ADVANCE COMPUTER ARCHITECTURE

UNIT-4

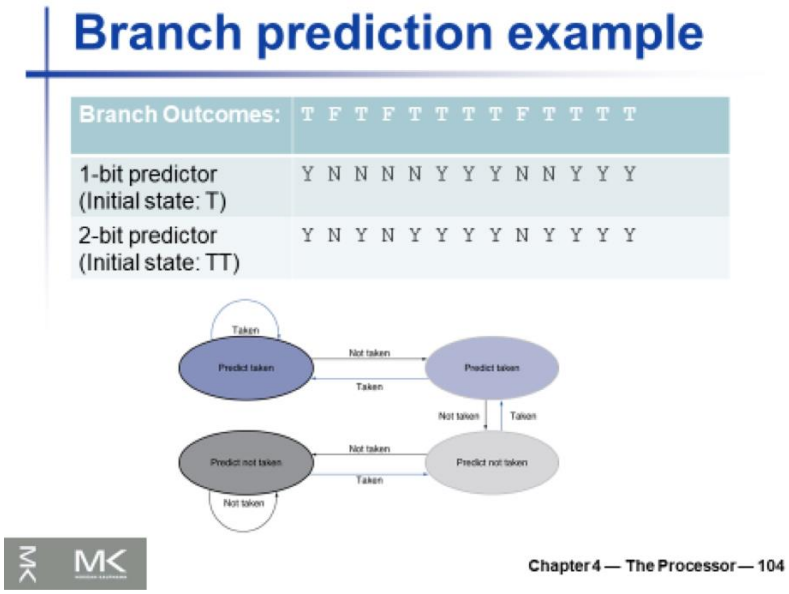
BRANCH PREDICTION:

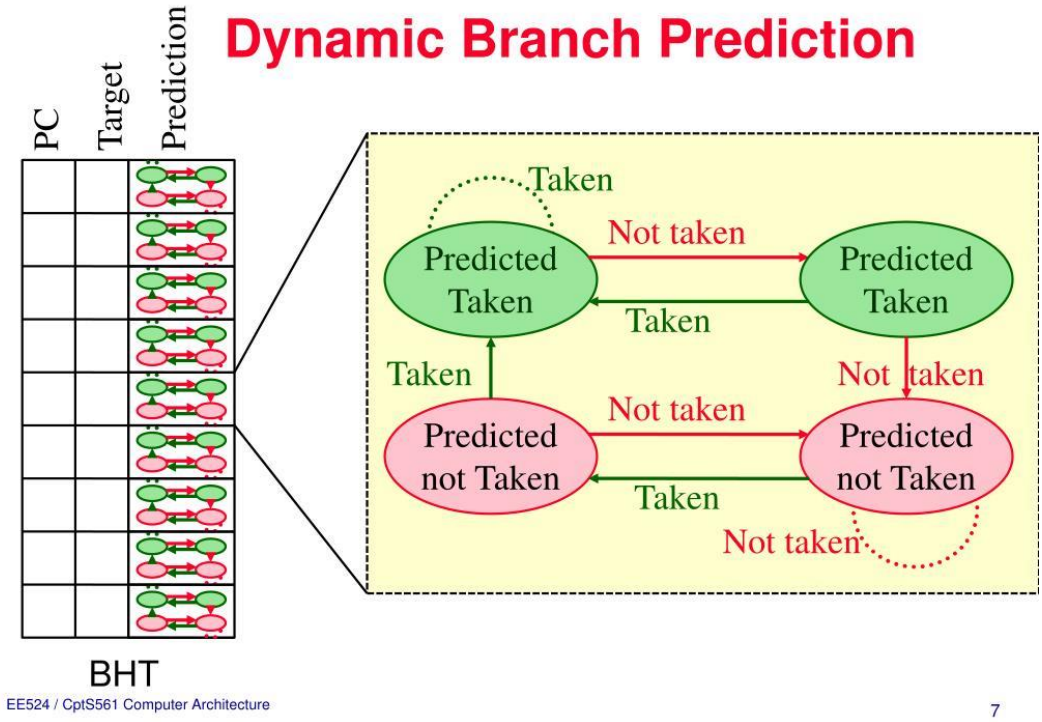
Branch prediction is an approach to computer architecture that attempts to mitigate the costs of branching. Branch predication speeds up the processing of branch instructions with CPUs using pipelining. The technique involves only executing certain instructions if certain predicates are true. Branch prediction is typically implemented in hardware using a branch predictor.

Branch prediction is also known as branch predication or simply as predication.

Branch prediction is a technique used to speed execution of instructions on processors that use pipelining. CPUs initially executed instructions one by one as they came in, but the introduction of pipelining meant that branching instructions could slow the processor down significantly as the processor has to wait for the conditional jump to be executed.

Branch prediction breaks instructions down into predicates, similar to predicate logic. A CPU using branch prediction only executes statements if a predicate is true. One example is using conditional logic. Since unnecessary code is not executed, the processor can work much more efficiently. Branch prediction is implemented in CPU logic with a branch predictor.





Why do we need branch prediction?

1. The gain produced by Pipelining can be reduced by the presence of program transfer instructions eg JMP, CALL, RET etc
2. They change the sequence causing all the instructions that entered the pipeline after program transfer instructions invalid
3. Thus no work is done as the pipeline stages are reloaded.

Branch prediction logic:

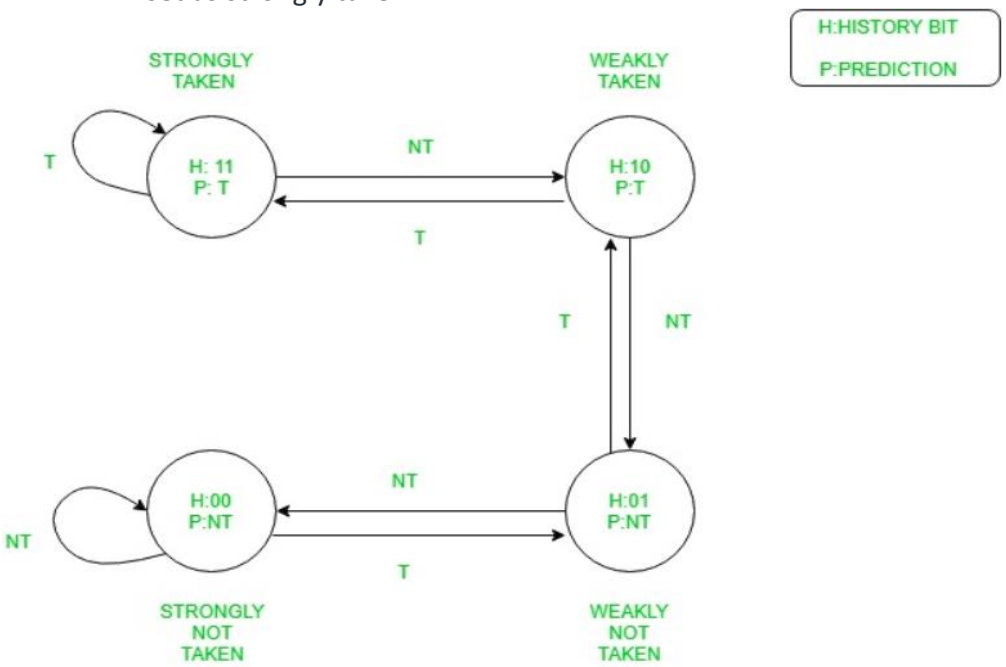
To avoid this problem, Pentium uses a scheme called Dynamic Branch Prediction. In this scheme, a prediction is made for the branch instruction currently in the pipeline. The prediction will either be taken or not taken. If the prediction is true then the pipeline will not be flushed and no clock cycles will be lost. If the prediction is false then the pipeline is flushed and starts over with the current instruction.
It is implemented using 4 way set associated cache with 256 entries. This is called **Branch Target Buffer (BTB)**. The directory entry for each line consists of:

- **Valid bit:** Indicates whether the entry is valid or not.
- **History bit:** Track how often bit has been taken.
-

Source memory address is from where the branch instruction was fetched. If the directory entry is valid then the target address of the branch is stored in corresponding data entry in BTB.

