parallelism into instruction level parallelism.

- Loop unrolling can be implemented by two techniques:
 1. By the compiler- statically
 2. By the hardware unit- Dynamically (Run-time)

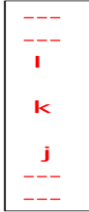
Dependences

When the execution of an instruction depends on the result of previous instructions then the instructions are said to be dependent instructions and the phenomenon is called as **dependences**.

- To exploit ILP we must determine which instructions can be executed in parallel.
- If two instructions are dependent, they cannot be executed in parallel and must be executed in sequence. I.e. instruction level parallelism cannot be exploited in that condition.
- There are three different types of dependences:
 1. Data dependences.
 2. Name dependences.
 3. Control dependences.

Data Dependences

- It is also called as **True Dependence**. To get a clear idea on the data dependences, consider the following set of instructions i, k and j written sequentially one after the other:



- In the above sequence of instructions, an instruction **j** is said to be **data dependent** on instruction **i** for two cases:
 1. Instruction **i** produces a result that may be used by instruction **j**.
 2. Instruction **j** is data dependent on instruction **k** and instruction **k** is data dependent on instruction **i**. This process is called as **chain dependence**. Chain dependence can occur within any number of instructions.
- Impact of data dependence leads to three different hazards : Read After Write (RAW), Write After Read (WAR) and Write After Write (WAW)

Example for data dependence:

```

Loop:      L.D      F0,0(R1)
           ADD.D    F4,F0,F2
           S.D      F4,0(R1)

           DADDIU   R1,R1,-8
           BNE      R1,R2,Loop
  
```

- When any two instructions are data dependent, they cannot be executed in parallel and they have to follow sequential execution.
- Processor having pipeline interlock can cause stalls between the stages whenever there is a data dependence between the instruction at the time of execution.
- Processor without pipeline interlock relies on compiler scheduling technology which will never execute dependent instructions.

Limitations of data dependence: Data dependence between the instructions conveys three things. All the limitations are interdependent to each other and leads to another:

1. Causes hazard
2. Dictates execution order
3. Limits the parallelism.

Techniques to avoid dependence limitations: Basically there are two techniques to avoid data dependence:

1. Maintain dependences but avoid hazards: This technique states that the dependences between the instructions are kept as it is and instructions are rearranged to avoid the hazards occurring due to dependent instructions. The technique is called **code scheduling**. Code scheduling can be done in two ways:

- a. At the compiler level → Also called as **static scheduling** where compiler re arranges the code at the compile time before execution.
- b. At Hardware level → Also called as **dynamic scheduling** where a dedicated hardware is used to rearrange the code during their execution.

2. Eliminate dependences by code transformation technique: This technique eliminates the data dependences between the instructions by transformation of the code. The technique can be applied to the memory operations. To get a better understanding of this technique , consider the simple example shown below:

```
I1: add      r1,r1,#8      [r1 → [r1]+8]
I2: ld       r2,0(r1)      [r2 → mem[0+r1]]
```

In the above code shown, instruction **I1** adds the value of register **r1** to 8 and stores the result back in **r1** and the instruction **I2** loads the value from the memory location which is the result of **r1** in last instruction.

There is a data dependence between instruction **I1** and instruction **I2** were register **r1** is destination in instruction **I1** and register **r1** source in the instruction **I2**. The above data dependence can be eliminated by a simple code transformation technique were instruction **I2** `ld r2,0(r1)` is transformed and written as `ld r2,8(r1)` which is equivalent to `r2 → mem[8+r1]` were instruction **I1** can be eliminated by transforming instruction **I2**. Code can be written as shown below:

```
I: ld      r2,8(r1)
```

Name Dependences:

- Name dependences are the one that occurs when two instructions use same registers or memory location as destination called **Name**.
- Since there is no actual data flow between the instructions, they are not a **true dependence**.
- There are two types of name dependence:
 1. **Antidependence:** Antidependence between the sequence of instructions i and j occurs when instruction j writes a register/ memory location that i reads it.
Example below shows the phenomenon of antidependence

```

DADDIU  R1, R1, -8
        ↓
BNE     R1, R2, Loop

```

In the above example, the instruction DADDIU uses R1 as source and instruction BNE uses R1 as destination register.

2. Output Dependence: The dependence occurs when the sequence of instructions i and j write the value into the same register. I.e. both the instructions uses same destination register. For example, the instructions shown below is an example for output dependence where **r1** is destination register in both the instructions:

I: sub r1,r4,r3

J: add r1,r2,r3

Techniques to overcome Name dependence: There is only one possible technique to overcome name dependence and the technique is called **Renaming** : destination registers or memory locations are given a different names to resolve name dependence.

➔ **Renaming** can be implemented in two ways :

1. **Statically** : which is done by the compiler at the compile time.
2. **Dynamically:** This is done by a dedicated hardware at the time of execution.

Possible Data Hazards:

There are three possible data hazards : Read After Write (RAW), Write After Read(WAR) and Write After Write (WAW).

Note: Please refer notes of the last unit for all the above mentioned hazards.

Control Dependence

Control dependence are the one that determines the ordering of any instruction **i** with respect to branch instruction so that instruction **i** is executed only when it is needed based on the outcome of the branch instruction.

- Below mentioned example gives a clear idea on how it actually appears

```
if p1 {  
  instri;  
};
```

In this example **instr_i** is a control dependent instruction which depends on the outcome of the branch instruction **p1**. **I.e.** instr_i is executed only if the branch instruction **p1** is evaluated to true.

Constraints imposed by control dependence: consider the following set of instructions to get a clear picture on the constraints ruled by branch instructions. Basically imposes two constraints:

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

1. An instruction that is control dependent on branch instruction cannot be moved before branch instruction. In this example instruction **S1** cannot be moved before the branch instruction **p1**.
2. An instruction that is not control dependent on branch instruction cannot be moved after branch instruction. I.e. an independent instruction cannot come within the control of branch instruction.

The above mentioned constraints are very important when scheduling the instructions and the mentioned constraints has to be maintained.

Preserving control dependence by a simple pipeline:

- Ideal case says that it is very much required to follow a strict program order for control dependent instructions in the pipeline execution.
- To preserve control dependence, pipeline follows a simple technique where it ensures that instruction that is control dependent on branch instruction is not executed until the outcome of the branch instruction is known.

Violating control dependence:

- Before knowing exactly the techniques of violating control dependence, it is required to know what is violating control dependence is.
- **Violating Control dependence** is a technique of not preserving the strict program order for the instructions that is control dependent on branch instructions. The technique is implemented in order to achieve parallelism to another extent.
- Control dependence cannot be violated all the time and can be violated only for some cases if the violation of the control dependence does not lead to incorrectness of the program result.
- The rule conveys that to maintain correctness in the program, there are two critical factors: **exception behavior** and **dataflow behavior**. If both of these factors are maintained, then the program correctness is ensured even if the control dependence is not maintained in the program.

Preserving exception behavior:

Preserving exception behavior means reordering of the instruction execution must not lead to any new exceptions in the program.

Example code written below gives a clear idea on this:

```
DADDU    R2,R3,R4
BEQZ     R2,L1
LW       R1,0(R2)
```

L1:

Above written instruction, a memory protection exception (An exception caused when a content from memory location **zero** is read since it is a reserved location) is handled through the

instruction **BEQZ R2, L1** which checks if **R2** value is zero or not and if it is zero, it skips loading of the content from that memory location.

In this case if we ignore control dependence and move instruction **LW R1, 0(R2)** before branch instruction, it leads to an exception **memory protection exception**.

Preserving data flow behavior:

Data flow means actual flow of data values between the instructions that produce results and the one that uses those results

Preserving data flow behavior means when you have the instructions of this kind, their order must be maintained and are not allowed alter the sequence in order to maintain the correctness of the program.

To illustrate this, consider the example shown below:

```
L:      DADDU      R1, R2, R3
        BEQZ      R4, L
        DSUBU     R1, R5, R6
        . . .
        OR        R7, R1, R8
```

In the above example, **OR** instruction appears to be an instruction which is not control dependent on any branch instructions. But the value of **R1** used by **OR** instruction depends on the outcome of the branch instruction **BEQZ** whether it is taken or not taken. I.e. it will have different values based on the outcome.

In this condition data flow occurs for an instruction **OR** based on the branch instruction outcome. Hence if you move even the instruction **OR** before the branch instruction, it may lead to program incorrectness and in this case data flow behavior needs to be preserved.

II. Basic Compiler Techniques for Exposing ILP

Basic Pipeline scheduling and Loop unrolling:

In the first outset we will discuss about pipeline scheduling and then with the knowledge of pipeline scheduling, we will discuss the technique of loop unrolling.

Pipeline Scheduling:

- Pipeline Scheduling is one of the techniques to overcome the pipeline stalls occurring due to the instruction dependency.
- This technique separates the dependent instruction from a source instruction by a distance that is equal to pipeline latency of source instruction so as to avoid stall(s) in the pipeline.

Some terms:

Source instruction: An instruction that generates the result to be used by subsequent instructions.

Dependent instruction: An instruction that is dependent on prior instruction's result for the source operand.

Latency: It is an execution time taken by a particular functional unit after entering in to execution stage in a pipeline. (Basically delay time in producing the result).

Below table shows the different latency values as an example when there is a dependency between two different instructions.

Instruction producing result	Instruction using result	Latency
FP ALU op	FP ALU op	3
FP ALU op	SD	2
LD	FP ALU op	1
LD	SD	0
Int ALU op	Int ALU op	0

Basic Pipeline Scheduling (An Example):

We will consider an example to know how exactly the basic pipeline scheduling technique is implemented. As an example let us consider the 'c' code written below.

```
for(i=4; i>0; i=i-1)
```

```
  x[i]=x[i] + s;
```

Given the intervening Latency assumptions:

Instruction producing result	Instruction using result	Latency
FPALU op	FPALU op	3
FPALU op	SD	2
LD	FPALU op	1
LD	SD	0
IntALU op	IntALU op	0

Note: By default there will be one clock cycle delay for the branch instruction.

