**cost of IC**

**amdahls law**

the performance improvement to be gained form faster mode of exec is limited by the amount of time the faster mode can be used

speedup from enhancem depends on two factors

1. Fraction of time that can be enhanced or fraction enhanced which is always less than one
2. Improvement gained by **enhanced execution mode** or speedup enhanced

The **law of diminishing returns** (also **law of diminishing marginal returns** or **law of increasing relative cost**) states that in all productive processes, adding more of one factor of production, while holding all others constant ("[*ceteris paribus*](http://en.wikipedia.org/wiki/Ceteris_paribus)"), will at some point yield lower per-unit returns.

**Processor performance equation**

Clock cycle time—Hardware technology and organization

CPI—Organization and instruction set architecture

Instruction count—Instruction set architecture and compiler technology

**Instruction level parallelism**

Pipeline cpi= ideal pl cpi+ structural stall+ data hazard stalls+control stalls

**Loop level parallelism**

Vector instructions

**Dependences**

1. Data
2. Name
3. Control

**Data dependence**

1. I produces results the j uses (I and J are consecutive instructions)
2. Chain dependence

A processor can avoid overlap or simultaneous exec of two depen I’s if it has pl interlocks

We can use scheduling (s & d ) to avoid hazards but there will be dependence

**Name dependences**

When two I’s use same memory location of regs but there is no data flow btw them

1. **Antidependence** two ins I and J are antidependent if j writes to reg or mem loc where I reads from. The order should be preserved so that I reads correct value
2. **Output dependence** if two ins I and j write to same mem loc or reg. order should be maintained so that final value is of j

Name depen I’s can be reordered and execute simultaneously if we do register renaming

**Structural Hazards**

If some combination of instructions cannot be accommodated because of a resource conflict, the machine is said to have a structural hazard.

Common instances of structural hazards arise when

http://www.cs.iastate.edu/~prabhu/images/blueball_1.gifSome functional unit is not fully pipelined. Then a sequence of instructions using that unpipelined unit cannot proceed at the rate of one per clock cycle   
http://www.cs.iastate.edu/~prabhu/images/blueball_1.gifSome resource has not been duplicated enough to allow all combinations of instructions in the pipeline to execute.

**Data hazards**

Hazard occurs when there is depend.bcoz f depend there may be chage in order of access to operands. This may lead to stall.

1. **Raw** j tries to read value b4 I writes it
2. **Waw** j tries to write b4 I writes it (**output dependence)**
3. **War** j tries to wr a value b4 I reads it

**Control dependences(branch)**

1. I dat z control dependent on branch should not be moved b4 it
2. I dat z not depen on branch should not be moved after it

**Properties critical to pgrm correctness and preserved by data and control dependence are program flow and exceptional behavior**

**Compilers ability to pl depends on ilp avail in prgm and func unit latencies**

**Loop unrolling**

Loop unrolling increases number of I’s relative to branch and overhead I’s. unrolling replicates loops body multiple times changing the termination condition

Loop unrolling can be used to improve scheduling.

It eliminates branch so it allows I’s from dfrnt iters to be scheduled together

If we replicate I’s when we unrolled loop then use of same registers would stop us from effectively scheduling

Thus we have to use dfrnt regs for dfrnt iterations.this increases req no of regs

Loop unrolling increases the size of the code

The gain from scheduling the unrolled loop is larger cause we can schedule the unrolled I’s to reduce the stalls

This scheduling necessitates realizing that loads and stores are independent and can be interchanged

3 limitations to loop unrolling

Decrease in overhead

Increase in size results in register shortfall called register pressure

It is caused because scheduling to increase ILP causes to increa live values

Compiler limitations

**Reducing branch costs with prediction**

Static branch prediction and dynamic

One static way is to just predict branch as taken or we may use profile info

One way is to use profile info from earlier runs

**Dynamic branch prediction**

Branch prediction buffer

Indexed by lower portion of address of branch instruction

A bit tells that whether the branch is taken recently or not

2-bit prediction(prediction must miss twice to change it) accuracy is 99 to 82%

**Correlating branch predictors or two level predictors**

Checks the other branches also along with this branch

Hw is simple coz shiftregister is enough

**Tournament predictors**

Uses to predictors

One based on local and other on global

They use 2 bit saturation counter to choose among local and global

**Limitation** wth smple pl is that it uses in order issue and execution

**Dynamic**

Both structural and data hazards can b checked during ID stage . and I is issued from ID when all the hazards have been cleared

To do this issue process is divided into two. Check for structural hazards and wait till absence of data hazards

The issuing is still in order but executions and thus completion is out of order

Out of order exec introduces possibility of war and raw hazards and also causes complications in exception behavior