Working of Branch Prediction:

1. BTB is a lookaside cache that sits to the side of Decode Instruction(DI) stage of 2 pipelines and monitors for branch instructions.
2. The first time that a branch instruction enters the pipeline, the BTB uses its source memory to perform a lookup in the cache.
3. Since the instruction was never seen before, it is BTB miss. It predicts that the branch will not be taken even though it is unconditional jump instruction.
4. When the instruction reaches the EU(execution unit), the branch will either be taken or not taken. If taken, the next instruction to be executed will be fetched from the branch target address. If not taken, there will be a sequential fetch of instructions.
5. When a branch is taken for the first time, the execution unit provides feedback to the branch prediction. The branch target address is sent back which is recorded in BTB.
6. A directory entry is made containing the source memory address and history bit is set as strongly taken.



The diagram is explained by the following table:

History Bits	Resulting Description	Prediction made	If branch taken	If branch not taken
11	Strongly Taken	Branch Taken	Remains in same state	Downgraded to weakly taken
10	Weakly Taken	Branch Taken	Upgraded to strongly taken	Downgraded to weakly not taken
01	Weakly Not Taken	Branch Not Taken	Upgraded to weakly taken	Downgraded to strongly not taken

History Bits	Resulting Description	Prediction made	If branch taken	If branch not taken
00	Strongly Not Taken	Branch Not Taken	Upgraded to weakly not taken	Remains in same state

DIRECTION PREDICTOR:

Branch prediction (BP) is one of an earliest executions methods that still discover the importance of modern architecture [3]. Recently, branch prediction (BP) has prompted to the advancement of branch prediction techniques that accomplish better result and accuracy.

Basically branch prediction predicts two problems:

1) direction predicting, and

2) calculating the target address.

Branch prediction schemes are of two types: static branch schemes and dynamic branch schemes.

A static branch scheme (software techniques) is very simple and easy. This scheme assembles the majority of the data/information prior to the execution of the program or during the compile time and it doesn't require any hardware whereas, a dynamic branch scheme (hardware techniques) is based on the hardware and it assembles the information during the run-time of the program. Dynamic schemes are more assorted as they keep track during run-time of the program execution

Techniques for Static Branch Prediction Static prediction techniques are very simple to analyze and require low cost [5]and less energy to execute instructions because it does not require any history table of instructions as well as hardware component [6]. In most of the system, compilers can provide good coverage for such types of branches.

Single direction prediction: Single direction is the easiest strategy in static prediction schemes. In this prediction, the directions of all branches will dependably go in a similar way, regardless of whether the branch prediction took or not. In this prediction, when the branch prediction is taken it gives a better result as compared to the not taken prediction

Backward taken forward not taken (BTFT): In branch taken forward not taken schemes, most of the loops are backward jump loops and which will be taken more often than forwarding loops. This will upgrade the execution of the prediction

Program based prediction: This prediction use structural based information of the program. A distinguished case of this prediction is portrayed by the Ball et al. [9]and Calder et. al [10]. A set of heuristics presents by Ball et al. [9] are based on the operands, opcode, and information of executed branches

Profile-based branch prediction: This branch prediction utilizes information from the previous execution of the program with different inputs

hierarchical predictors:

Hierarchical architecture views the whole system as a hierarchy structure, in which the software system is decomposed into logical modules or subsystems at different levels in the hierarchy. This approach is typically used in designing system software such as network protocols and operating systems.

In system software hierarchy design, a low-level subsystem gives services to its adjacent upper level subsystems, which invoke the methods in the lower level. The lower layer provides more specific functionality such as I/O services, transaction, scheduling, security services, etc. The middle layer provides more domain dependent functions such as business logic and core processing services. And, the upper layer provides more abstract functionality in the form of user interface such as GUIs, shell programming facilities, etc.

It is also used in organization of the class libraries such as .NET class library in namespace hierarchy. All the design types can implement this hierarchical architecture and often combine with other architecture styles.

Hierarchical architectural styles is divided as −

- Main-subroutine
- Master-slave
- Virtual machine

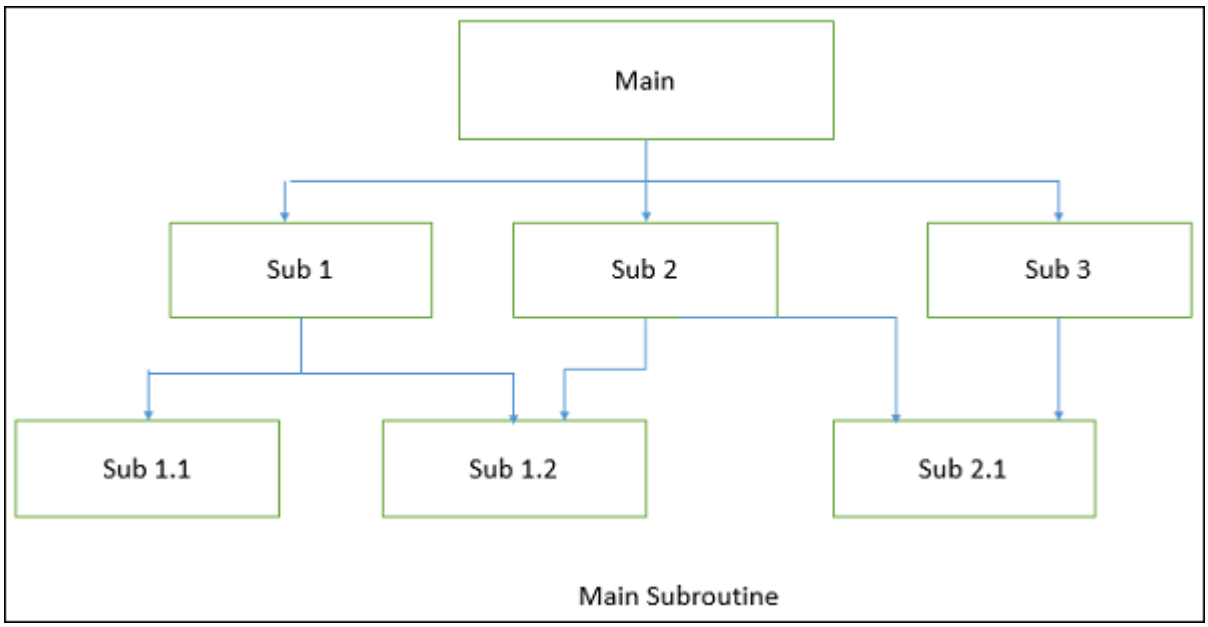
Main-subroutine

The aim of this style is to reuse the modules and freely develop individual modules or subroutine. In this style, a software system is divided into subroutines by using top-down refinement according to desired functionality of the system.

These refinements lead vertically until the decomposed modules is simple enough to have its exclusive independent responsibility. Functionality may be reused and shared by multiple callers in the upper layers.

There are two ways by which data is passed as parameters to subroutines, namely −

- **Pass by Value** − Subroutines only use the past data, but can't modify it.
- **Pass by Reference** − Subroutines use as well as change the value of the data referenced by the parameter.



Advantages

- Easy to decompose the system based on hierarchy refinement.
- Can be used in a subsystem of object oriented design.

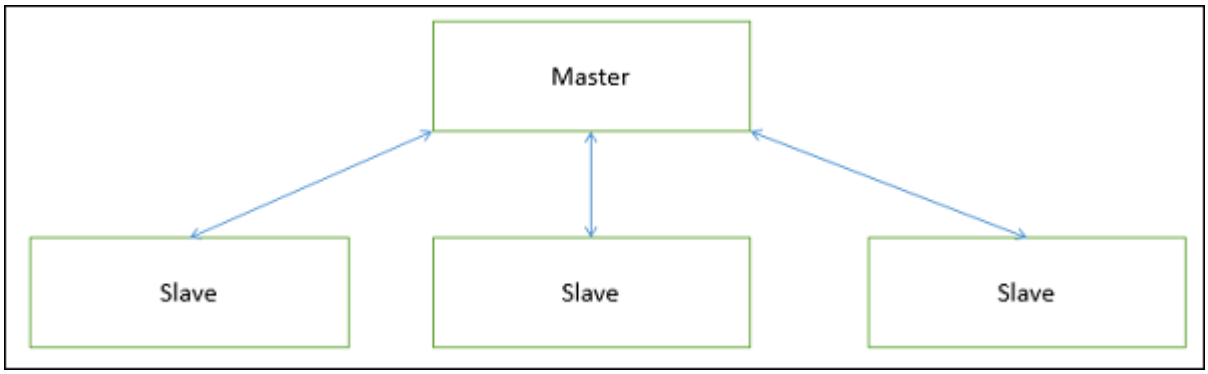
Disadvantages

- Vulnerable as it contains globally shared data.
- Tight coupling may cause more ripple effects of changes.

