ADVANCED COMPUTER ARCHITECTURES

Claudio Tessa

January 5, 2026

Contents

1 Pipelining: basic concepts

Pipelining is an implementation technique where multiple instructions are overlapped in execution. Like an assembly line, every step of the pipeline completes a part of an instruction.

1.1 RISC-V instruction set

RISC architectures are characterized by a few key **properties**:

- All operations on data apply to data in registers and change the entire register.
- The only operations that affect memory are load and store operations.
- The instruction formats are few in number.

These simple properties lead to dramatic simplification in the implementation of the pipelining.

There are three classes of instructions:

1. **ALU instructions** - These instructions take two registers (or a register and a sign-extended immediate), operate on them, and store the result into a third register.
   - `add rd, rs1, rs2` $\iff$ `rd = rs1 + rs2`
   - `addi rd, rs1, 4` $\iff$ `rd = rs1 + 4`
2. **Load and store instructions** - These instructions take a register source and an immediate, called *offset*. The sum of the content of the source register and the offset is used as the memory address. Another register is also taken to performing the operation.
   - `ld rd, offset (rs1)` $\iff$ `rd = Memory[rs1 + offset]`
   - `sd rs2, offset (rs1)` $\iff$ `Memory[rs1 + offset] = rs2`
3. **Branches and Jumps** - Branches are conditional transfers of control, while jumps are unconditional.
   - `beq rs1, rs2, L1` $\iff$ `if (rs1 == rs2) then go to L1`
   - `j L1` $\iff$ `go to L1`

1.2 Phases of execution of RISC-V instructions

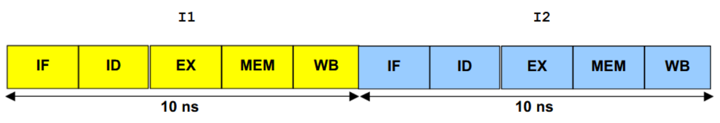
Every instruction takes at most 5 clock cycles. The 5 clock cycles are as follows:

1. **IF (Instruction Fetch)** - Send the program counter (PC) to memory and fetch the current instruction from memory. Update the PC to the next sequential PC by, adding 4 as every instruction is 4 bytes long in memory.
2. **ID (Instruction Decode)** - Decode the instruction and read the registers specified in the instruction. Decoding is done in parallel with reading registers.
3. **EX (execution)** - The ALU operates on the operands prepared in the previous cycle.

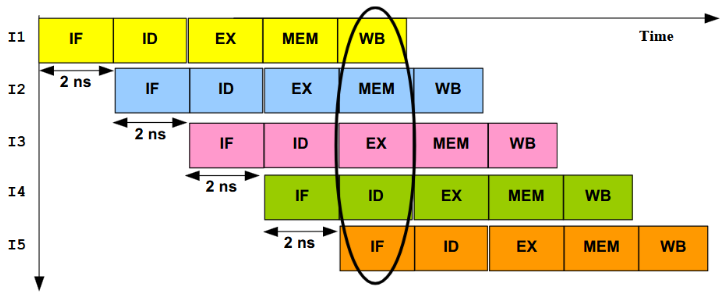
4. **MEM (Memory Access)** - Based on the instruction, the memory writes or reads data using the effective address computed in the previous cycle.
5. **WB (Write Back)** - Register-Register ALU instruction or lead instruction

1.3 RISC-V Pipelining

Instead of executing every instruction sequentially, as follows:



we can pipeline the execution by simply starting a new instruction on each clock cycle:



Each of the clock cycles now becomes a *pipe stage*. Although each instruction takes 5 clock cycles to complete, during each clock cycle the hardware will initiate a new instruction.

However, we must ensure that the instructions in different stages of the pipeline do not interfere with one another. This separation is done by introducing **pipeline registers** between successive stages of the pipeline, so that at the end of a clock cycle all the results from a given stage are stored into a register that is used as the input to the next stage on the next cycle.

1.4 Pipeline Hazards

A **hazard** (conflict) is a situation that prevents the next instruction from executing during its designated clock cycle. Hazards reduce the performance from the ideal speedup gained by pipelining. There are three classes of hazards:

1. **Structural hazards** - Attempt to use the same resource from different instructions simultaneously.
2. **Data hazards** - Attempt to use a result before it is ready.
3. **Control hazards** - Attempt to make a decision on the next instruction to execute before the condition is evaluated.

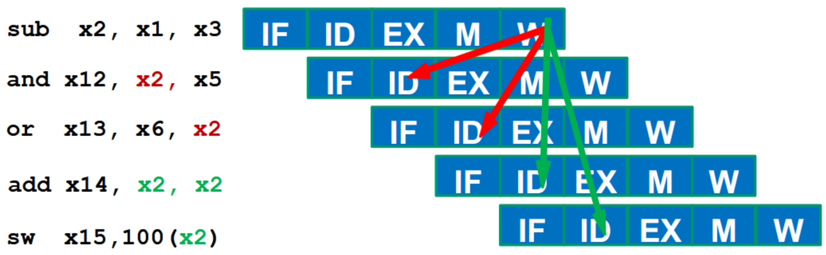
Hazards in the pipeline can make it necessary to *stall* the pipeline.