‘C’ code written above first access the last element of the array and then for every iteration it decrements the value of counter to point to the next element in the array in a decrementing order. In every iteration it reads the value from the location of the array adds a constant value *s* with it and then stores the value back in the same accessed location.

Before carrying out pipeline scheduling, when the code is given in the above type of format, the very first step is we need to realize the code in the MIPS format.

Step 1: Equivalent MIPS code:

- The code given above access the last location in the beginning, I.e. the fourth location of the array and MIPS follows the Byte addressability form were in every memory location is treated as 1 byte (8 bits). That means to access location number 4 in MIPS byte addressability form value 32 needs to be used. Likewise multiples of 8 is used to access each and every memory location in MIPS.
- Assume register **R1** has the value 32 in it. i.e. address the location 4 in array. Also assume that *s* is a floating point constant value and is stored in register **F2**.
- **Note:** MIPS has 32 general purpose registers out of which 16 registers are used to operate on integer values and are named with odd series with prefix **R** (**R1,R3....R31**) and another 16 registers are used to operate on floating point values and are named with even series of numbers with the prefix **F** (**F0,F2....F30**).
- Equivalent MIPS code for the instruction is written as shown below:

```

Loop: LD    F0, 0(R1)      ;F0=array el.

      ADDD F4,F0,F2      ;add scalar in F2

      SD    F4,0(R1)      ;store result

      SUBI R1,R1,#8       ;decrement pointer

      BNEZ R1, Loop      ;branch
  
```

Note: suffix D in the instruction suggest floating point (double) operation and suffix I suggest integer operations.

Step 2: Pipelined execution considering the intervening latency:

In step 2, dependent instructions are identified and their latency values are analyzed and stalls are inserted between the pipeline executions.

- | | |
|-----------------|------------------------|
| 1. Loop: | LD F0, 0(R1) |
| 2. | Stall |
| 3. | ADDD F4,F0,F2 |
| 4. | Stall |
| 5. | Stall |
| 6. | SD 0(R1),F4 |
| 7. | SUBI R1,R1,#8 |
| 8. | Stall |
| 9. | BNEZ R1, Loop |
| 10. | Stall |

Stalls in the execution are being inserted according the latency values mentioned in the table. Here all the data dependent instructions are identified and accordingly stalls are introduced between them to avoid the hazards occurring the pipeline. Here each stall indicates a no operation in that particular clock cycle. As an example you can see that there is a stall in cycle 2. It means it is a waste cycle.

As you can see that, execution of a single iteration consumes 10 clock cycles for 5 instructions out of which there are 5 stalls. So next step tries to re arrange the code to minimize the stall occurring which is called as **code scheduling**.

Step 3: Code scheduling

This is the last step, where the independent instructions are identified and are reordered so as to minimize the stalls in the program. While reordering, independent instructions are inserted in the stall area so as to maintain the latencies between the dependent instructions.

Steps:

- In step2 we have identified the dependent instructions and inserted stalls wherever necessary.
- In this step we need to identify the independent instruction that can be moved into the stall area.
- Just refer back to the step2, in clock cycle 7 there is an independent instruction and there is a very first stall in clock cycle 2. So move an independent instruction from clock cycle 7 into clock cycle 2 which minimizes one stall as shown below

1. Loop:	LD	F0, 0(R1)
2.	SUBI	R1, R1, #8
3.	ADDD	F4, F0, F2
4.	Stall	
5.	Stall	
6.	SD	F4, 8(R1)
7.	BNEZ	R1, Loop
8.	Stall	

- Next we need to check whether the stalls can be eliminated still. There is a stall occurred due to branch delay and we know that there are techniques three different techniques to overcome that out of which **from before** is a best one where in independent instruction before the branch instruction is moved into the delay slot.
- Observe that there is a **store** instruction in clock cycle 6 which is dependent on the instruction present in clock cycle 3. So the dependent instruction has to be separated from source instruction at least by 2 clock cycles due to which stalls are being introduced. It means that at least 2 clock cycles has to be there but it can be more than that also.
- With the above conclusion we can move the instruction from clock cycle 6 into clock cycle 8 in the delay slot by which we can reduce a stall in clock cycle 5 and also eliminates a stall in clock cycle 8 as shown below

1. Loop: LD F0, 0(R1)
2. SUBI R1,R1,#8
3. ADDDF4,F0,F2
4. Stall
5. BNEZ R1, Loop
6. SD F4,8(R1)

The final scheduled code takes only six clock cycles to execute a single iteration of the loop were in 4 stall are being reduced by **code scheduling**.

Note: Intermediate steps are written for you to understand the technique. No need to produce the same when it comes to exam.

Loop Unrolling:

- Code scheduling technique which we have seen so far aims on increasing ILP within a basic block were in a single basic block is considered at a time and parallelism is achieved on independent instructions. If we consider the instructions which we have used for code scheduling, in that each iteration is treated as one basic block and ILP is achieved within each iteration.
- Better performances can be achieved if ILP is done across the basic block were in two or more basic blocks are considered as one basic block and executed in parallel.
- Loop Unrolling is one of the techniques to increase ILP across the basic block. The technique replicates the copy of loop body multiple times so as to increase ILP across different iterations of the loop.

Some concepts of loop unrolling to be known:

- There is a maximum limit imposed for the replication number of the loop body. That is if there are **n** iterations, all **n** iterations cannot be replicated for two simple reasons:
 1. **Availability of the registers:** There is a limit for the number of available registers and compiler replicates loop body until the registers are available.
 2. **Cache overhead:** Finally any instruction under execution has to be brought to the cache. More the number of unrolled iterations more is the code size and needs more space in cache which causes cache overhead.

- While unrolling, basically replicates the iterations based on the multiples of the total iterations. For example consider a particular loop unrolling technique can replicate 4 copies at a time and there are 6 iterations to be executed in total. At that time it will unroll 3 copies at a time which is a multiple of 6 which will execute unrolled loop two times.

We will consider the same example considered for code scheduling to see how loop unrolling technique increases ILP for the body of loop shown below realized in MIPS.

Assume that loop unrolling technique creates four copies of the loop body, considering the number of loop iterations is also equal to 4.

```
Loop: LD    F0, 0(R1)
      ADDDF4, F0, F2
      SD     F4, 0(R1)
      SUBI   R1, R1, #8
      BNEZ   R1, Loop
```

Before starting with the implementation of loop unrolling technique, first create the number copies of the body of the loop required. Here, four iterations needs to be executed and unrolling technique can create four copies at a time. So create 4 copies of the loop body. This is called as **unrolled loop**.

```
Loop: LD    F0, 0(R1)
      ADDDF4, F0, F2
      SD     F4, 0(R1)
      SUBI   R1, R1, #8
      BNEZ   R1, LOOP
      LD     F0, 0(R1)
      ADDDF4, F0, F2
      SD     F4, 0(R1)
      SUBI   R1, R1, #8
      BNEZ   R1, LOOP
      LD     F0, 0(R1)
      ADDDF4, F0, F2
      SD     F4, 0(R1)
      SUBI   R1, R1, #8
      BNEZ   R1, LOOP
      LD     F0, 0(R1)
      ADDDF4, F0, F2
      SD     F4, 0(R1)
      SUBI   R1, R1, #8
      BNEZ   R1, LOOP
```

We know that when a number of unrolled copies is formed, those unrolled instructions has to be executed mandatorily. In this example four copies are created and all four copies has to be executed compulsorily. But in this unrolled loop, you can see that there are branch instructions for after every copy of the loop body which is an unnecessary branch instruction or a **redundant branch** instruction. So at the first out set those redundant branch instructions has to be eliminated keeping only one branch instruction at the end for the termination purpose. After eliminating that instruction, following is the code shown:

```
LD    F0, 0(R1)
      ADDDF4, F0, F2
      SD    F4, 0(R1)
      SUBI  R1, R1, #8
      LD    F0, 0(R1)
      ADDDF4, F0, F2
      SD    F4, 0(R1)
      SUBI  R1, R1, #8
      LD    F0, 0(R1)
      ADDDF4, F0, F2
      SD    F4, 0(R1)
      SUBI  R1, R1, #8
      LD    F0, 0(R1)
      ADDDF4, F0, F2
      SD    F4, 0(R1)
      SUBI  R1, R1, #8
      BNEZ R1, LOOP    //only one branch termination instruction
```

Note: The above step can be directly written while unrolling the copies. Just for the understanding purpose it is been illustrated over here.

Next we will carry on the loop unrolling technique according to the steps in a sequence. There are three different steps:

Step 1: Eliminating Data dependences:

In this step, data dependences between the different iterations of loop are eliminated by a **code transformation** technique which is being discussed in one of the previous section. Along with

the elimination of dependences required changes has to be made for the code that is affected by the transformation.

- As you can see that there is data dependence between **SUBI** instruction of the previous iteration and **LD** instruction of the next iteration in all four copies as shown below.

```

Loop:  LD    F0,0(R1)
       ADDD  F4,F0,F2
       SD    F4,0(R1)
       SUBI  R1,R1,#8
       LD    F0,0(R1)
       ADDD  F4,F0,F2
       SD    F4,0(R1)
       SUBI  R1,R1,#8
       LD    F0,0(R1)
       ADDD  F4,F0,F2
       SD    F4,0(R1)
       SUBI  R1,R1,#8
       LD    F0,0(R1)
       ADDD  F4,F0,F2
       SD    F4,0(R1)
       SUBI  R1,R1,#8
       BNEZ  R1,Loop
  
```

- In the second iteration, **R1** value is decreased by 8 bits and that decreased value is used to access the next memory location. Instead of writing two different instructions for that, it can be directly written as:

LD F0, -8(R1) which is equivalent to

```

SUBI  R1,R1,#8
LD    F0,0(R1)
  
```

this eliminates the data dependence between the instruction by ruling out the instruction **SUBI**. But value of **R1** remains with initial value 32 and to access the location address 16 for the second iteration, **R1** has to be subtracted by 16, then 24 and so on. Finally for the last iteration **R1** is made zero by subtracting it directly by 32 instead of subtracting **R1** four times by 8. So this step eliminates the data dependency and even the loop termination statement has to be adjusted.