Master-Slave

This approach applies the 'divide and conquer' principle and supports fault computation and computational accuracy. It is a modification of the main-subroutine architecture that provides reliability of system and fault tolerance.

In this architecture, slaves provide duplicate services to the master, and the master chooses a particular result among slaves by a certain selection strategy. The slaves may perform the same functional task by different algorithms and methods or totally different functionality. It includes parallel computing in which all the slaves can be executed in parallel.



The implementation of the Master-Slave pattern follows five steps −

- Specify how the computation of the task can be divided into a set of equal sub-tasks and identify the sub-services that are needed to process a sub-task.
- Specify how the final result of the whole service can be computed with the help of the results obtained from processing individual sub-tasks.
- Define an interface for the sub-service identified in step 1. It will be implemented by the slave and used by the master to delegate the processing of individual sub-tasks.
- Implement the slave components according to the specifications developed in the previous step.
- Implement the master according to the specifications developed in step 1 to 3.

Applications

- Suitable for applications where reliability of software is critical issue.
- Widely applied in the areas of parallel and distributed computing.

Advantages

- Faster computation and easy scalability.
- Provides robustness as slaves can be duplicated.
- Slave can be implemented differently to minimize semantic errors.

Disadvantages

- Communication overhead.
- Not all problems can be divided.
- Hard to implement and portability issue.

Virtual Machine Architecture

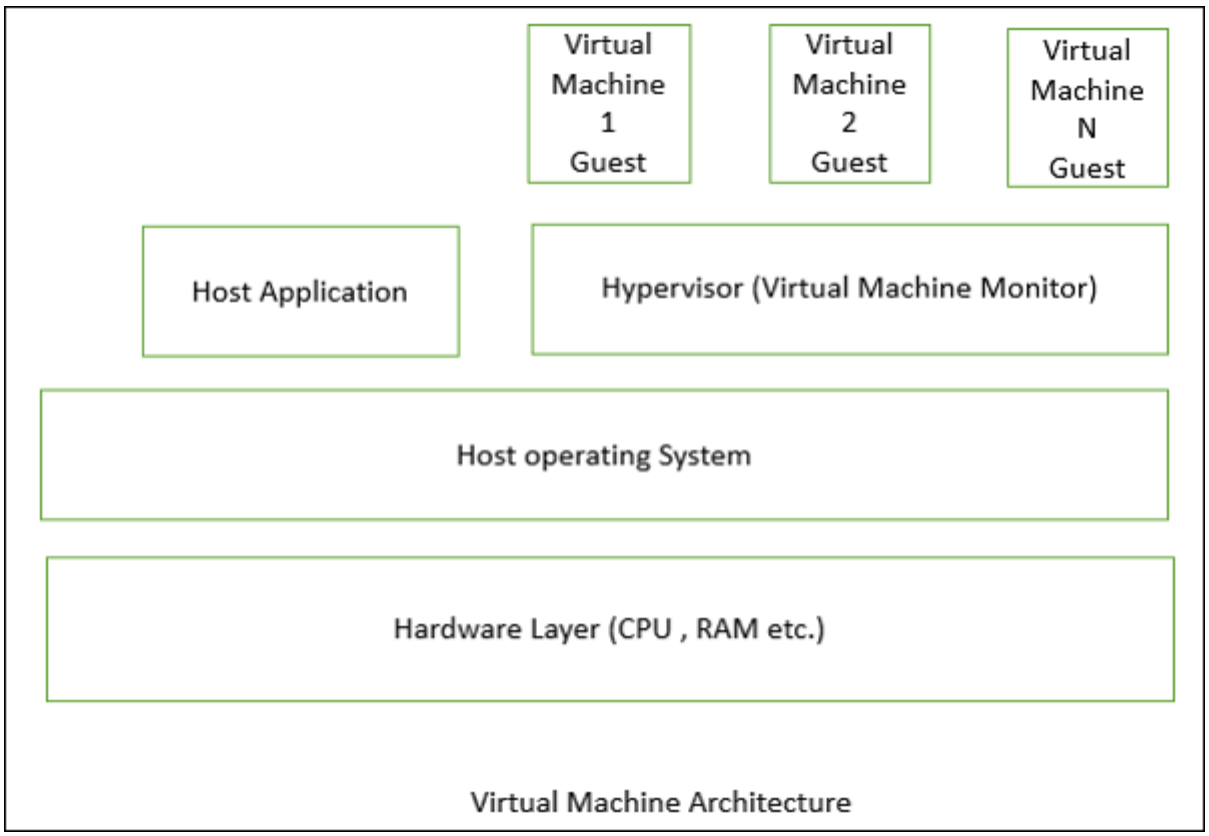
Virtual Machine architecture pretends some functionality, which is not native to the hardware and/or software on which it is implemented. A virtual machine is built upon an existing system and provides a virtual abstraction, a set of attributes, and operations.

In virtual machine architecture, the master uses the ‘same’ subservice’ from the slave and performs functions such as split work, call slaves, and combine results. It allows developers to simulate and test platforms, which have not yet been built, and simulate "disaster'' modes that would be too complex, costly, or dangerous to test with the real system.

In most cases, a virtual machine splits a programming language or application environment from an execution platform. The main objective is to provide **portability**. Interpretation of a particular module via a Virtual Machine may be perceived as −

- The interpretation engine chooses an instruction from the module being interpreted.
- Based on the instruction, the engine updates the virtual machine’s internal state and the above process is repeated.

The following figure shows the architecture of a standard VM infrastructure on a single physical machine.



The **hypervisor,** also called the **virtual machine monitor**, runs on the host OS and allocates matched resources to each guest OS. When the guest makes a system-call, the hypervisor intercepts and translates it into the corresponding system-call supported by the host OS. The hypervisor controls each virtual machine access to the CPU, memory, persistent storage, I/O devices, and the network.

Applications

Virtual machine architecture is suitable in the following domains −

- Suitable for solving a problem by simulation or translation if there is no direct solution.
- Sample applications include interpreters of microprogramming, XML processing, script command language execution, rule-based system execution, Smalltalk and Java interpreter typed programming language.
- Common examples of virtual machines are interpreters, rule-based systems, syntactic shells, and command language processors.

Advantages

- Portability and machine platform independency.
- Simplicity of software development.
- Provides flexibility through the ability to interrupt and query the program.
- Simulation for disaster working model.
- Introduce modifications at runtime.

Disadvantages

- Slow execution of the interpreter due to the interpreter nature.
- There is a performance cost because of the additional computation involved in execution.

Layered Style

In this approach, the system is decomposed into a number of higher and lower layers in a hierarchy, and each layer has its own sole responsibility in the system.

- Each layer consists of a group of related classes that are encapsulated in a package, in a deployed component, or as a group of subroutines in the format of method library or header file.
- Each layer provides service to the layer above it and serves as a client to the layer below i.e. request to layer i +1 invokes the services provided by the layer i via the interface of layer i. The response may go back to the layer i +1 if the task is completed; otherwise layer i continually invokes services from layer i -1 below.

Applications

Layered style is suitable in the following areas −

- Applications that involve distinct classes of services that can be organized hierarchically.
- Any application that can be decomposed into application-specific and platform-specific portions.
- Applications that have clear divisions between core services, critical services, and user interface services, etc.

Advantages

- Design based on incremental levels of abstraction.
- Provides enhancement independence as changes to the function of one layer affects at most two other layers.
- Separation of the standard interface and its implementation.
- Implemented by using component-based technology which makes the system much easier to allow for plug-and-play of new components.
- Each layer can be an abstract machine deployed independently which support portability.
- Easy to decompose the system based on the definition of the tasks in a top-down refinement manner
- Different implementations (with identical interfaces) of the same layer can be used interchangeably

Disadvantages

- Many applications or systems are not easily structured in a layered fashion.
- Lower runtime performance since a client's request or a response to client must go through potentially several layers.
- There are also performance concerns on overhead on the data marshaling and buffering by each layer.
- Opening of interlayer communication may cause deadlocks and "bridging" may cause tight coupling.
- Exceptions and error handling is an issue in the layered architecture, since faults in one layer must spread upwards to all calling layers.

