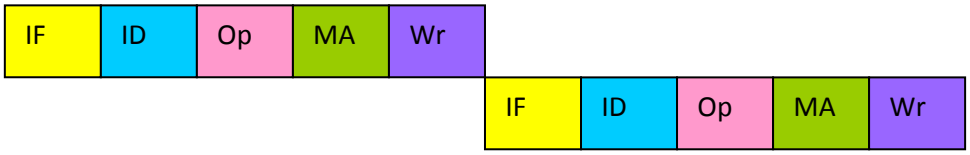
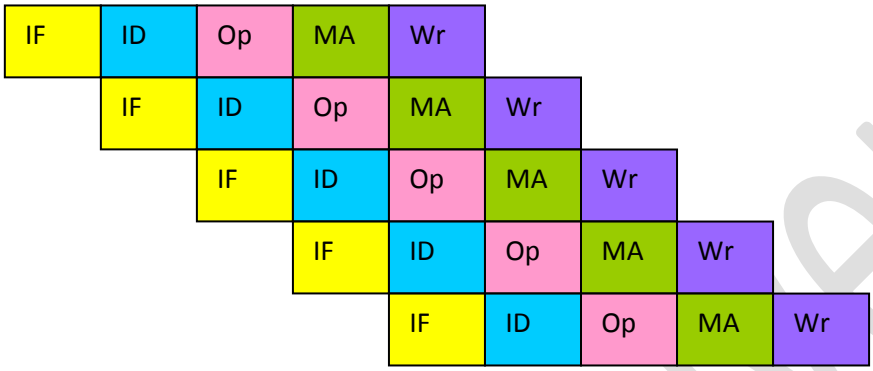
Pipelining

Pipelining

Compare executing one instruction at a time:



Versus Pipelining:



Five Pipeline Stages:

- IF (Instruction Fetch): Fetch instruction from memory
- ID (Instruction Decode): Read Register(s) while Decoding Instruction
- Op (ALU Operation): ALU operation or calculate memory address
- MA (Memory Access): Access an operand in Memory (Memory Load/Save)
- Wr (Register Writeback): Write the Loaded memory into a register

Each pipeline stage takes one clock cycle

- Clock cycle must be long enough to accommodate slowest operation. Examples:
  - Memory access: 200 ps
  - ALU operation: 200 ps
  - Register read/write: 100 ps
  - Therefore Cycle time = 200 ps

TimePerPipeInstruction = TimePerInstruction / #ofPipeStages

Speedup approaches the number of pipe stages

- **Throughput** = Instructions executed per time period (e.g., seconds)
- Piplelining increases instruction throughput
- Instruction execution time (or **Response Time**) does not shorten

Holding registers exist between each pipeline stage:

- What would we need to hold between each stage?

GNA University

Structural Hazards

Structural Hazards: In some cases pipelining must stall or finagle due to required delays.
Eg. Load takes an extra long time

Data Hazard

Data is generated in earlier instructions and used in later instructions

Example 1: Read after compute

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

Here: $s0 is generated in ALU Op stage in add and used in ALU Op stage in next instruction
Solution: Move ALU Op output to next instruction X input

- **Forward** the result to a later instruction, or
- **Bypass** the register file to the expected input

Example 2: Read after load

```
lw      $s0, index
addi$s0,$s0,1
```

Here $s0 is generated in Mem Access stage and used in ALU Op stage of next instruction
Problem: If access is to memory instead of cache, load can be particularly long
Solution: **Pipeline Stall**: Wait until loaded data becomes available before ALU Op stage

- **Bubble = No-operation** = stall the next instruction

Solution Implementation: Hardware and/or Software

- Software: Compiler inserts NOP (no-operation) to avoid data hazard
- Software: Compiler rearranges instructions to avoid data hazard
- Hardware: Pipeline interlock: Bubble insertion or data forwarding avoid data hazards

Control Hazard

A earlier instruction makes a decision impacting later instructions

- Branches are 13% of instruction in SPECint2000.
- How can **loop unrolling** ensure fast execution?