Note that there are *no* structural hazards in the RISC-V architecture, as the instruction memory is separated from the data memory, and the same register can be read and written in the same clock cycle by different instructions.

1.5 Data Hazards

Data hazards can arise when instructions that are dependent on each other are *too close* in the pipeline. Data hazards generate **Read After Write (RAW)** hazards in the pipeline. RAW hazards happen when instruction $n + 1$ tries to read a source operand before the previous instruction n has written its value in said register.

Example



We will assume that the RF (register file) reads from registers in the second half of the clock cycle and writes to registers in the first half of the clock cycle. In this way, it is possible to read and write the same register in the same clock cycle without any stall.

Now, to solve the data hazard problem, we can use different techniques:

- **Compilation techniques** (*static-time techniques*) - They must be implemented before running the program, at compile time.
  - Insertion of `nop` (no operation) instructions. A `nop` effectively makes the processor "wait" (by executing empty instructions) one clock cycle. Enough `nop` can be inserted until the hazard is no longer there.
  - Instruction scheduling to avoid that correlating instructions are too close. The compiler tries to insert independent instructions among correlated instructions, otherwise inserts `nop`.
- **Hardware techniques** (*runtime techniques*) - Can be implemented directly at runtime when an hazard is detected.
  - Insertion of *stalls* in the pipeline. Stalls effectively make the pipeline wait a few clock cycles until the current instruction finishes and the hazard is no longer there. They are analogous in performance to `nop` instructions, but they stop the instructions at any stage, rather than just making the processor wait before starting a new instruction.
  - Data forwarding or bypassing. It works by using temporary results stored in the pipeline registers instead of waiting for the write back of results in the RF. In other words, the result from the ALU can be fed back in the ALU at the next stage, rather than waiting for it to be stored and then read from a register.

1.6 Performance Evaluation Of Pipelines

To evaluate the performance we use the following metrics: **IC** (Instruction Count), **CPI** (Clocks Per Instructions), and **IPC** (Instructions Per Clock):

- $\text{CPI} = \dfrac{\text{IC} + \text{stalls} + 4}{IC}$
- $\text{IPC} = \dfrac{1}{\text{CPI}}$
- $\text{MIPS} = \dfrac{f_\text{clock}}{\text{CPI} \cdot 10^6}$, where
  - f_clock is the frequency of the clock
  - MIPS stands for Millions of Instructions per Second

Example

Let's evaluate the performance of a loop. Consider n iterations of a loop composed of m instructions per iteration, requiring k stalls per iteration:

- $\text{IC}_\text{per\_iter} = m$
- $\text{CPI}_\text{per\_iter} = \dfrac{\text{IC}_\text{per\_iter} + \text{stalls}_\text{per\_iter} + 4}{\text{IC}_\text{per\_iter}} = \dfrac{m+k+4}{m}$
- $\text{MIPS}_\text{per\_iter} = \dfrac{f_\text{clock}}{\text{CPI}_\text{per\_iter} \cdot 10^6}$

Let's now evaluate the *asymptotic* performance on the same loop:

- $\text{IC}_\text{AS} = m \cdot n$
- $\text{CPI}_\text{AS} = \lim_{n\to\infty} \dfrac{\text{IC}_\text{AS} + \text{stalls}_\text{AS} + 4}{\text{IC}_\text{AS}} = \lim_{n\to\infty} \dfrac{m \cdot n + k \cdot n + 4}{m \cdot n} = \dfrac{m+k}{m}$
- $\text{MIPS}_\text{AS} = \dfrac{f_\text{clock}}{\text{CPI}_\text{AS} \cdot 10^6}$

Overall, the *ideal* CPI on a pipelined processor would be 1, but stalls cause the pipeline performance to degrade from the ideal performance, so we have:

$$\text{CPI}_\text{avg} = \frac{\text{CPI}_\text{ideal} + \text{Pipe Stall Cycles per Instruction}}{1 + \text{Pipe Stall Cycles per Instruction}}$$

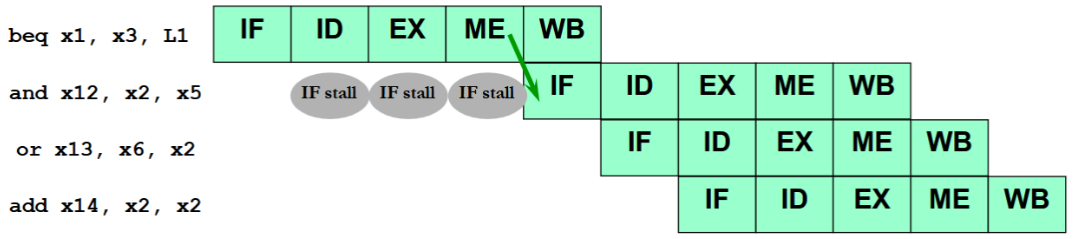
Pipeline stall cycles per instructions are due to structural hazards, data hazards, control hazards, and memory stalls.