IF CONVERSION AND CONDITIONAL MOVE:

If-Conversion Technique:

The if-conversion is important for understanding how the predication works in the hardware. The if-conversion talks about how the compiler creates the code that will be executed along both paths. Let's see an example. Suppose we have the following C code,

```
if (cond) {
   x = arr[i];
   y += 1;
} else {
   x = arr[j];
   y -= 1;
}
```

The if-conversion means that the compiler will compile the work on both paths, then:

```
x1 = arr[i];
x2 = arr[j];
y1 = y + 1;
y2 = y + 2;
x = cond?x1:x2;
y = cond?y1:y2;
```

But there is still a question on how do we compile the decision-making tool cond?A:B;
Someone may think that this is going to be like,

```
BEQ ..., Label
MOV x, x2
B Done
Label:
   MOV x, x1
Done:
```

Well, this is WRONG. The reason is that we want to use the predication for not predicting anything. However, the conversion above still has one conditional branch that we must guess the direction and the target. Because we have two decision-making tools in our program so there will be 2 branches that we need to guess. Let's say if we have a Share predictor, now we are going to have 2 mispredictions. Meanwhile, we still do both paths so there is also a 50% waste. However, if you look back to the C code, you will find that there may be only 1 misprediction without wastes. Therefore, the cond?A:B; can not be converted in that way and we must not add branches to our conversion.

Conditional Move Technique:
So, the technique that we are going to use for a decision-making tool is the conditional move.
For **MIPS**, the conditional move should be,

MOVZ Rd, Rs, Rt

which means,

```
if (Rt == 0) {
   Rd = Rs;
}
```

Or,

MOVN Rd, Rs, Rt

which means,

```
if (Rt != 0) {
   Rd = Rs;
}
```

For x86, we have a whole set of the conditional move (CMOV) instructions such as, CMOVZ (move when zero), CMOVNZ (move when not zero), CMOVGT (move when greater than), etc. The condition is determined by the flags or we called the conditional codes.

Let's see the pseudocode for the decision-making tool x=cond?x1:x2; by MIPS,

```
R3 = cond;
MOV R1, x1
MOV R2, x2
MOVN x, R1, cond
MOVZ x, R2, cond
```

Performance of MOVZ and MOVN
Let's say we have the following program with branches,

```
BEQZ R1, Else
ADDI R2, R2, 1
B End
Else:
   ADDI R3, R3, 1
End:
```

After if-conversion, this should be,

```
ADDI R4, R2, 1
ADDI R5, R3, 1
MOVN R2, R4, R1
MOVZ R3, R5, R1
```

Let's see we have a predictor for the original program with 80% accuracy and there's a 40-instruction penalty if misprediction. According to the code, if the branch is taken, we will have 2 instructions, however, if the branch is not taken, we will have 3 instructions. So, on average, we will have 2.5 instructions. So, the expected overall number of instructions for this program is,

2.5 + (1 - 80%) * 40 = 10.5

If we consider the program after the if-conversion, there will be 4 instructions without branches, so we don't have to worry about the misprediction problem. Thus, the overall number of instructions for this program is 4. And we can find out that the predications actually have better performance.

Requirements for Conditional Move Instruction

For utilizing the conditional move instruction,

- we need the compiler's support
- we will remove the hard-to-predict branches
- we need more registers than the original code because the results from both paths should be kept
- we should execute more instructions because we take both paths and we have to add additional instructions to select the results

Note that we can use the technique to make all the instructions conditional in order to deal with the problems of more registers and additional instructions for result selection. This technique is called a full predication.

Condition Move Vs. Full Predication

Now let's compare what we need for the condition move and what we need for the full predication.

For condition move,

- We need a separate operation code (opcode) to tell us this is a conditional instruction. we usually have a separate opcode for each particular condition, so we need a number of opcodes.

For full predication,

- We add conditional bits to every instruction so every instruction word contains some bits that tell us what is the condition.

Full Predication: An Example

Let's say we have the following program with branches,

```
BEQZ R1, Else
ADDI R2, R2, 1
B End
Else:
   ADDI R3, R3, 1
End:
```

If we convert this program to full predication (i.e. the Itanium), we get

```
MP.EQZ P1, P2, R1    // if R1 == 0, P1 = True, P2 = False
(P2) ADDI R2, R2, 1  // if P2 == True, do the ADDI
(P1) ADDI R3, R3, 1  // if P1 == True, do the ADDI
```

You can easily find out that the code above is much simplified than the if-conversion code. We can also use the same registers and there is now no additional work for moving the results into the destination registers.

INSTRUCTION LEVEL PARALLELISM:

Instruction Level Parallelism (ILP) is used to refer to the architecture in which multiple operations can be performed parallelly in a particular process, with its own set of resources – address space, registers, identifiers, state, program counters. It refers to the compiler design techniques and processors designed to execute operations, like memory load and store, integer addition, float multiplication, in parallel to improve the performance of the processors. Examples of architectures that exploit ILP are VLIWs, Superscalar Architecture.
ILP processors have the same execution hardware as RISC processors. The machines without ILP have complex hardware which is hard to implement. A typical ILP allows multiple-cycle operations to be pipelined.

Example:
Suppose, 4 operations can be carried out in single clock cycle. So there will be 4 functional units, each attached to one of the operations, branch unit, and common register file in the ILP execution hardware. The sub-operations that can be performed by the functional units are Integer ALU, Integer Multiplication, Floating Point Operations, Load, Store. Let the respective latencies be 1, 2, 3, 2, 1.

Let the sequence of instructions be –

1. y1 = x1*1010
2. y2 = x2*1100
3. z1 = y1+0010
4. z2 = y2+0101
5. t1 = t1+1
6. p = q*1000
7. clr = clr+0010
8. r = r+0001

Sequential record of execution vs. Instruction-level Parallel record of execution –

CYCLE	OPERATION
1	y1 = x1*1010
2	nop
3	nop
4	y2 = x2*1100
5	nop
6	nop
7	z1 = y1+0010
8	z2 = y2+0101
9	t1 = t1+1
10	p = q*1000
11	clr = clr+0010
12	r = r+0001

Fig. a

CYCLE	INT ALU	INT ALU	FLOAT ALU	FLOAT ALU
1	t1 = t1+1	clr = clr+0010	y1 = x1*1010	y2 = x2*1100
2	r = r+0001		p = q*1000	
3	nop			
4	z1 = y1+0010	z2 = y2+0101		

Fig. b

Fig. a shows sequential execution of operations.

Fig. b shows use of ILP in improving performance of the processor.

The 'nop's or the 'no operations' in the above diagram are used to show idle time of processor. Since latency of floating-point operations is 3, hence multiplications take 3 cycles and processor has to remain idle for that time period. However, in Fig. b processor can utilize those nop's to execute other operations while previous ones are still being executed.

While in sequential execution, each cycle has only one operation being executed, in processor with ILP, cycle 1 has 4 operations, cycle 2 has 2 operations. In cycle 3 there is 'nop' as the next two operations are dependent on first two multiplication operations. The sequential processor takes 12 cycles to execute 8 operations whereas processor with ILP takes only 4 cycles.

Architecture:

Instruction Level Parallelism is achieved when multiple operations are performed in single cycle, that is done by either executing them simultaneously or by utilizing gaps between two successive operations that is created due to the latencies.

Now, the decision of when to execute an operation depends largely on the compiler rather than hardware. However, extent of compiler's control depends on type of ILP architecture where information regarding parallelism given by compiler to hardware via program varies. The classification of ILP architectures can be done in the following ways –

1. **Sequential Architecture :**
   Here, program is not expected to explicitly convey any information regarding parallelism to hardware, like superscalar architecture.
2. **Dependence Architectures :**
   Here, program explicitly mentions information regarding dependencies between operations like dataflow architecture.