Example:

```
Label:  addi    $s3,$s3,1
        slti    $t0,$s3,25
        bne     $t0,$zero,Label
        sw      $s3,index
```

Solution 1: Do branch test in Inst Decode stage. This results in a one-stage delay if branch taken
Problem: If > 5 pipelined stages, may not work.
Flush instructions if branch not correctly predicted

Solution 2: **Delayed Branch**. Always execute instruction after the branch.
Solution used by real MIPS hardware

```
Label:  slti    $t0,$s3,24
        bne     $t0,$zero,Label
        addi    $s3,$s3,1
        sw      $s3,index
```

Solution 3: Predict result. When answer correct, proceed at full speed. Stall otherwise
Static branch prediction: Take backward branches; do not take forward branches
Dynamic branch prediction: Keep history of each branch and follow recent behavior.
Example 1:

If last conditional branch taken, then

GNA University

take next conditional branch
Else do not take conditional branch
Example 2: Hash table indexed by instruction address indicates branch history

Deep Pipeline: Each stage is divided into multiple stages in order to increase the number of instructions that can be executed simultaneously and reduce cycle time.
- Used by Pentium 4.

Multiple Issue Architectures

Perform multiple instructions in every pipeline stage
IPC: Instructions per clock cycle
IPC used when CPI < 1
CPI = 0.25 => IPC = 4
IPC = 1/CPI

Issue Packet: Set of instructions that execute together in one clock cycle
Also known as **Instruction Group**
Can have no register data dependencies between instructions in group

Example:
What does the following code do??? Each row of instructions execute simultaneously.

	Arithmetic ALU	**Access Memory ALU**
Loop:	addi $s1,$s1,4	lw $t1, 0($s2)
	addi $s2,$s2,4	lw $t0, -4($s1)
	addu $t0,$t0,$t1	
	bne $s2,$s4,Loop	sw $t0, 0($s1)

What is the IPC for the above code?

Static Multiple Issue: Compiler determines sets of instructions to execute together
- Also known as **VLIW: Very Long Instruction Word**
- Instructions are paired, aligned on a 64-word boundary, and issued together.

Example: Intel **IA-64** Itanium I and II
- A stop between two instructions indicate that they below to different groups
- Each instruction bundle includes:
  - Up to 3 instructions of 41 bits each, and
  - A 5-bit template field indicating the type of execution units to use:
  - Five execution units include:
    - Integer ALU,
    - Non-integer ALU (shifts, multimedia operations)
    - Memory access
    - Floating point unit
    - Branch unit

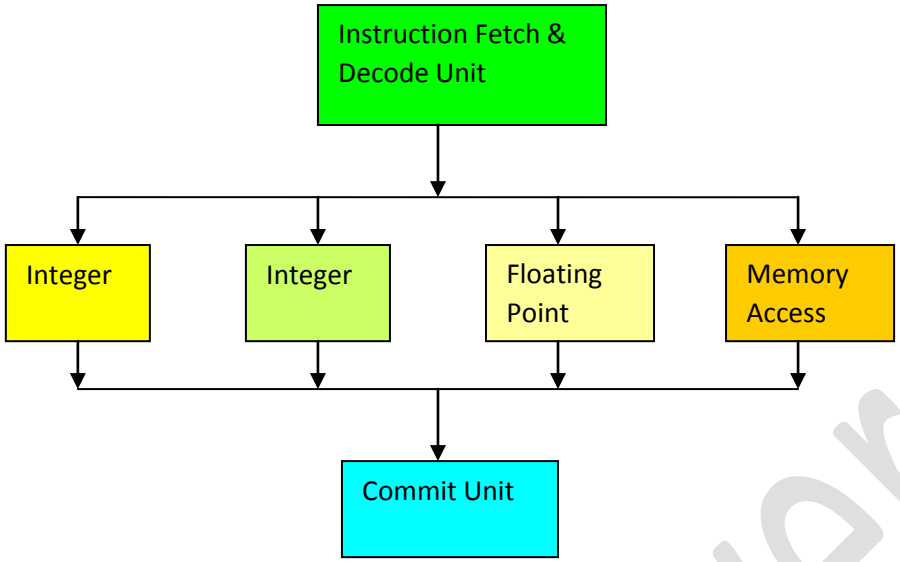
Branch Predication uses Predicate
- Both Then and Else clauses can be loaded simultaneously
- A Predicate flag indicates if instruction should be counted or not
  - If (predicate) statement 1
  - Else (~predicate) statement 2
  - When the condition is decided, the predicated statements can complete