- Below shown is the code after eliminating the data dependence.

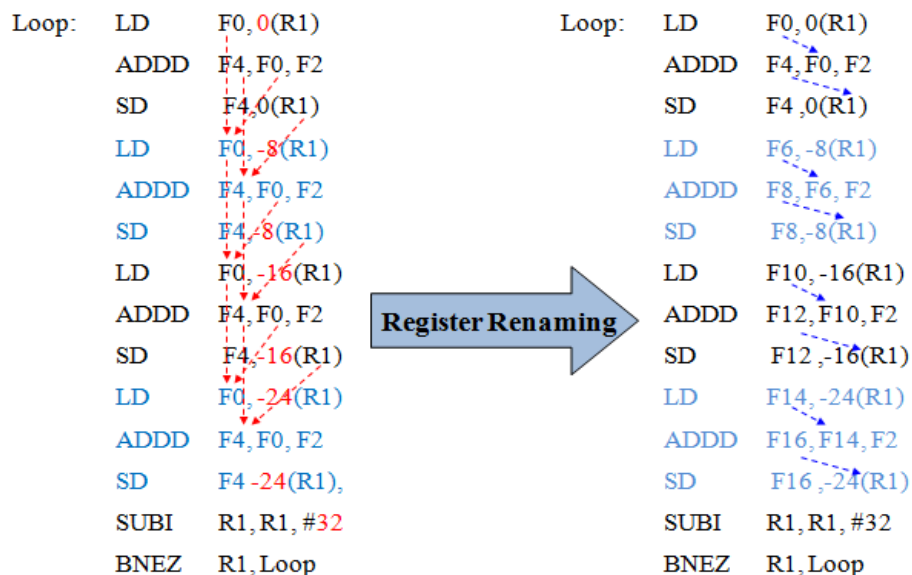
```

Loop: LD    F0, 0(R1)
      ADDD  F4, F0, F2
      SD    F4, 0(R1)
      LD    F0, -8(R1)
      ADDD  F4, F0, F2
      SD    F4, -8(R1)
      LD    F0, -16(R1)
      ADDD  F4, F0, F2
      SD    F4, -16(R1)
      LD    F0, -24(R1)
      ADDD  F4, F0, F2
      SD    F4, -24(R1),
      SUBI  R1, R1, #32
      BNEZ  R1, Loop

```

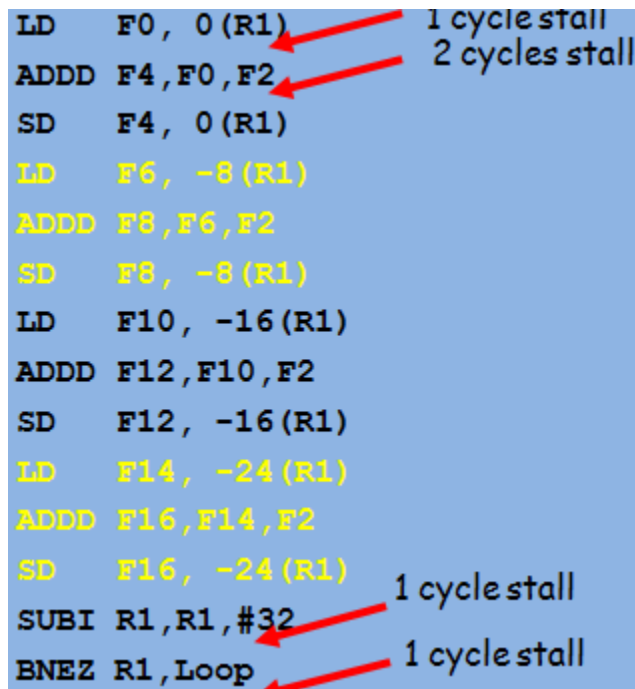
Step 2: Eliminating Name dependences:

Code that is got in the step 1 contains same register names for the same operations used in different iterations I.e. load, add and store this is called **Name dependence**. Using the same register names doesn't allow the reordering of the instructions as it may produce different results. So this step aims in the elimination of the name dependences by simple register renaming technique where in different registers are used for same operations of the different iterations. After using different register names, code is written as shown below:



Left side is the instructions before eliminating name dependence.

Next before moving into step3 we will find out the dependent instructions and insert the stalls between them. Let us consider the same latency value that is being considered for the last example.



```

LD    F0, 0(R1)
ADDD  F4, F0, F2
SD    F4, 0(R1)
LD    F6, -8(R1)
ADDD  F8, F6, F2
SD    F8, -8(R1)
LD    F10, -16(R1)
ADDD  F12, F10, F2
SD    F12, -16(R1)
LD    F14, -24(R1)
ADDD  F16, F14, F2
SD    F16, -24(R1)
SUBI  R1, R1, #32
BNEZ  R1, Loop
  
```

Stalls indicated in the diagram:

- 1 cycle stall between LD F0, 0(R1) and ADDD F4, F0, F2
- 2 cycles stall between ADDD F4, F0, F2 and SD F4, 0(R1)
- 1 cycle stall between SD F16, -24(R1) and SUBI R1, R1, #32
- 1 cycle stall between SUBI R1, R1, #32 and BNEZ R1, Loop

As shown above, every copy of unrolled loop has 3 which mean for 4 iterations altogether there 12 stalls are being inserted. Then there are 2 extra stalls, one due to delayed branch and the other due to dependency of branch instruction on the previous instruction. Totally for 4 iterations of the unrolled loop there are 14 stalls which kill 14 clocks of execution and there are 14 instructions to be executed which again require 14 clocks. Altogether for 4 iterations it requires 28 clock cycles which means that for a single iteration, it requires 7 clock cycles which is even more than basic scheduling. So these instructions need to be scheduled in order to reduce the number of stalls. The next step is **scheduling** unrolled loop

Step 3: Scheduling unrolled loop:

The last step which re orders the instruction to reduce the stalls caused by dependent instructions. We know that in order to carry out loop unrolling iterations has to independent of other iterations. Scheduling mechanism follows a very simple technique and is done in two steps: first step combines all the similar operations of different instruction in a sequence and the second step does the fine reordering to eliminate stalls.

In this program, every iteration has three operations in a sequence: Load, Add and Store. Operations of different iterations are combined together in a sequence as first Load, second Add and finally Store operation as shown below:

```
Loop: LD    F0,0(R1)
      LD    F6,-8(R1)
      LD    F10,-16(R1)
      LD    F14,-24(R1)
      ADDD  F4,F0,F2
      ADDD  F8,F6,F2
      ADDD  F12,F10,F2
      ADDD  F16,F14,F2
      SD    F4,0(R1)
      SD    F8,-8(R1)
      SD    F12,-16(R1)
      SD    F16,-24(R1)
      DSUBI R1,R1,#32
      stall
      BNEZ R1,Loop
      stall
```

In the above instructions the stalls within the loop is eliminated by combining the similar instructions load, add and store of different iterations together. Now the instruction as only two stalls as shown in the above code.

Stall in a delay slot due to branch instruction can be eliminated by delayed branching method were in **from before** is used to eliminate stall. Instruction **SD F16, -24(R1)** is moved under delay slot and the instruction gets transformed into **SD F16, 8(R1)** and another instruction **SD F12, -16(R1)** is moved after instruction **DSUBI R1,R1,#32** which is transformed as **SD F12, 16(R1)**. The final scheduled code is shown below:

```
Loop: LD    F0,0(R1)
      LD    F6,-8(R1)
      LD    F10,-16(R1)
      LD    F14,-24(R1)
      ADDD  F4,F0,F2
      ADDD  F8,F6,F2
      ADDD  F12,F10,F2
      ADDD  F16,F14,F2
```

```
SD    F4,0(R1)
SD    F8,-8(R1)
DSUBI R1,R1,#32
SD    F12,16(R1)
BNEZ R1,Loop
SD    F16,8(R1)
```

Steps performed by the compiler for loop unrolling:

1. First determine whether the loop iterations are independent.
2. Rename the registers used for different iterations → name dependency
3. Eliminate branch instructions and adjust loop termination at the end
4. Determine loads and stores in unrolled loop can be interchanged
→ requires analyzing memory addresses and finding that they do not refer to the same address.
5. Schedule the code, preserving any dependences needed to yield same result as the original code

III. Reducing Branch Costs with Prediction

To minimize the performance loss occurring due to branch instructions, based on the behavior of branch instructions, branch instructions can be predicted if it is taken or not taken. This method is called as **Branch Prediction**.

→ Branch prediction can be done in two methods:

1. Static branch prediction.
2. Dynamic branch prediction.

Static Branch Prediction:

Branch prediction done at the compile time by the compiler is called as **static branch prediction**. Here compiler is the one responsible for carrying out the prediction based on its behavior.

Static branch prediction can be done in two ways:

1. **Simplest scheme:** It is a fixed prediction scheme where prediction method is fixed either as **taken** or **not taken** and the every time prediction is done according to that.