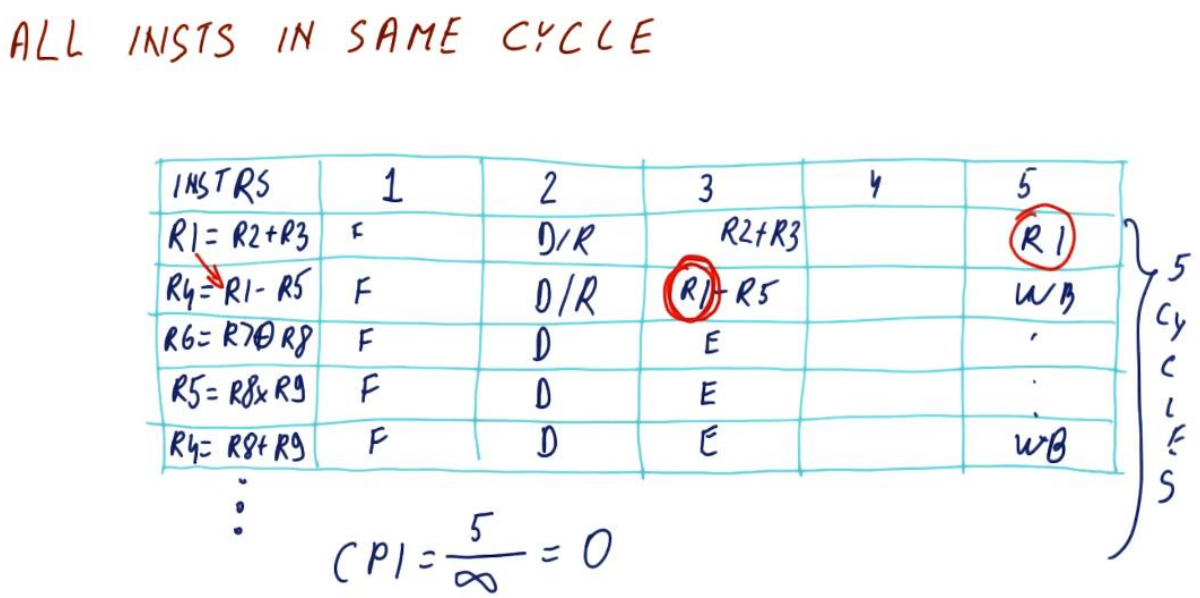
3. **Independence Architecture :**
   Here, program gives information regarding which operations are independent of each other so that they can be executed instead of the 'nop's.

In order to apply ILP, compiler and hardware must determine data dependencies, independent operations, and scheduling of these independent operations, assignment of functional unit, and register to store data.

Parallel instruction execution:

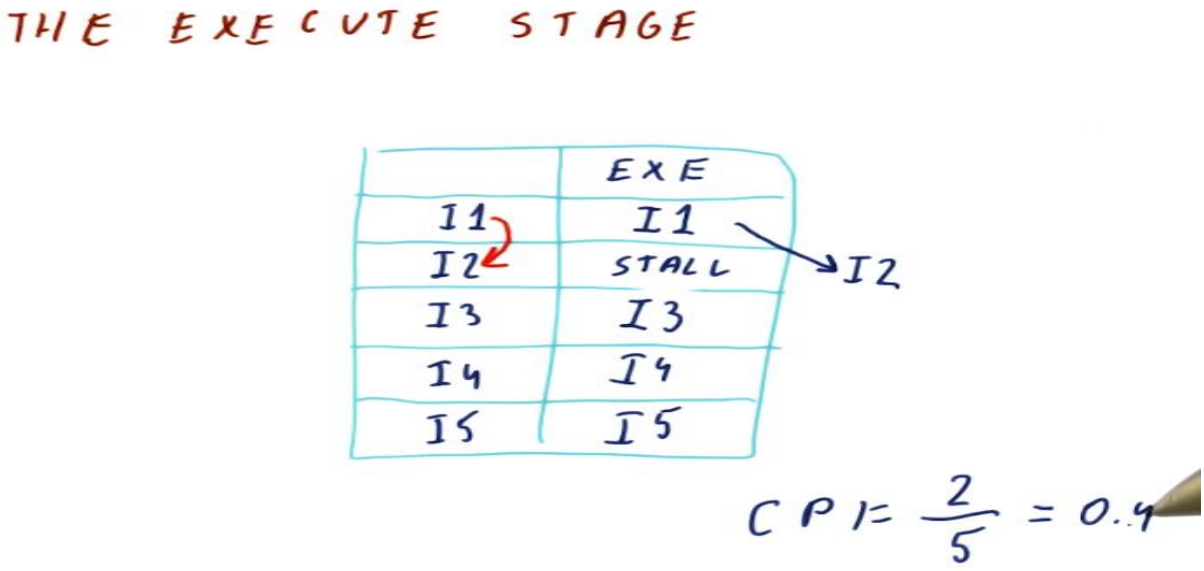
The below excerpt from the lectures showcases the hazards that can manifest due to data dependencies in parallel processor pipelines. As you can see here multiple instructions are destined to be executed and, if all instructions are able to be executed in parallel, it will take 5 cycles to execute a number of instructions. As the number of instructions approaches infinity, our CPI approaches 0.

This isn't realistic - some instructions will depend upon earlier instructions to write values to registers before the instruction is executed. If this isn't done, we will encounter data hazards wherein the registers we use for our instructions will contain stale values, ultimately causing the program to execute incorrectly.



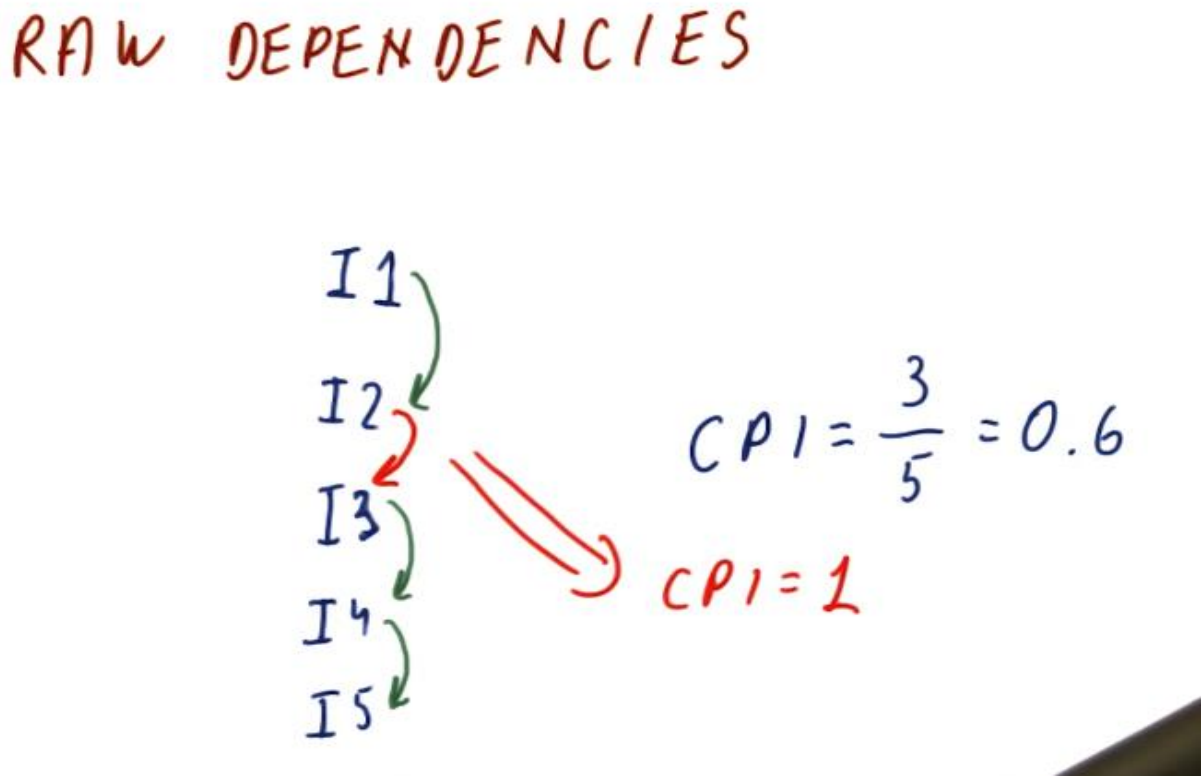
The execute stage:
A technique that can be used to resolve data dependencies and remove hazards. Unfortunately, this technique wont' work for us when we're executing instructions in parallel. The instruction that contains the data dependency will not receive the outcome of the forwarded instruction before it executes because they are executed in parallel.