Dynamic Multiple-Issue: Hardware decides dynamically if instructions can execute together
- Also known as **Superscalar** or Dynamic Pipeline Scheduling
- Advantages over static multiple issue:
  - Cache misses may cause unexpected delays
  - Speculation on dynamic branch prediction allows multiple simultaneous executions

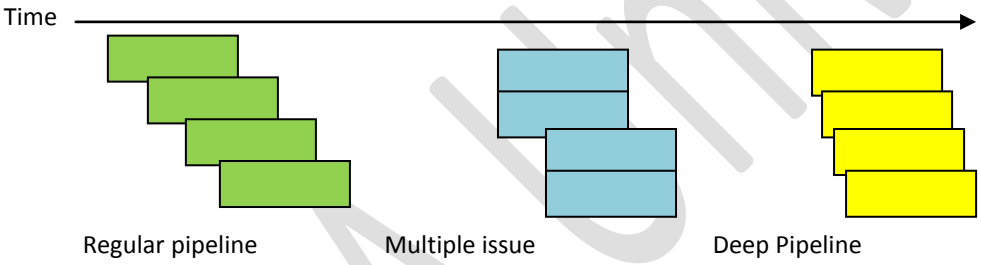
GNA University

Pipeline divided into three major units or stages:

- Instruction Fetch and Decode Unit: Instructions are fetched and decoded in order
- Multiple Functional Units: Instructions occur out-of-order if correct data is available. Requires:
  - Instruction A output fed to Instruction B input, when necessary
  - Predicate conditions are implemented correctly
- Commit Unit: Instructions must commit in order (written to programmer-visible state)



Today all high-end processors use multiple issue



GNA University

Pipeline Exercise:

1) Consider the following code:

```
main:
#                   Register conventions:
#                   S3 = input array
    la      $s3,InArr
#                   S4 = output array
    la     $s4,OutArr

#                   do 5 times
    addi   $t0,$zero,0
    addi   $t5,$zero,5
Loop:
#                       output[index++] = input[index++]
    lw     $t1,0($s3)
    sw     $t1,0($s4)
    addi   $s3,$s3,4
    addi   $s4,$s4,4
    addi   $t5,$t5,-1
#                   enddo
    bne    $t0,$t5,Loop
    nop
#                   return
    jr     $ra
```

Assume the following architecture specifics:

Delayed branch: Instruction after branch always occurs.
Stall or Bubble: Occurs if ALU generates data which needs to be used in the next cycle.
Forwarding or Bypass: Ensures results get to next instruction as fast as possible.

- No Bypass: result register must be stored in reg file before it can be used
  - Register Writeback: stores to register file in first half of clock cycle
  - Instruction Decode: reads from register file in second half of clock cycle.
  - Two extra cycles are required (2 stalls)
- Bypass is available: Result register is provided to any stages needed directly.
  - If result is available after ALU Operation: (e.g., no data cache access)
    - ALU output fed to ALU input
    - Zero extra cycles are required.
  - For load word (lw) result, available after Completion 1 stage
    - One stall is required for next instruction ALU input
    - Zero stalls are required for Mem Access input (i.e., sw)

GNA University

1) Complete the following table as much as possible.

Time	Instruction Fetch	Instruction Decode	ALU Operation	Memory Access	Register Writeback
Time 0	PC=0->4	Instr=	Rs= Rt/Imm= Rd=	ALUout= Rt= Rd=	Data cache= StoreReg= Rd=
Time 1	PC=4->8	Instr= la $s3, InArr (lui)	Rs= Rt/Imm= Rd=	ALUout= Rt= Rd=	Data cache= StoreReg= Rd=
Time 2	PC=	Instr= la $s4,OutArr (lui)	Rs=$0 Rt/Imm=InArr Rd=$s3	ALUout= Rt= Rd=	Data cache= StoreReg= Rd=
Time 3	PC=	Instr=	Rs=$0 Rt/Imm=OutArr Rd=$s4	ALUout= InArr+0 Rt= Rd=$s3	Data cache= StoreReg= Rd=
Time 4	PC=	Instr=	Rs= Rt/Imm= Rd=	ALUout= OutArr+0 Rt= Rd=$s4	Data cache= StoreReg= 0x10010000 Rd=$s3
Time 5	PC=	Instr=	Rs= Rt/Imm= Rd=	ALUout= Rt= Rd=	Datat cache= StoreReg= 0x10010020 Rd=$s4
Time 6	PC=	Instr=	Rs= Rt/Imm= Rd=	ALUout= Rt= Rd=	Data cache= StoreReg= Rd=

2) Perfect World: Calculate the time to execute this code in a perfect world assuming no stalls. How many cycles would be required? Assume no delayed branch implementation.