2 Control hazards

Control hazards can cause a greater loss in performance than data hazards. When a branch is executed, it may or may not change the PC. If a branch changes the PC to its target address, it is a **taken branch**. If it falls through, it is **not taken**.

2.1 Conservative solution

To feed the pipeline, we need to fetch a new instruction at each clock cycle, but the branch decision is not yet ready. This problem to choose the correct instruction to be fetched after a branch is called **Control Hazard**. Control hazards arise from the pipelining of conditional branches and other `jump` instructions changing the PC. They reduce the performance from the ideal speedup gained by the pipelining because it is needed to stall the pipeline until branch resolution.



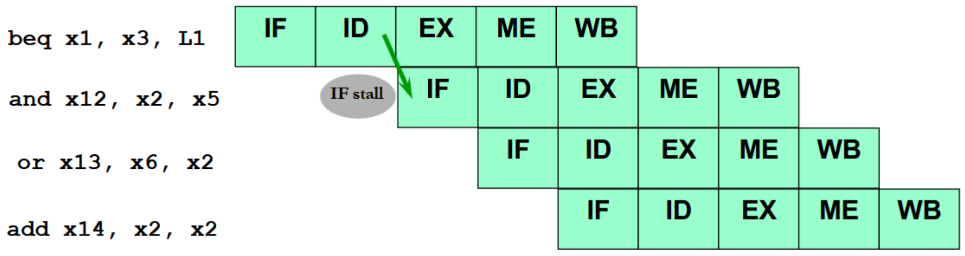
This way, each branch costs a penalty of 3 stalls to decide and fetch the correct instruction flow in the pipeline.

2.2 Early evaluation of branches in the ID stage

The idea is to improve the performance by

1. Comparing the registers to derive the branch outcome
2. Computing the branch target address
3. Updating the PC register

as soon as possible in the pipeline. The RISC-V pipeline anticipates the steps 1, 2, and 3 **during the ID stage**.



Her we need to insert only 1 stall before the target address is calculated.

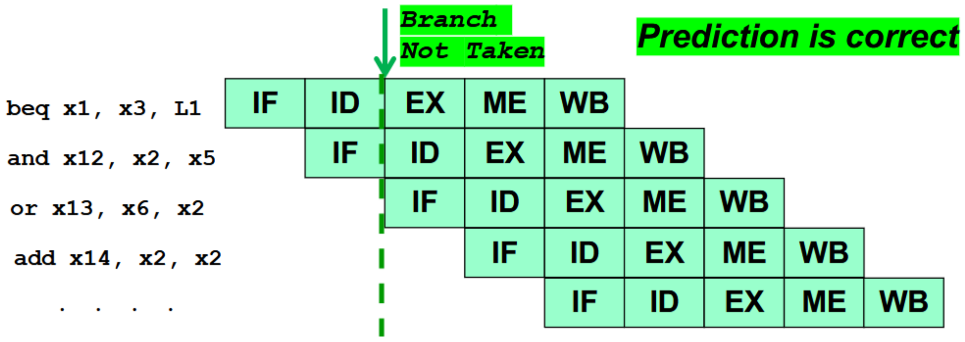
2.3 Static branch prediction techniques

In static branch prediction techniques, the prediction is fixed at **compile time** for each branch, during the entire execution of the program. There are different types of static branch prediction techniques.

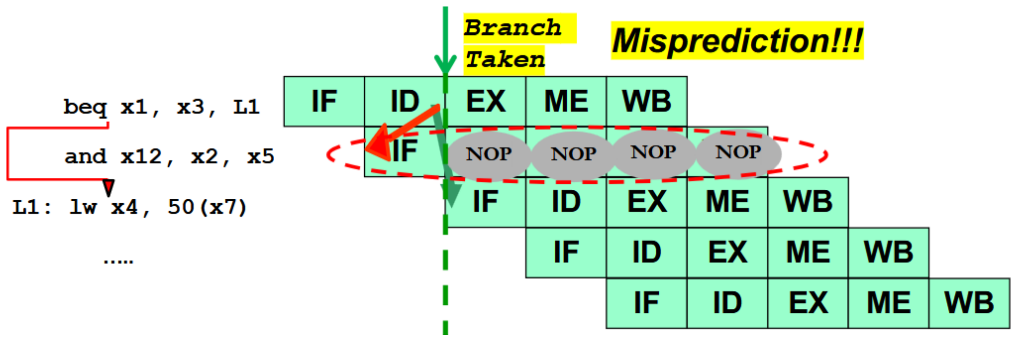
2.3.1 Branch Always Not Taken

This is the **easiest prediction**, always assume that the branch will be **not taken**. It is suitable for `if-then-else` statements, when the `then` clause is the most likely and the program will continue sequentially.