In order to remove the hazard related to this data dependency, we must stall the instruction until the previous instruction executes. Then, we will forward the result of the executed instruction to the dependent instruction. This will increase our CPI.



RAW dependencies:

how we can calculate the CPI for a set of instructions based upon the number of data dependencies that exists between them. As you can see, the more RAW dependencies that exist, the higher our CPI.

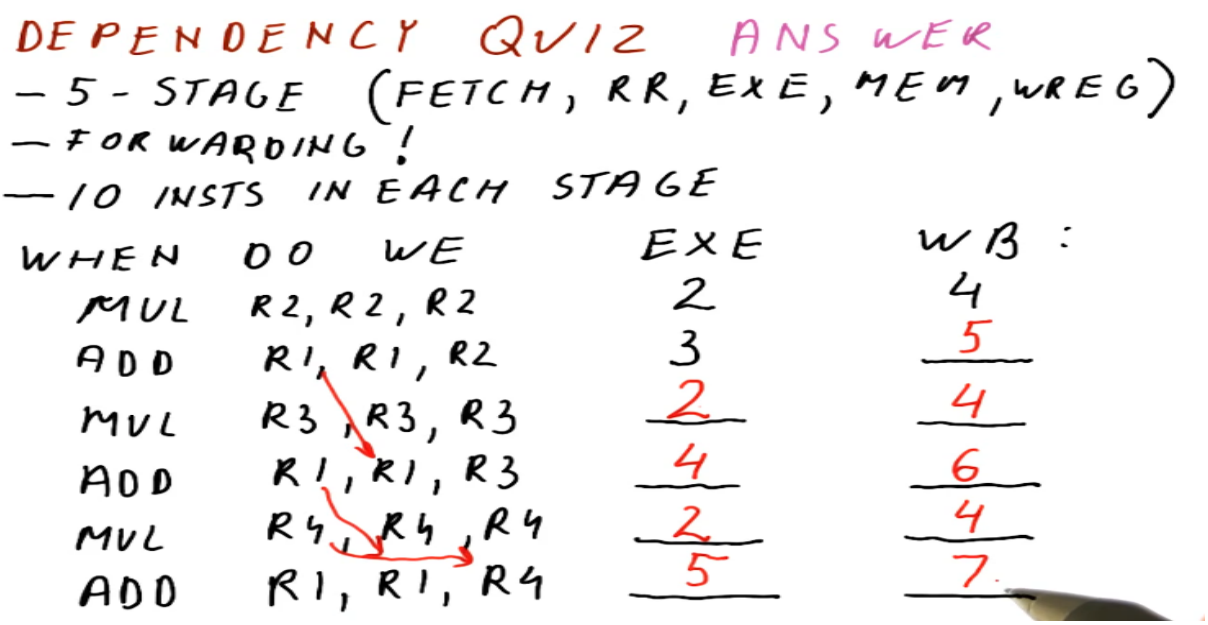


WAW dependencies:

The below excerpt from the lectures showcases how a WAW dependency can manifest due to a RAW dependency in previous instructions. The WAW dependency poses a hazard because the previous instruction is stalled due to a RAW dependency. If this goes ignored, the dependent instruction in the WAW dependency will write to a register before the previous write, executing instructions out of order. In order to solve this, the dependent instruction must be stalled twice so that the previous instruction can write.

Dependency Quiz:

Below is a quiz from the lectures in the class that showcases how multiple RAW and WAW dependencies are handled in the processor pipeline.



DUBLICATING REGISTER VALUES:

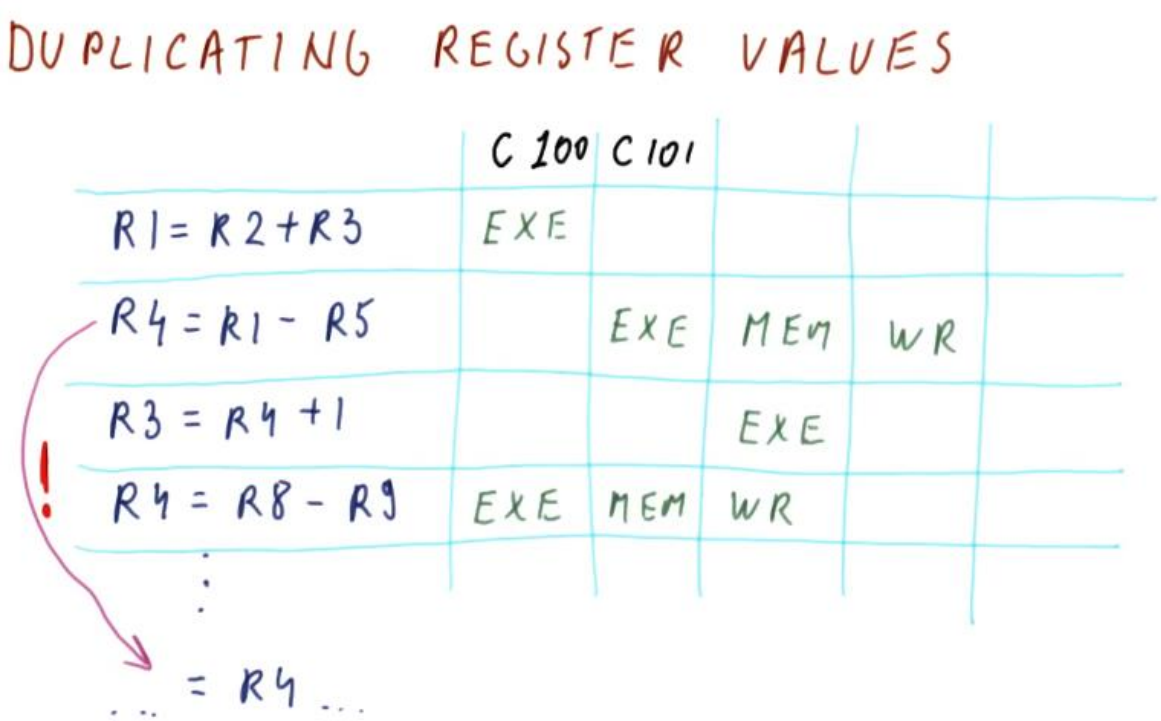
A true dependency is the RAW (read after write) dependency - pretty obvious because this is how the program is actually intended to execute. Both the WAR (write after read) and WAW (write after write) dependencies are false dependencies. In both of these dependencies, we are re-using registers for instructions when we could change the registers being used to eliminate the false dependencies completely.

Duplicating register values:

This is a complicated technique in which we save multiple **versions** of a value for a register that is being written to. In the excerpt below, two different writes occur for the register: R4. This register is used in two different read operations, but what value will be used if the register is written twice?

To remove the hazard posed by this WAW dependency, we store both versions of R4 and then each read operation will utilize the appropriate version based upon the chronological order of the writes. The third instruction in this example will use the outcome of the write in instruction two for its read operation, even though instruction four writes to R4 before instruction two does. A future instruction will use the most previous version of R4 for its read operation, even though the write by instruction two occurs after the write by instruction four.

Essentially, we are keeping track of all possible versions of a register in a parallel pipeline and different read operations that require the value of register will use the value generated by the most recent write operation.