3) Real World: Show dependencies between instructions.

- Circle the registers where dependencies exist.
- Indicate where a stall is required for the current code. Indicate the number of stalls that are required for any instruction requiring a stall, on the front page of this exercise.
- Indicate the total number of cycles to process the full code in as-is condition. Assume delayed branch is implemented.

	No Bypass	Bypass is Available
Number of Cycles Required		

4) Assume you are developing a compiler which will reorder instructions to optimize implementation. Use arrows on the first page to show the order of the instructions that you would implement. Indicate the total number of cycles to execute the code, assuming no bypass. Can you get it to the perfect number you had in question 2?

GNA University

5) With Static Multiple Issue, we execute two instructions simultaneously via 2 simultaneous ALUs. How would you use Static Multiple Issue by pairing up instructions to execute this code faster? Assume one ALU can process load/store and one ALU processes arithmetic (or non-memory including branch) operations. Be careful that you don't change the results of the registers, and give a wrong answer! Show which instructions can be matched together. How many cycles are required, assuming one cycle per row?

Memory Access ALU (e.g., lw/sw/la)	Arithmetic ALU (e.g., add, addi, bne)

GNA University

Branch Prediction

Consider the effectiveness of branch prediction for the following two prediction algorithms on two sets of conditions. The two conditions include:

A) A loop occurs for 5 times before completing

B) A loop occurs for 5 times before completing. An if statement in the loop alternates between if and else conditions, based on incrementing an indexing and testing for odd results.

What is the probability of correct guessing in each case, assuming that the first prediction is that the branch is TAKEN? Unconditional branches do not affect prediction memory.

```
index = 0;
do {
        if (index++ == odd)   (beq)
                Do odd;
        else                          (b)
                Do even;
        endif
} while (index < 5);          (beq)
```

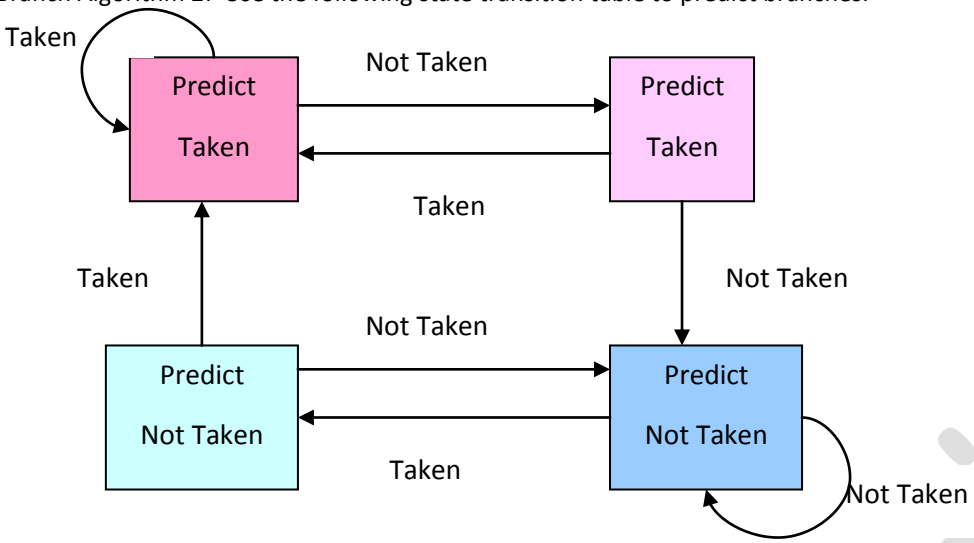
1) Branch Algorithm 1: Use the previous branch result as the current prediction. (If previous branch occurred, assume it will again.)

1A)

1B)

GNA University

2) Branch Algorithm 2: Use the following state transition table to predict branches:



Example: For the above, assume that you start in Predict Taken 1:
Take: Predict Taken 1
Not Take: Predict Taken 2
Not Take: Predict Not Taken 1
Not Take: Predict Not Taken 2

2A)

2B)

In your own words, how does the above state diagram make decisions about prediction?

GNA University