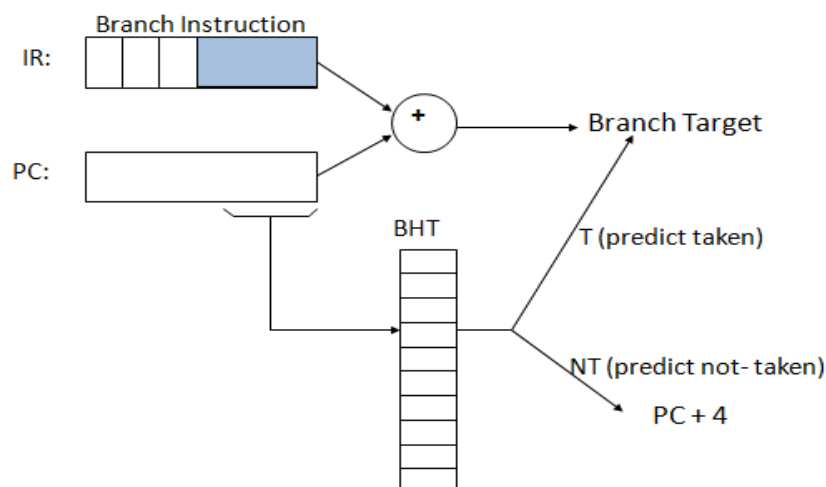
2. **Accurate technique:** This technique is a better technique and gives accurate result when compared to the technique discussed previously. Here prediction is done based on the profile information of the branch instruction. It uses 2 techniques:

- In one of the technique, compiler has a fixed behavior on branch code and it remains fixed from the beginning of the time and prediction is done accordingly every time.
- In another technique, compiler maintains the history of last runs for a particular branch code and based on that outcome prediction is carried out.

Dynamic Branch Prediction:

Branch prediction done at the execution time by the usage of a dedicated hardware is called as **Dynamic Branch Prediction** technique.

- To implement Dynamic branch prediction, technique uses a dedicated hardware called **Branch Prediction Buffer** or **Branch History Table**.
- A Branch History Table (BHT) is a small memory element indexed by lower portion of address of branch instruction.
for example: if branch instruction has 4 bits address, then in that lower 2 bits are used as index for the Branch History Table.
→Memory part contains information in terms of bits which says whether a branch was recently taken or not taken. Basically maintains the recent history of the branch instructions and prediction is done according to its outcome.
→Anytime if the prediction goes wrong, prediction bit is inverted and stored back with a updated information.
- Typical hardware structure for Branch history Table is shown below:



- The hardware scheme consists of the units Instruction Register (IR), Program Counter (PC) and Branch History Table (BHT).

- Whenever Instruction Register loads branch instruction, lower address of the branch instruction is used to point to the index of Branch History Table.
- Based on the information stored in the index, branch prediction is carried out.
- If branch history is taken it will fetch the Branch Target address and if branch history is not taken, program counter is incremented to fetch next sequential instruction.

Branch Prediction schemes:

There are four different branch prediction schemes:

1. One bit predictor
2. Two bit predictor
3. Correlating predictor
4. Tournament predictor

1. Branch predictor using BHT of one bit predictors:

→Hardware scheme for this technique is very similar to the one discussed just now.

→BHT consists of two parts: Index part and memory part. Index uses the lower portion of PC for the branch instruction. Memory part contains the history of branch instruction stored in terms of one bit. I.e. 0 or 1. When it is 0 branch is predicted as Not Taken and when it is 1 branch is predicted as Taken.

→If the prediction goes wrong anytime, the value is inverted. i.e. 0 will become 1 and 1 will become 0.

→To get a better understanding of this technique, consider the example shown below:

```
for ( i=0; i<100; i++ )
```

```
for ( j=0; j<3; j++ )
```

```
// whatever code
```

Here we will check how well the prediction scheme is carried out for the inner loop shown in the example.

iterations	0	1	2	3	0	1	2	3	0	1	2	3
State/prediction	N*	T	T	T*	N*	T	T	T*	N*	T	T	T*
Outcome	T	T	T	N	T	T	T	N	T	T	T	N

→The table consists of 2 rows where the first row consists of prediction and the second row consists of the actual outcome of the branch instruction. Iteration of inner loop are plotted in different columns.

→* Symbol indicates the wrong prediction. As shown in the table, whenever there is a wrong prediction, stored value in the history table is inverted immediately i.e. T becomes N and N becomes T.

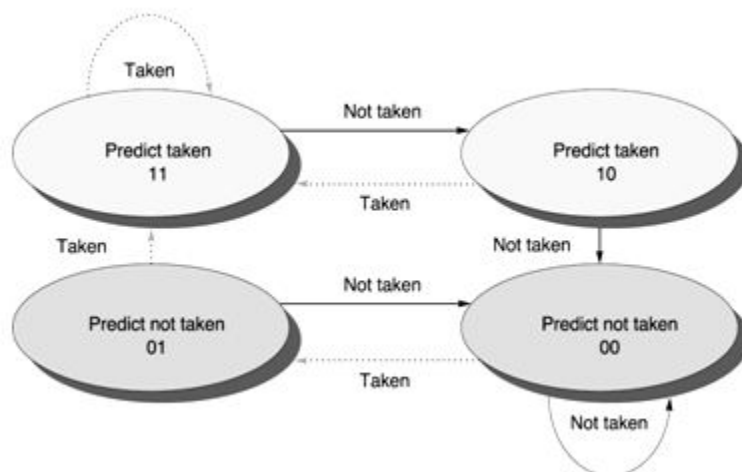
→There are 2 wrong predictions for every iteration as the scheme changes its mind too quickly.

2. Branch Prediction using BHT of two bit predictors:

→This technique is similar to the one discussed in the previous section with only one difference, memory part of the Branch History Table used 2 bits to store information. There will be 4 different values from 0 to 3 in which value 3 and 2 are used for representing branch as taken and the other two values are used for representing branch as not taken.

→This technique overcomes the disadvantage of 1 bit predictor where prediction must miss continuously twice before the value is changed.

→To know about the working of this scheme, consider the state transition diagram plotted below:



→As you can see that values 11 and 10 is reserved for taken and values 00 and 01 is reserved for not taken. As shown when the prediction misses two times, only then the state is changed.

→We will consider the same example considered before and let us check if this scheme leads to a better prediction or not.

iterations	0	1	2	0	1	2	0	1	2	0	1	2
State/Prediction	N*	n*	T	T*	t	T	T	T*	t	T	T	T*
Outcome	T	T	T	N	T	T	T	N	T	T	T	N

→When the prediction misses two times, only at that time the value of the history table is changed.

As you can see from the above table, technique leads to better prediction compared to 1 bit prediction scheme, where there is only one wrong prediction for each iteration.

3. Correlated Prediction scheme:

Branch predictors that use the behavior of other branches to make prediction are called as **correlated predictors**.

→These predictors are also called as two level predictors. Prediction scheme is very useful when a particular branch instruction depends on the outcome of previous branch instructions as show below in the example:

```
if (d ==0 )
    d =1;
    if (d == 1) {
```

In the above example, the second branch instruction outcome depends on the outcome of the previous branch instruction. I.e. if first branch instruction is taken then the second branch instruction will be taken. In these cases the correlated predictors are the better prediction scheme.

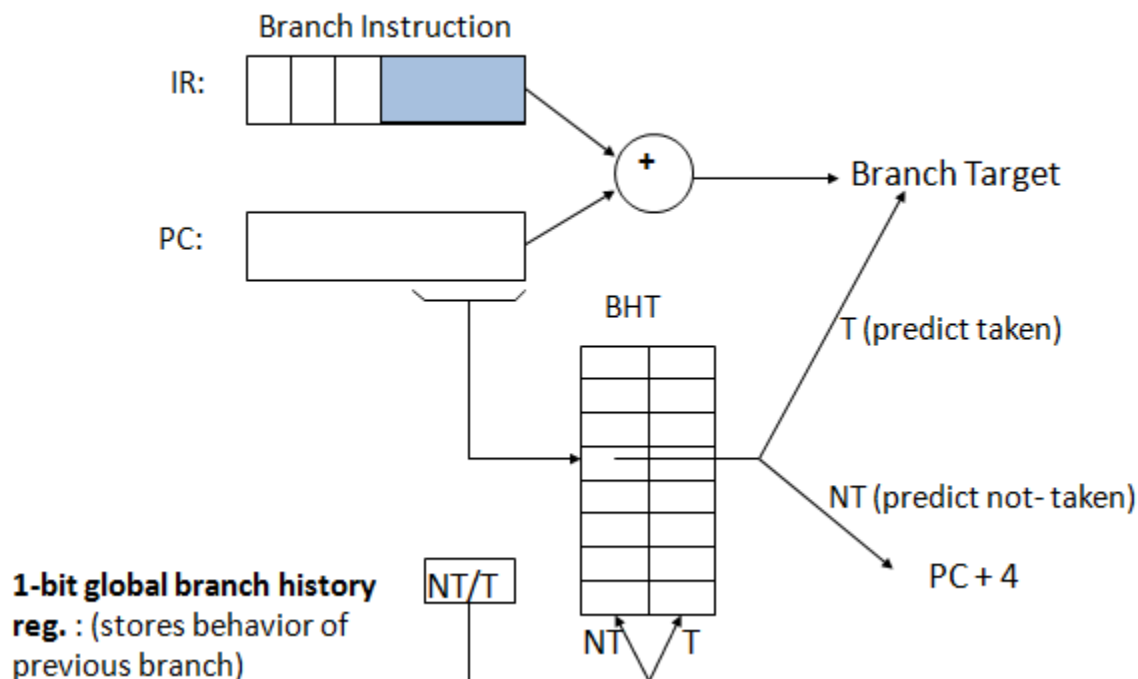
Hardware scheme of Correlated predictors:

Hardware scheme of correlated predictors is almost similar to basic hardware structure of branch history table with only one difference. Correlated predictors have an additional unit called as **global branch history register**. This register stores the information regarding the behavior of previous branches. For example if the register is a one bit register, it stores the behavior of

previous one branch. If it is 2 bit, then it stores the behavior of previous two branches and likewise.

Apart from this additional unit, rest all the units are present and work in the similar way as discussed for basic branch history table.

Typical hardware structure for Correlated predictor is shown below. The hardware scheme uses 1 bit global branch history register which stores the information of previous one branch:



→ In the hardware scheme of the correlated predictor, BHT has an index part and memory part. In this scheme, index part of BHT is the combination of Branch history registers and lower portion address of the branch instruction

For example: If there are 4 bits in a branch instruction and if there is 1 bit in a branch history register, then the index value of the Branch History Table will be 3 bits which is formed by the concatenation of 2 bits lower address of branch instruction and 1 bit value of Branch history register.