DS may intro imprecise exceptions.an excep is imp when the state of Prcr is not exactly same as if the I’s were executed in strict program order

**Reasons.**

The pipeline may have already completed the I’s that are later in program order than the I’s causing exception

The ppl might not yet have cmpltd I’s that are befor the I’s that cause exception in program order

To allow out of oder exception id stage is divided into two stages

Issue-decode I’s and look for structural hazards

Read operands-wait till no data hazards and read operands

In a dynamically scheduled pipeline, all instructions pass through the issue

stage in order (in-order issue); however, they can be stalled or bypass each other

in the second stage (read operands) and thus enter execution out of order

**tomasulo**

This scheme, invented by Robert Tomasulo, tracks when operands

for instructions are available, to minimize RAW hazards, and introduces

register renaming, to minimize WAW and WAR hazards

*Register renaming* eliminates

these hazards by renaming all destination registers, including those with a pending

read or write for an earlier instruction, so that the out-of-order write does not

affect any instructions that depend on an earlier value of an operand.

Register renaming is done by reservation stations.they buffer operand of I’s that are waiting to issue. the reservation stations fetch and buffer operands as soon as they are available eliminating the need to get it from register.RS are designated by pending Is. Finally, when successive writes to a register overlap in execution, only the last one is actually used to update the register. As instructions are issued, the register specifiers foru

pending operands are renamed to the names of the reservation station, which provides

register renaming.

As rs are more than regs it can solve name dependences.

**Steps in tomasulo**

**Issue**:if der is matching RS issue the I to station with operand values if they are in R.if der z no RS den der is structural hazard den da I stalls util the station or buffer is freed.if they r not in Reg’s keep track of fnal units that vl produce the oprands.dis step eliminates war and waw hs by renaming R

**Execute** if one or more operands is not available monitor the common data bus while waiting for it to be computed.when an operand becomes available it is placed into any reservation station awaiting it.when all the prrands are avail the op can be xec at fnal unit.by delaying I until oprands r avail v eliminate raw hazards.

Loads and stores are two step p. 1st step is calculations of eff addr as soon as base reg is avail and eff addr is store in load or store buff.loads and stores are maintained in program order by computing their eff addr.

To preserve exception behavior no I is allowed to execture until all the branches before the I have been executed.

**Write result**

Wen result is avail wr it on cdb and frm der into regs and into reservation stations

**Scoreboarding**

*Scoreboarding*

is a technique for allowing instructions to execute out of order when there are

sufficient resources and no data dependences;

The goal of a scoreboard is to maintain an execution rate of one instruction

per clock cycle (when there are no structural hazards) by executing an instruction

as early as possible.

The scoreboard takes full responsibility for instruction issue

and execution, including all hazard detection. Taking advantage of out-of-order

execution requires multiple instructions to be in their EX stage simultaneously.

This can be achieved with multiple functional units, with pipelined functional

units, or with both. Since these two capabilities—pipelined functional units and

multiple functional units—are essentially equivalent for the purposes of pipeline

control, we will assume the processor has multiple functional units.

Every instruction goes through the scoreboard, where a record of the data

dependences is constructed; The scoreboard then determines when

the instruction can read its operands and begin execution. If the scoreboard

decides the instruction cannot execute immediately, it monitors every change in

the hardware and decides when the instruction can execute. The scoreboard also

controls when an instruction can write its result into the destination register.

Thus, all hazard detection and resolution is centralized in the scoreboard

Each instruction undergoes four steps in executing.

1. *Issue*—If a functional unit for the instruction is free and no other active

instruction has the same destination register, the scoreboard issues the

instruction to the functional unit and updates its internal data structure.

By ensuring that no other active functional unit wants to write its result into the destination

register, we guarantee that WAW hazards cannot be present. If a structural or

WAW hazard exists, then the instruction issue stalls, and no further instructions

will issue until these hazards are cleared

2. *Read operands*—The scoreboard monitors the availability of the source operands.

A source operand is available if no earlier issued active instruction is

going to write it. When the source operands are available, the scoreboard tells

the functional unit to proceed to read the operands from the registers and

begin execution. The scoreboard resolves RAW hazards dynamically in this

step, and instructions may be sent into execution out of order. This step,

together with issue, completes the function of the ID step in the simple MIPS

pipeline.

*Execution*—The functional unit begins execution upon receiving operands.

When the result is ready, it notifies the scoreboard that it has completed

execution.

*Write result*—Once the scoreboard is aware that the functional unit has completed

execution, the scoreboard checks for WAR hazards and stalls the completing

instruction, if necessary.