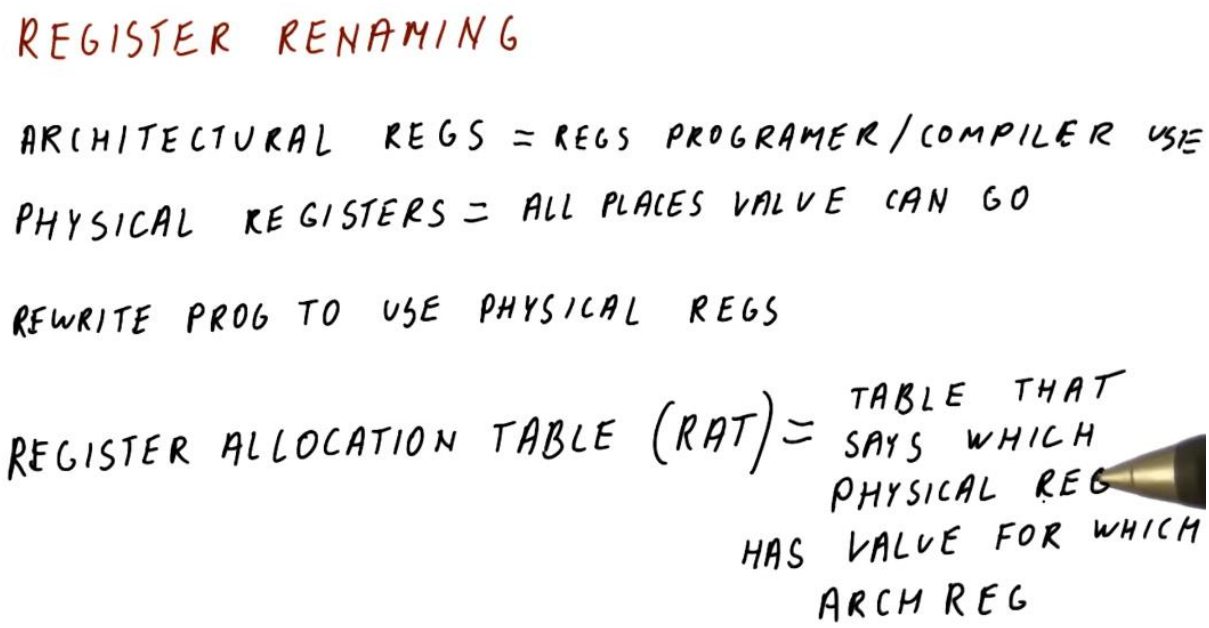


Register renaming:
So, duplicating and managing register values can get pretty complicated and time consuming. Another technique processors can use to remove hazards created by data dependencies is **register renaming**.

In the below excerpt from the lectures, we define some terms:

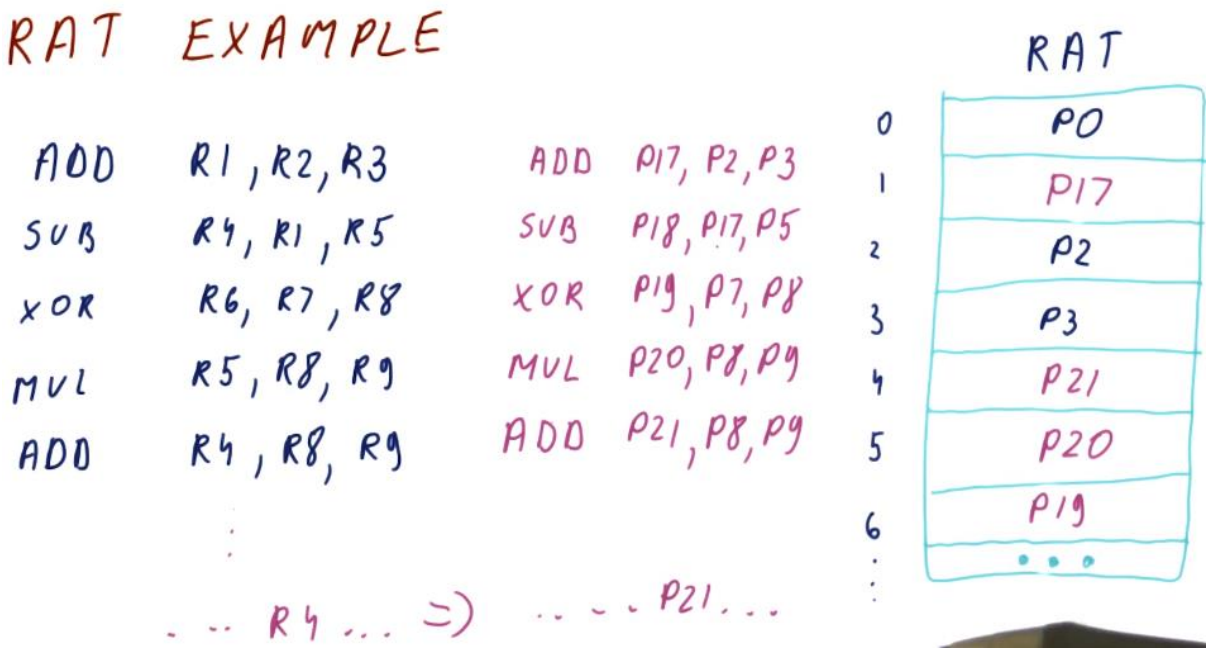
- **architectural registers** - these are register names used by the programmer and the compiler.
- **physical registers** - these are the physical registers available to the processor to actually store real values.
- **register allocation table** - this is a table used by the processor to translate architectural register names to physical registers. This table defines which physical register contains the data for a specified architectural register.

The processor rewrites the program it's executing to use the physical registers. Through register renaming, it will have more registers available to store values and avoid hazards presented by data dependencies.



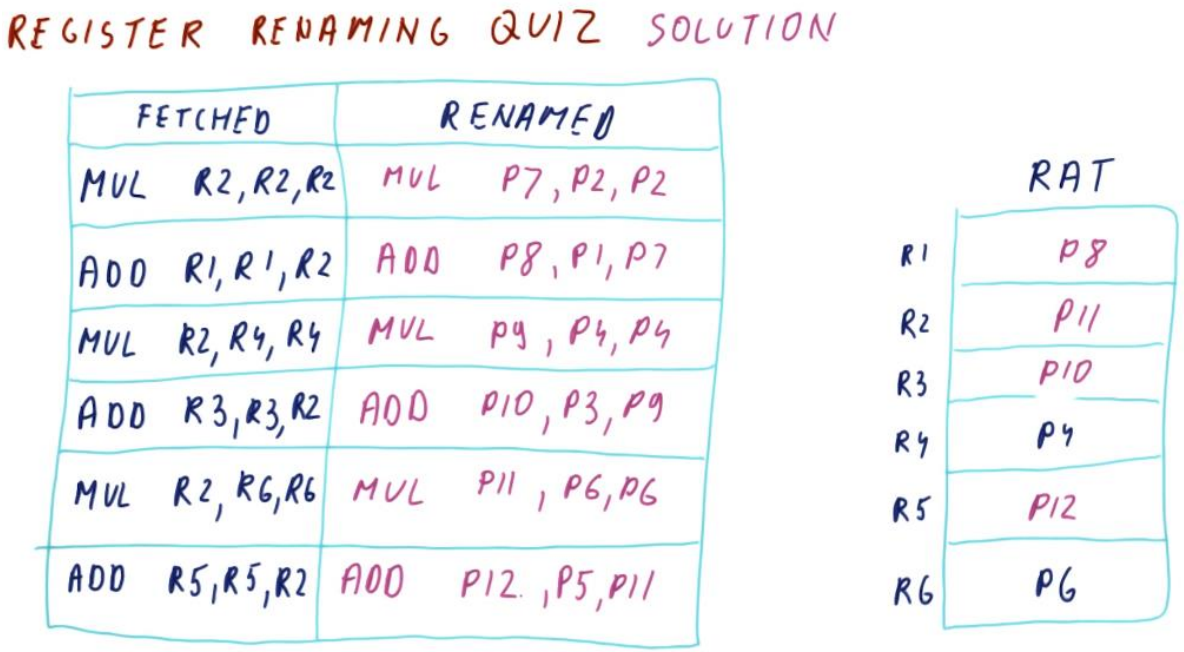
RAT example:
The below excerpt from the class showcases how a RAT works. Each time a new value is produced, a different physical register is used to store that value. Values for registers that are being read are acquired from the RAT after it translates the architectural name to the physical register. We call the WAW and WAR dependencies name dependencies because the name is the whole problem. If we rename the register that the value is being written to for each write, the instructions won't overwrite the work of the other instructions if they're executing in parallel.

In this example, we can see that each write increments the physical register number that's being written to. This removes the hazard of a register being overwritten by a later instruction before an earlier instruction can conduct its write. Because the writes are to different physical registers, when an instruction is decoded and needs to read a register, the value it's reading is going to be from the instruction prior, not the most recent write to the register.