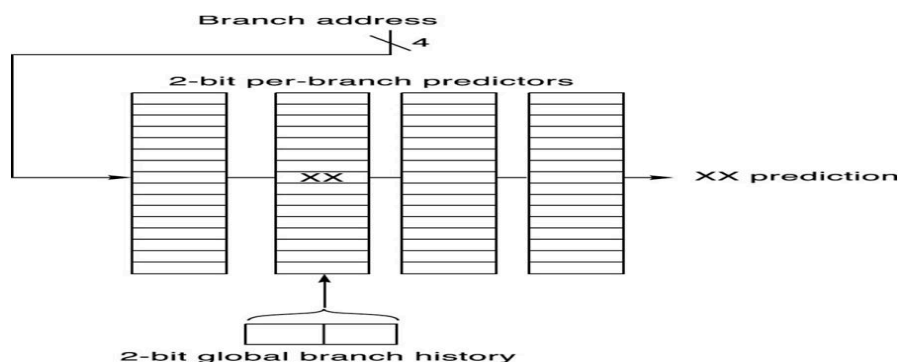
→ As it is shown in the above figure, every index of BHT has 2 columns and these columns are called as **DIRection predictor entries**. These entries are dependent on the value of branch history register. If there are m bits in branch history register, then there will be 2^m **DIRection Predictor (DIRP)** entries. In the above hardware scheme as you can see that there are 2 **DIRection predictor entries** which is formed by using 1 bit branch history register which gives $2^1 = 2$ **DIRP**. Similarly 2 bits will have $2^2 = 4$ entries and so on.

→When it is 1 bit Branch History Register it will have 2 values ranging from 0 to 1 in binary. When there are 2 bits in branch history register, it will have 4 values ranging from 0 to 3 in binary.

→Then the prediction scheme is similar to the last technique where in if the prediction is **taken**, it will fetch BTA and if it is **not taken**, it will fetch sequential instruction.

→Correlated prediction scheme is denoted by **(m, n)** scheme which means that scheme uses the behavior of last m branches in Branch History Register and the memory part of BHT uses n bit prediction scheme to maintain the information(n=1 or 2).

→To get a clear picture on the prediction scheme, consider a prediction scheme A(2,2) which means that there are behavior of last 2 branches and the scheme stores the information in 2 bit value. In this condition, there will be $2^m = 2^2 = 4$ DIRP entries and each entry is having the information stored in 2 bits. Assume that lower portion of branch address is 4 bits. The arrangement of the hardware is done as shown below:



→To illustrate the working of correlated prediction scheme, consider the same example that is discussed previously. Assume that the scheme uses 1 bit prediction and the Branch history register stores 2 bit information. I.e. it maintains the behavior of last two branches.

→m value is equal to 2 and there are $2^2 = 4$ DIRP entries in the BHT which has the information of 1 bit. DIRP values for different history will be equal to NN(not taken, not taken), NT(Not Take, Taken), TN(Taken, Not taken) and TT(Taken,Taken). All these DIRP entries are maintained in 4 different rows and outcome of last two branches are initially considered as NN by default since it is executing for the first time.

→Consider the same example to look into the working of prediction scheme and prediction is plotted for the inner loop as done before.

```
for ( i=0; i<100; i++ )
  for ( j=0; j<3; j++ ) // whatever code
```

		iterations											
		0	1	2	0	1	2	0	1	2	0	1	2
State/prediction	BHR=NN	N*	T	T	T	T	T	T	T	T	T	T	T
"active pattern"	BHR=NT	N	N*	T	T	T	T	T	T	T	T	T	T
	BHR=TN	N	N	N	N	N*	T	T	T	T	T	T	T
	BHR=TT	N	N	N*	T*	N	N	N*	T*	N	N	N*	T*
Outcome	N N	T	T	T	N	T	T	T	N	T	T	T	N

In the above table, initially all entries are made as not taken by default.

→Note: Please refer slides for the same example which is plotted using 3 bit BHR entries.

Calculating size of Branch History Table:

There is a simple formula to calculate the total size of a branch history table used in the correlated prediction scheme.

For a (m,n) predictor, there are:

- 2^m predictors for each branch
- n bits in each predictors
- N entries in BHT corresponding to N branches

Then Size of BHT is given by the formula:

$$\text{Size of BHT} = 2^m \times n \times \text{Number of entries in the BHT}$$

- Example – Assuming 8K bits in the BHT, how many entries are possible for the following predictors:

Predictor	No. of Entries
(0,1)	8K entries
(0,2)	4K entries
(2,2)	1K entries
(12,2)	1 entry

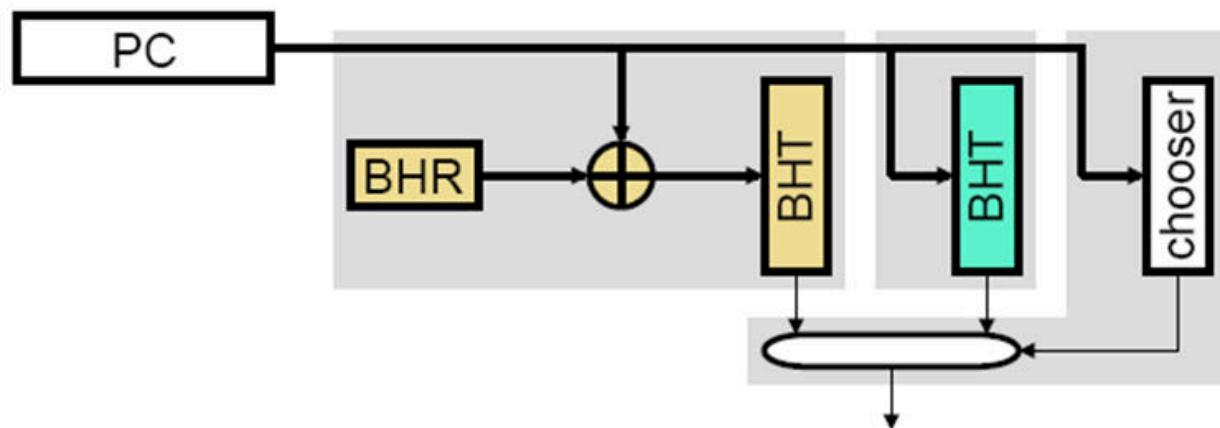
Note: Go through the problem and try to apply the values in the formula. Please revert back if any doubts are there.

4. Hybrid (Tournament) Predictor:

The Prediction scheme combines the idea of simple BHT predictor and correlated predictor scheme to get a better prediction over the branches as the correlated predictors would not be accurate for the independent branches.

→Hardware of this scheme has both the prediction schemes combined and when there is a independent branch, it uses simple BHT and when there is dependent branches that needs history, it uses correlated predictor.

→ Hardware configuration of the scheme is shown below:



→The configuration has two BHTs, first BHT is used for correlated predictors and the second BHT is used for simple BHT predictor.

→ Whenever there is a Branch instruction under execution both scheme works and produces the result.

→ There is one more memory H/W unit called chooser which maintains the information for branch on which technique to choose. Whether correlated or simple scheme. Even this has index and memory part where index is the lower portion address of branch instruction.

→ There is a Multiplexer to choose one out of two outcomes produced by BHT based on the information produced by chooser H/W.

IV. Reducing Data Hazard with dynamic scheduling

It is a technique where hardware rearranges the instruction execution to reduce the stalls while maintaining data flow and exception behavior.

→ Technique is called dynamic because code scheduling is done at **execution time** by a dedicated H/W unit.

→ The dynamic scheduling technique is proved to be a better technique when compared to the static scheduling technique because of two major advantages in it:

1. Handling cases when dependences are unknown at compile time.
2. Helps to overcome the delays happening due to compiler overhead.

Dynamic Scheduling: The Idea

→ One of the major limitations in simple pipeline execution is that it uses **in-order issue** and **in-order execution** of the instructions. I.e. instructions are executed as they are ordered during the compile time. The main problem of this technique is that it avoids execution of other instructions when there is a stall.

→ For example, consider the following set of instructions:

DIVD F0,F2,F4

ADDD F10,F0,F8

SUBD F12,F8,F14

In the above instructions, there is a data dependency for the ADDD instruction. Because of this even the independent instruction SUBD is also stalled in pipeline.

→ To overcome this disadvantage, dynamic scheduling uses in-order instruction issue and out-of-order instruction execution as soon as data operands are available. I.e. no instructions are halted if there is no dependency between them.

→ However the WAR and WAW hazards are being handled by the technique of **register renaming**.

→ To facilitate out-of-order execution, ID stage is being split up into two stages:

1. Issue: This stage decodes the instruction and checks for the structural hazard
2. Read operands: This stage waits until there is no data hazard (Dependency) and then reads the operands.

Default features of dynamic scheduling Hardware:

1. Fetches into an instruction queue and from there it is sent to the execution units.

2. There has to be multiple functional units to allow parallel execution.
3. Passes instruction issue in- order and execution in out-of-order.

There are different techniques to carry out-of-order execution. One of them is Tomasulo's Algorithm (Basic Tomasulo's Hardware).

V. DYNAMIC SCHEDULING USING TOMASULO'S APPROACH

→ It is a dynamic scheduling algorithm which is being named after Robert Tomasulo.

→ Scheme is used for IBM 360/91 Floating Point Unit to allow out-of-order execution which was invented even before invention of Cache systems.

→ Algorithm tracks when operands are available to minimize RAW hazards and to eliminate WAR and WAW hazards it introduces **register renaming**.

→ The main goal behind the usage of this algorithm is to achieve high FP performance without special compilers.

Features of the Architecture that motivated dynamic scheduling:

1. 360 architecture had only 4 FP registers which prevented compiler scheduling.
2. Long memory access and Long FP delay which made it carry out-of-order execution.
3. Have a multiple functional units for doing the operation in parallel.

Reservation station in Tomasulo's Algorithm:

→ Tomasulo's algorithm uses a special buffer called reservation station for each and every functional unit present in the Hardware.

→ Reservation station is the one that buffers the operands of instructions waiting to be issued if available and if not available waits until the operands become available.

→ As soon as the result is available and if it is required, it directly fetches the operands eliminating the need to get it from registers.

→ Reservation station also provides the functionality of **register renaming**, where destination register in the instruction is replaced by the name of reservation station.

Advantages of Reservation station usage rather than Register files:

1. Does the functionality of hazard detection and execution.
2. Results are passed directly to functional unit from the reservation station.

Reservation Station Structure:

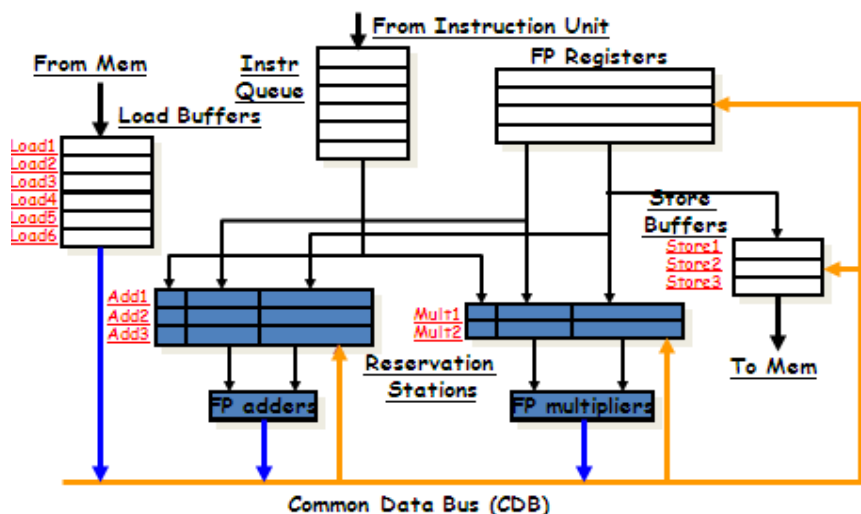
<i>Busy</i>	<i>Op</i>	<i>V_j</i>	<i>V_k</i>	<i>Q_j</i>	<i>Q_k</i>

Typical FP reservation station has following fields:

1. **Busy:** indicates if reservation station (Functional unit) is available or not.
2. **Op:** Op indicates the type of operation done by the functional unit (add, sub, mul and div). This field is not there in **load** and **store** buffers
3. **V_j, V_k:** Indicates value of source operand if it is readily available. For load operation, there is only one field **V** and for store there is no **V** field.
4. **Q_j, Q_k :** Indicates the name of the reservation station producing the source operand. If it is already available, this field will be empty. **Store buffer has only q field.**

Register result status—Indicates which functional unit will write each register, if one exists. Blank when no pending instructions that will write that register.

Tomasulo-based FPU for MIPS (Basic Tomasulo's Hardware)



The hardware consists of multiple **floating point** functional units (adders and multipliers) and multiple **load** and **store** functional units. Functional unit adders are responsible for doing addition and subtraction operation and multiplier unit is responsible for doing multiplication and division operation.

→ Each functional unit has a separate reservation stations and load and store reservation stations are called as load and store buffers.

→ Bypassing of the result into different units is facilitated by a common bus called Common Data Bus (CDB) which has connection to all reservation station and general purpose registers.

Execution in Tomasulo's algorithm is split up into three different stages:

1. **Issue:** Get the instruction from the instruction queue and if the reservation station is free issue instruction into the reservation station along with operand values.
 → If operand values are not readily available, then keep track of the functional unit (Reservation station) that is processing them.
 → If Reservation station itself is not free, then wait until it becomes free and then issue the instruction.
2. **Execute:** If both operands are ready in the reservation station, then execute the instruction. If not ready watch for the common data bus that produces the result and if it is applicable fetch the operand value.
3. **Write result:** After finishing the execution, write the result on a common data bus and send it to the required units. Once writing the result over a common data bus, mark the reservation station as available.

Common Data Bus is a bus that broadcasts the result into different units required. Bus carries the name of the source along with the data and when there is a match of the source, required functional unit will grasp the data into it.

Tomasulo's example:

To illustrate the technique of Tomasulo's algorithm, following instructions are considered and let us check how they are being executed using the Tomasulo's hardware

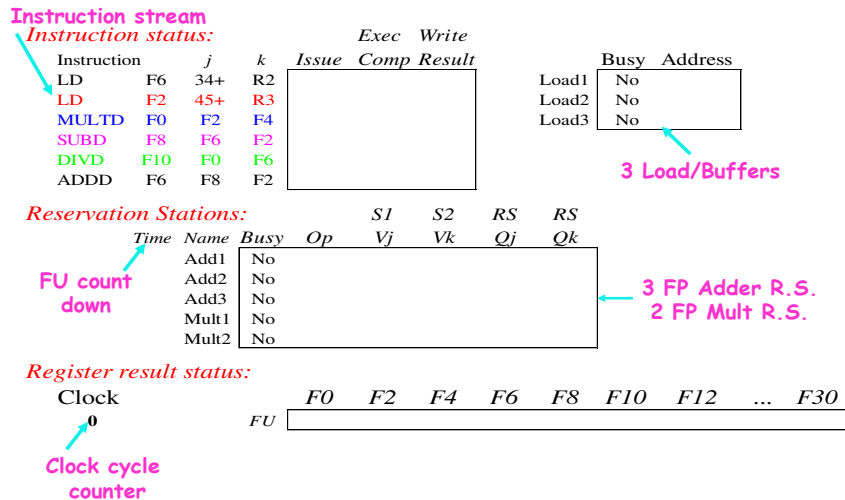
Instruction		<i>j</i>	<i>k</i>
LD	F6	34+	R2
LD	F2	45+	R3
MUL TD	F0	F2	F4
SUBD	F8	F6	F2
DIVD	F10	F0	F6
ADDD	F6	F8	F2

Assume that hardware configuration has 3 adder units, 2 multiplier units and 3 load buffers for doing the operation. Each instruction has to go through three different stages of the execution. Let the latency value for addition and subtraction be equal to 2 clock cycles, 10 clock cycles for multiplication and 40 clock cycles for division. Load operation by default takes 2 clock cycles.

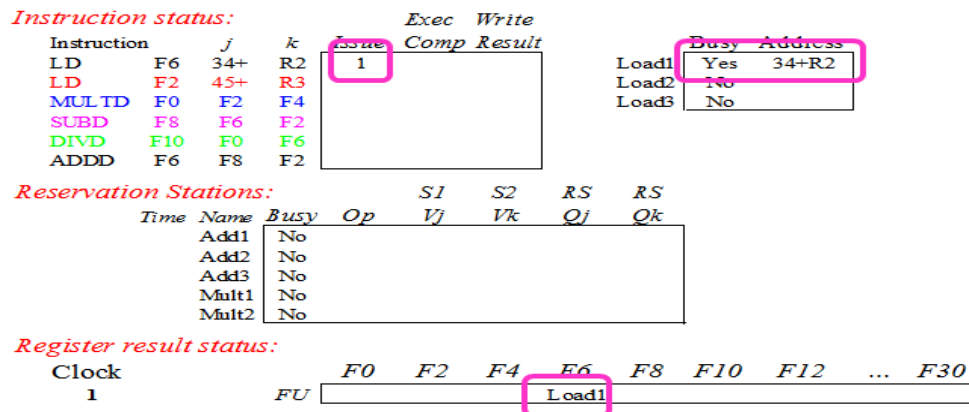
The execution follows in-order issue were in every clock cycle there will be an instruction issue if the reservation station is available.

1. Initial configuration of the system at Clock cycle 0:

Tomasulo Example



2. At clock cycle 1:



In this clock cycle, first instruction LD is issued and operands are loaded into load buffer making it busy and name of corresponding reservation station is written in register result status for the destination register **F6** of the load instruction.

3. At clock cycle 2:

Instruction status:

Instruction	j	k	Issue	Exec	Write
LD F6 34+ R2			1		
LD F2 45+ R3			2		
MULTD F0 F2 F4					
SUBD F8 F6 F2					
DIVD F10 F0 F6					
ADDD F6 F8 F2					

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
Add1	No						
Add2	No						
Add3	No						
Mult1	No						
Mult2	No						

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
2	FU	Load2		Load1					

At cycle 2 second instruction is issued and operands are loaded into load buffer and corresponding destination register F2 of the instruction is replaced with Load2

4. At clock cycle 3:

Instruction status:

Instruction	j	k	Issue	Exec	Write
LD F6 34+ R2			1	3	
LD F2 45+ R3			2		
MULTD F0 F2 F4			3		
SUBD F8 F6 F2					
DIVD F10 F0 F6					
ADDD F6 F8 F2					

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
Add1	No						
Add2	No						
Add3	No						
Mult1	Yes	MULTD		R(F4)	Load2		
Mult2	No						

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
3	Mult1	Load2		Load1					

In this cycle instruction 3 MULTD is issued and operands are loaded into reservation station mult1. First Operand value F2 is being operated by Load2 unit and is written under Qj and second operand F4 is available and is written under Vk. In the mean time Load1 completes its execution and is ready to write the result in next clock cycle.

5. At clock cycle 4:

Instruction status:

Instruction	j	k	Exec Write			Load1	Load2	Load3	Busy	Address
			Issue	Comp	Result					
LD	F6	34+	R2	1	3	4			No	
LD	F2	45+	R3	2	4				Yes	45+R3
MULTD	F0	F2	F4	3					No	
SUBD	F8	F6	F2	4					No	
DIVD	F10	F0	F6						No	
ADDD	F6	F8	F2						No	

Reservation Stations:

Time	Name	Busy	Op	S1	S2	RS	RS
				Vj	Vk	Qj	Qk
Add1	Yes	SUBD	M(A1)				Load2
Add2	No						
Add3	No						
Mult1	Yes	MULTD		R(F4)		Load2	
Mult2	No						

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
4	FU	Mult1	Load2		M(A1)	Add1			

In this cycle, instruction 4 SUBD gets issued and operand values are loaded into Add1. Operand 1 is available and operand 2 is not available and is operated by Load2. Hence Load2 is written under Qk. Instruction 2 completes execution and instruction 1 writes result into the register and Load1 buffer becomes available.