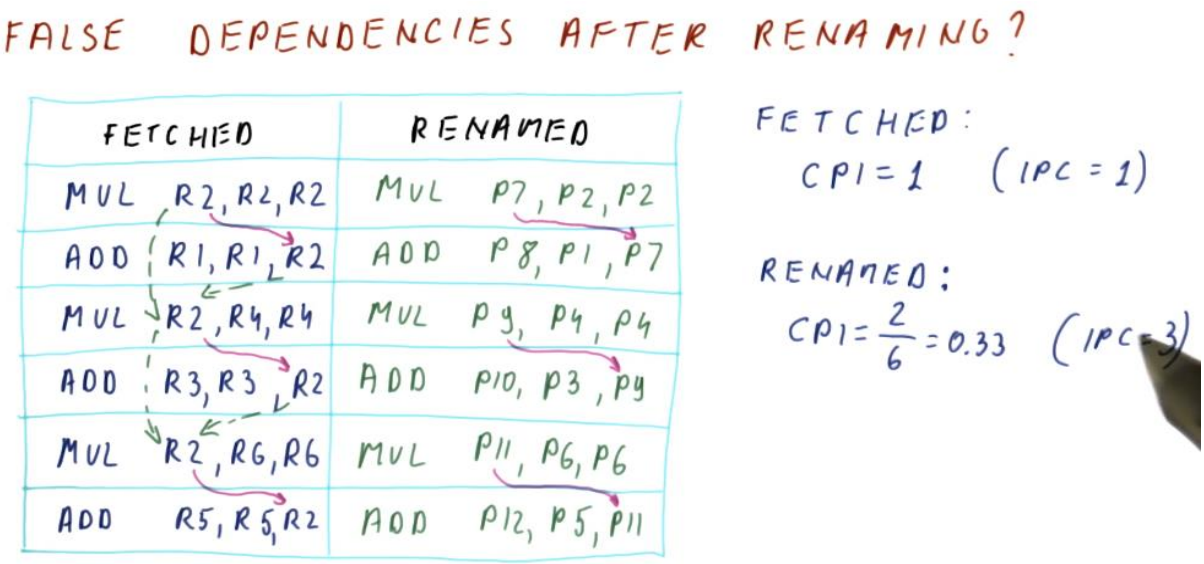
Register renaming quiz:

Below is a quiz for register renaming



False dependencies after renaming:

So, does renaming actually remove our WAR and WAW dependencies? Yes. we are shown a program that contains a set of dependencies for each instruction. Because each instruction has these dependencies, they must be executed in order, causing us to have a CPI of 1. However, with renaming, we are able to eliminate the WAR and WAW dependencies improving our performance to a CPI of 0.33.



ILP:

Instruction level parallelism (ILP) is a property of a program given the fact that it's running on an *ideal processor*. What's an ideal processor? An ideal processor has these attributes:

- Processor dispatches an entire instruction in 1 cycle.
- Processor can do any number of instructions in the same cycle.
    - The processor has to obey true dependencies when doing this.

So what are the steps to acquire the value of a program's ILP? A program's ILP is equal to the IPC when executing on an ideal processor following the rules stated above. Obviously ideal processors like this always aren't achievable, the ILP for a program will be different on actual real-world processors. ILP gives us a value, however, for the parallel nature of program - defining how many true dependencies exist within the code.

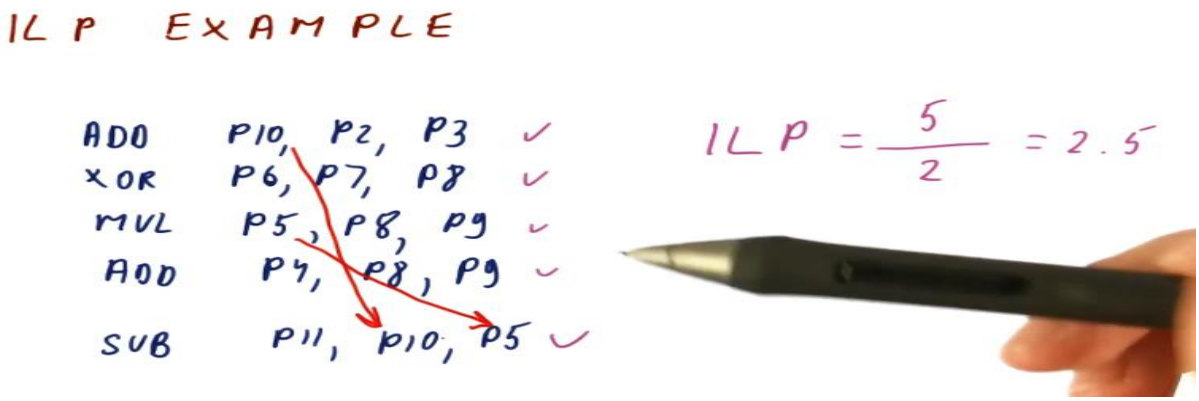
The steps to acquire the ILP of a program are:

1. Rename the registers, as shown previously.
2. "Execute" the code on the ideal processor.

ILP example

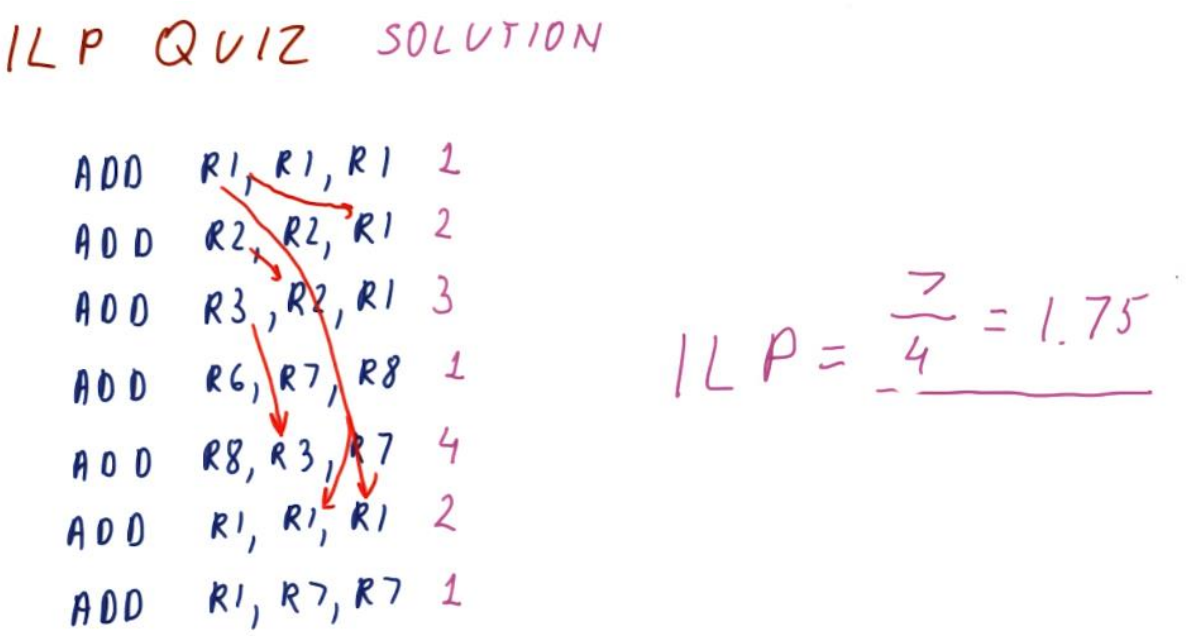
The below define how we can compute the ILP for a given set of instructions: ILP == num instructions/cycles. In this example, the professor identifies the true dependencies that exists for the set of instructions and sees that the fifth instruction cannot execute until the writes of instruction one and three are complete. Thus, this program will take 2 cycles on an ideal processor to execute.

A neat trick that is described in the lecture is that we don't even have to conduct register renaming in order to calculate the ILP. All we have to do is identify the true dependencies - the register renaming will take care of the false or name dependencies for us.



ILP quiz:

we just identify the true dependencies to determine on what cycle instructions are eligible to execute. After we determine the minimum number of cycles to resolve the true dependencies, we can find the ILP of the program.



Calculating ILP with structural and control dependencies

When calculating ILP, we don't account for structural dependencies. These dependencies result for architectural issues like not having an adder available for a specific operation, etc. Something outside of the programmer's control, solely relies with the manufacturing of the processor.

For control dependencies, we essentially ignore them. We assume that we have perfect same-cycle branch prediction. In the example below, we show that a branch instruction has a data dependency on instructions prior to it, however, because we have perfect branch prediction we know that we will jump to the Label. Thus, we fetch the instructions from the Label and execute them, even though our branch has not been executed, yet.