6. At clock cycle 5:

Instruction status:

Instruction	j	k	Exec Write			Load1	Load2	Load3	Busy	Address
			Issue	Comp	Result					
LD	F6	34+	R2	1	3	4			No	
LD	F2	45+	R3	2	4	5			No	
MULTD	F0	F2	F4	3					No	
SUBD	F8	F6	F2	4					No	
DIVD	F10	F0	F6	5					No	
ADDD	F6	F8	F2						No	

Reservation Stations:

Time	Name	Busy	Op	S1	S2	RS	RS
				Vj	Vk	Qj	Qk
2	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	No					
	Add3	No					
10	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
5	FU	Mult1	M(A2)		M(A1)	Add1	Mult2		

In this cycle, fifth instruction DIVD is issued and is loaded into the reservation station Mult2. Operand 1 is not available and is written under Qj and operand 2 is available. Load2 completes writing into the register and becomes available. Values of Load2 is taken by Add1 and mult1. Both operands of these instructions are available and both the instructions starts execution in the next clock cycle. Update the register result status correspondingly.

7. At clock cycle 6:

Instruction status:

Instruction	j	k	Issue	Comp	Result	Load1	Load2	Load3	Busy	Address
LD	F6	34+	R2	1	3	4			No	
LD	F2	45+	R3	2	4	5			No	
MULTD	F0	F2	F4	3					No	
SUBD	F8	F6	F2	4						
DIVD	F10	F0	F6	5						
ADDD	F6	F8	F2	6						

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
1	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
9	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
6	FU	Mult1	M(A2)		Add2	Add1	Mult2		

In this cycle, last instruction ADDD gets issued and operands are loaded into the Reservation station add2. First operand is not available and its name of the reservation station is written under Qj. Corresponding Register result status is also updated.

8. At clock cycle 7:

Instruction status:

Instruction	j	k	Issue	Comp	Result	Load1	Load2	Load3	Busy	Address
LD	F6	34+	R2	1	3	4			No	
LD	F2	45+	R3	2	4	5			No	
MULTD	F0	F2	F4	3					No	
SUBD	F8	F6	F2	4	7					
DIVD	F10	F0	F6	5						
ADDD	F6	F8	F2	6						

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
0	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
8	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
7	FU	Mult1	M(A2)		Add2	Add1	Mult2		

In this cycle, instruction 4 SUBD completes its execution.

9. At clock cycle 8:

Instruction status:

Instruction	j	k	Issue	Comp	Result	Load1	Load2	Load3	Busy	Address
LD	F6	34+	R2	1	3	4			No	
LD	F2	45+	R3	2	4	5			No	
MULTD	F0	F2	F4	3					No	
SUBD	F8	F6	F2	4	7	8				
DIVD	F10	F0	F6	5						
ADDD	F6	F8	F2	6						

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
2	Add2	Yes	ADDD	M(A1)	M(A2)		
	Add3	No					
7	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
8	FU	Mult1	M(A2)		Add2	M(A1)	Mult2		

In this cycle, SUBD writes result and the result is used by the reservation station Add2 and the reservation station Add1 is emptied for the later usage. In the clock cycle ADDD starts its execution.

10. At clock cycle 9 nothing will happen and in clock cycle 10 ADDD completes its execution and will write the result in next cycle.

Instruction status:

Instruction	j	k	Exec Write			Load1	Load2	Load3	Busy	Address
			Issue	Comp	Result					
LD	F6	34+	R2	1	3	4			No	
LD	F2	45+	R3	2	4	5			No	
MULTD	F0	F2	F4	3					No	
SUBD	F8	F6	F2	4	7	8				
DIVD	F10	F0	F6	5						
ADDD	F6	F8	F2	6	10					

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
0	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
5	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
10	FU	Mult1	M(A2)		Add2	(M-M)	Mult2		

11. AT clock cycle 11:

Instruction status:

Instruction	j	k	Exec Write			Load1	Load2	Load3	Busy	Address
			Issue	Comp	Result					
LD	F6	34+	R2	1	3	4			No	
LD	F2	45+	R3	2	4	5			No	
MULTD	F0	F2	F4	3					No	
SUBD	F8	F6	F2	4	7	8				
DIVD	F10	F0	F6	5						
ADDD	F6	F8	F2	6	10	11				

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
4	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30	
11	FU									
	Mult1	M(A2)		(M-M+N)	(M-M)	Mult2				

In this cycle, ADDD finishes writing of the result and reservation station ADD2 is made available and the result is written in register result status.

12. At clock cycle 12, 13 and 14 no operation will take place and in clock cycle 15 MULTD will finish its execution.

Instruction status:

Instruction	j	k	Exec Write			Load1	Load2	Load3	Busy	Address
			Issue	Comp	Result					
LD	F6	34+	R2	1	3	4			No	
LD	F2	45+	R3	2	4	5			No	
MULTD	F0	F2	F4	3	15				No	
SUBD	F8	F6	F2	4	7	8				
DIVD	F10	F0	F6	5						
ADDD	F6	F8	F2	6	10	11				

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
0	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
15	FU	Mult1	M(A2)		(M-M+N)	(M-M)	Mult2			

13. At clock cycle 16:

Instruction status:

Instruction	j	k	Issue	Exec	Write
LD	F6	34+	R2	1	3 4
LD	F2	45+	R3	2	4 5
MULTD	F0	F2	F4	3	15 16
SUBD	F8	F6	F2	4	7 8
DIVD	F10	F0	F6	5	
ADDD	F6	F8	F2	6	10 11

Load1: No, Load2: No, Load3: No

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
Add1		No					
Add2		No					
Add3		No					
Mult1		No					
40 Mult2		Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
16	FU	M*F4	M(A2)		(M-M+N)	(M-M)	Mult2		

In this cycle, MULTD will write the result and reservation station Mult1 is emptied. Result of Mult1 is used by the reservation station Mult2 and starts its execution in the next clock cycle.

14. Nothing will happen until clock cycle 56 and in cycle 56 DIVD finishes its execution

Instruction status:

Instruction	j	k	Issue	Exec	Write
LD	F6	34+	R2	1	3 4
LD	F2	45+	R3	2	4 5
MULTD	F0	F2	F4	3	15 16
SUBD	F8	F6	F2	4	7 8
DIVD	F10	F0	F6	5	56
ADDD	F6	F8	F2	6	10 11

Load1: No, Load2: No, Load3: No

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
Add1		No					
Add2		No					
Add3		No					
Mult1		No					
0 Mult2		Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
56	FU	M*F4	M(A2)		(M-M+N)	(M-M)	Mult2		

15. At clock cycle 57 Mult2 writes the result and the RS is emptied.

Instruction status:

Instruction	j	k	Issue	Exec	Write
LD	F6	34+	R2	1	3 4
LD	F2	45+	R3	2	4 5
MULTD	F0	F2	F4	3	15 16
SUBD	F8	F6	F2	4	7 8
DIVD	F10	F0	F6	5	56 57
ADDD	F6	F8	F2	6	10 11

Load1: No, Load2: No, Load3: No

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
Add1		No					
Add2		No					
Add3		No					
Mult1		No					
Mult2		Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
57	FU	M*F4	M(A2)		(M-M+N)	(M-M)	Result		

You can observe that there is in-order issue, out-of-order execution and out-of-order write.

VI. Hardware-Based Speculation:

It is one the technique to overcome the control dependence so as to increase ILP into another level.

→The Hardware approach for this is also called as **Extended Tomasulo's Approach**. The Tomasulo's algorithm carries out the out-of-order execution but there is not method to overcome the control dependence. But in the speculation hardware, required changes are made in hardware to speculate the branch instructions.

→Here, to overcome control dependence the technique uses speculation method. The speculation method speculates on the on the outcome of the branches and executing the program as if the guess is correct. Dynamic prediction technique just fetches the instruction but this method executes the fetched instructions as predicted.

→Three key ideas are combined in Hardware speculation

1. **Dynamic Branch prediction:** predicts the direction of a branch instruction if it is taken or not taken.
2. **Speculative execution:** Carries the execution in the predicted direction with the ability to undo changes if the prediction is carried out in a wrong manner.
3. **Dynamic scheduling:** The technique carries in-order instruction issue and out-of-order instruction execution.

Apart from this, hardware, there is an additional step in the execution called as **instruction commit**. This step writes the value into memory location or register file when the instruction is found no more speculative.

Overall idea is to carry in-order instruction issue, out-of-order instruction execution and in-order instruction issue.

Extending Tomasulo's approach support speculation:

→Adding of commit phase in the execution is done by the usage of an additional hardware called **Reorder Buffer** which is a small memory element like reservation station.

→Reorder buffer performs two functionalities:

1. The buffer holds the operated result between the instruction completion and instruction commit.
2. Register renaming function done by reservation station in basic Tomasulo's algorithm is being replaced by Reorder buffer to accomplish the commit stage successfully.
3. Store buffers of basic Tomasulo's are being replaced by Reorder buffer.

Structure of Reorder buffer:

type	dest	value	fin

Above shown is the structure of reorder buffer, it has four fields and one pointer called head.

Four fields are:

1. **Type:** This field indicates the type instruction that is getting operated on.
2. **Dest:** This field stores the name of the destination register that needs to be updated after commit stage.
3. **Value:** This field holds the operated value till it is being committed.
4. **Fin:** This field indicates the whether the execution is completed or not. If the head is pointing to that area, it can be committed.

There is one pointer called head pointer which points to the instruction committing in-order according to the sequence.

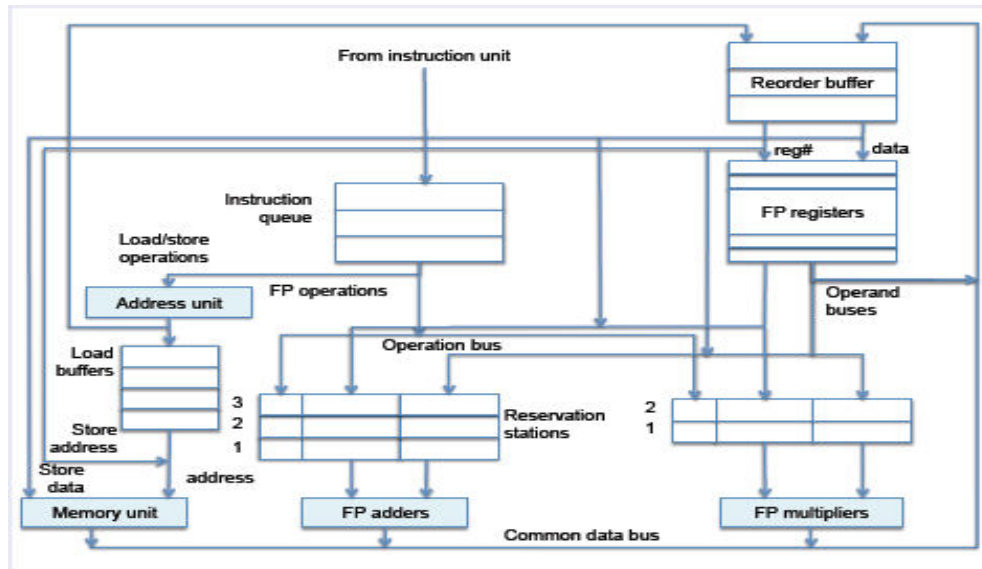
Structure of Reservation Station:

busy	op	ROB#	Qj	Qk	Vj	Vk
busy	op	ROB#	Qj	Qk	Vj	Vk
busy	op	ROB#	Qj	Qk	Vj	Vk
busy	op	ROB#	Qj	Qk	Vj	Vk

There is a little change in the Reservation Structure and its functionality when used in Speculation algorithm.

→Functionality of the Reservation station is to hold the values of the operands when operated until it reaches the write stage. Apart from that when you look into the structure there is one extra field **ROB#** which stores the name of the Reorder buffer entry which is involved in the operation. Rest all the fields are similar to the one discussed before.

Tomasulo's algorithm with speculation (Extended Tomasulo's Hardware)



Above is the hardware configuration for the speculation hardware which is almost similar to the basic Tomasulo's hardware with very small differences:

1. Common Data Bus (CDB) in this hardware is connected to all the units except the general purpose registers.
2. Result is written into the memory location and register file only by the Reorder buffer at the commit stage.

Apart from this, instruction execution has four different stages:

1. **Issue:** Get the instruction from the instruction queue and load the instruction into Reservation station and Reorder buffer if both units are available. If any one unit is unavailable raise the structural hazard until it becomes empty.
→ Load the operands into the reservation station and destination register into Reorder Buffer. If operands are unavailable wait for the Reorder buffer producing the result.
2. **Execute:** The stage is very similar to the one discussed in the previous section. If the operands available start the execution or else monitor CDB till it produces result.
3. **Write:** In this stage write the result over CDB and send the value into Reorder buffer with a Reorder buffer tag.
4. **Commit:** A final stage, were in commit the values into the memory location and register file if the header is pointing to the completed instruction.

Note: For the example problem of this technique please refer the slide supplied to you. If you have any doubts please clear it. 😊

VII, Exploiting ILP Using Multiple Issue and Static Scheduling

Taking Advantage of More ILP with Multiple Issues:

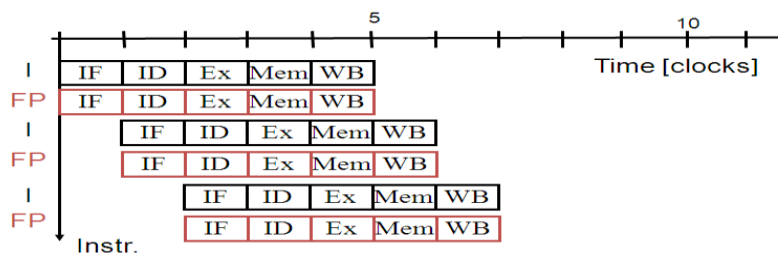
So far we have seen different ILP techniques, where there was only one instruction getting issued in a single clock cycle. More ILP can be achieved by issuing more than one independent instruction in a single clock cycle using **Multiple Issue Processors**

Multiple Issue processors are the one that allow multiple instructions to issue in a single clock cycle. There are three types of multiple issue processors:

1. **Statically scheduled superscalar processor:** The processor issues a variable amount of instruction in a single clock cycle and does the in-order execution at the compile time.
2. **Dynamically scheduled superscalar processor:** The processor issues variable amount of instructions in a single clock cycle and carries out-of-order execution.
3. **Very Long Instruction Word (VLIW) processor:** Issues a fixed amount of instructions as a packet format and carries out in-order execution at the compile time.

Super scalar MIPS:

As an example, consider a superscalar MIPS processor that can issue two instructions in a single clock cycle. One of them is FP and the other one is any other instruction. In that case there will be two pipelines running in parallel in single clock cycle. The main disadvantage is that if the first pipeline is blocked, even the second pipeline has to be blocked even though it is an independent instruction. Typical pipeline structure is shown as below:



Loop Unrolling in Superscalar: Example

To illustrate the technique of loop unrolling in superscalar processor, consider the following MIPS instructions which are unrolled four times by the loop unrolling technique. Assume that a superscalar processor has two functional units one for doing integer operations and the other for doing floating point operations.

```

Loop: LD    F0, 0(R1)      ;F0=array el.
      ADDD  F4,F0,F2      ;add scalar in F2
      SD    0(R1),F4      ;store result
      SUBI  R1,R1,#8      ;decrement pointer
      BNEZ  R1, Loop      ;branch

```

Given the latency value for the dependent instructions:

Instruction producing result	Instruction using result	Latency
FP ALU op	FP ALU op	3
FP ALU op	SD	1
LD	FP ALU op	1
LD	SD	0
Int ALU op	Int ALU op	0

Before unrolling it using superscalar technique, first realize the loop unrolling in the simple MIPS processor and then reorder it accordingly based on the latency cycle. Following is the unrolled loop using MIPS processor:

```

Loop: LD    F0,0(R1)
      LD    F6,-8(R1)
      LD    F10,-16(R1)
      LD    F14,-24(R1)

      ADDD  F4,F0,F2
      ADDD  F8,F6,F2
      ADDD  F12,F10,F2
      ADDD  F16,F14,F2
      SD    F4 ,0(R1)
      SD    F8 ,-8(R1)

      SUBI  R1,R1,#32
      SD    F12, 16(R1)

      BNEZ  R1,Loop
      SD    F16, 8(R1)

```

Next rearrange the same instructions using superscalar functional unit as shown below:

CIK	Integer Instr.	FP Instr.
1	Loop: LD F0,0(R1)	
2	LD F6,-8(R1)	
3	LD F10,-16(R1)	ADDD F4,F0,F2
4	LD F14,-24(R1)	ADDD F8,F6,F2
5	SD F4,0(R1)	ADDD F12,F10,F2
6	SD -8(R1),F8	ADDD F16,F14,F2
7	SUBI R1,R1,#32	
8	SD F12,16(R1)	
9	BNEZ R1,Loop	
10	SD F16,8(R1)	

Basic VLW approach:

VLIW uses multiple, independent functional units which packages multiple independent operation into one very long instruction.

Compiler is responsible for finding out the independent instruction and arranging it in the form of packets.

One of the main differences in the superscalar and this approach is that, here if a previous instruction or the functional unit is blocked, then other units are not blocked if there are any independent instructions in the program.

Any functional unit can operate if any other functional unit is blocked in the packet of VLIW.

VLIW Loop unrolling example:

Consider the following MIPS code unrolled for 5 iterations at a time. Assume that a VLIW approach has 2 memory units, 2 FP units and 1 integer/branch unit:

Loop: LD	F0, 0(R1)	;F0=array el.
ADDD	F4,F0,F2	;add scalar in F2
SD	0(R1),F4	;store result
SUBI	R1,R1,#8	;decrement pointer
BNEZ	R1, Loop	;branch

For the above instructions, latency cycle is given by:

Instruction producing result	Instruction using result	Latency
FP ALU op	FP ALU op	3
FP ALU op	SD	1
LD	FP ALU op	1
LD	SD	0
Int ALU op	Int ALU op	0

Before applying VLIW approach, unroll the loop according to MIPS processor as shown below:

```

Loop: LD      F0,0(R1)
      LD      F6,-8(R1)
      LD      F10,-16(R1)
      LD      F14,-24(R1)
      LD      F18,-32(R1)
      ADDD    F4,F0,F2
      ADDD    F8,F6,F2
      ADDD    F12,F10,F2
      ADDD    F16,F14,F2
      ADDD    F20,F18,F2
      SD      F4,0(R1)
      SD      F8,-8(R1)
      SD      F12,-16(R1)
      SUBI    R1,R1,#40
      SD      F16,16(R1)
      BNEZ    R1,Loop
      SD      F20,8(R1)

```

Next, let us reorder the instructions according to the VLIW approach as shown below for the 5 functional units:

	Mem. Ref1	Mem Ref. 2	FP1	FP2	Int/Branch
1	LD F0, 0(R1)	LD F6, -8(R1)			
2	LD F10, -16(R1)	LD F14, -24(R1)			
3	LD F18, -32(R1)		ADDD F4, F0, F2	ADDD F8, F6, F2	
4			ADDD F12, F10, F2	ADDD F16, F14, F2	
5	SD F4, 0(R1)	SD F8, -8(R1)	ADDD F20, F18, F2		SUBI R1, R1, #40
6	SD F12, 24(R1)	SD F16, 16(R1)			
7	SD F20, 8(R1)				BNEZ R1, LOOP

Here instructions in each clock cycle is one packet of VLIW .