The next instruction in the pipeline is simply fetched as normal. If the branch outcome then is actually not taken, the pipeline will be fine.



What happens if the branch will be taken? In this case it will be necessary to **flush** the instruction after the branch (as it should have not been executed) and fetch the next instruction at the target address. If a branch is taken, it will cost 1 stall penalty.



This increases performance depending on how many *taken* branches there are.

2.3.2 Branch Always Taken

This is the dual case of the previous technique: we assume every branch as *taken*. It is suitable for backward branches such as `for` loops and `do-while` loops, that are most likely taken.

The problem here is that we need to know the *branch target address* to start fetching the instructions at the target. In the IF stage, we need to add a **branch target buffer** where to store the **predicted target address** based on the previous branch behavior.

This makes it similar to the predicted-not-taken technique: if the prediction is wrong, we get one cycle penalty, otherwise the pipeline continues as normal.

2.3.3 Backward Taken Forward Not Taken (BTFNT)

In this case the prediction is based on the **branch direction**: backward-going branches are predicted as *taken*, while forward-going branches are predicted as *not-taken*.

2.3.4 Profile-driven prediction

We **profile** the behavior of the application program by several early execution runs by using different datasets. The prediction is based on **profiling information** about the branch behavior collected during earlier runs.

The profile-driven prediction methos requires a **compiler hint bit** encoded by the compiler in the branch instruction format:

- Set to **1** if **taken** is the most probable branch outcome
- Set to **0** if **not taken** is the most probable branch outcome

2.3.5 Branch delay slot

Instead of stalling the pipeline for one cycle, the compiler tries to find a valid and useful instruction to be scheduled while the branch is being resolved (called the **branch delay slot**). There are 4 ways to schedule an instruction in the branch delay slot:

1. **From before** - The branch delay slot is scheduled with an *independent* instruction from before the branch. The instruction in the branch delay slot is *always executed*, regardless of the branch outcome. The execution will then continue, based on the branch outcome, in the right direction.
2. **From after** - dual to *from before.*
3. **From the taken path** - The branch delay slot is scheduled with one instruction from the target of the branch (*taken* path). This strategy is preferred when the branch is *taken with high probability*, such as `do-while` loop branches. If the branch is then *mispredicted* (not taken), the instruction in the delay slot must be flushed (unless it's ok to execute it anyway).
4. **From the not taken path** - The branch delay slot is scheduled with one instruction from the fall-through of the branch (*not taken* path). This strategy is preferred when the branch is *not taken with high probability*, such as `if-then-else` statements where the `else` path is less probable. If the branch is then *mispredicted* (taken), the instruction in the delay slot must be flushed (unless it's ok to execute it anyway).

2.4 Dynamic branch prediction techniques

In dynamic branch prediction techniques, the prediction for each branch can change at **runtime** during the program execution.

The idea is to use the past behavior (runtime behavior) to predict at runtime the future branch behavior. To do this, we use hardware to **dynamically** predict the outcome of a branch. This mean that the prediction can change at runtime if the branch changes its behavior during execution.

Dynamic branch prediction is based on *two interactive hardware blocks*:

1. **Branch Outcome Predictor** - To predict the direction of a branch (taken or not taken).

2. **Branch Target Buffer** - To predict the branch target address.

They are placed in the instruction fetch (IF) stage, to predict the next instruction to read in the instruction cache.

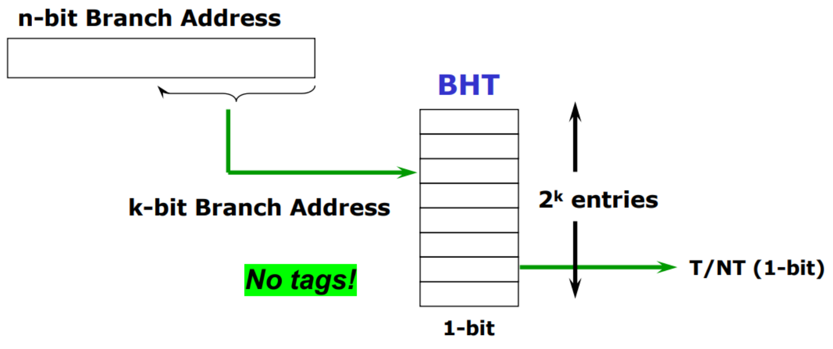
If the branch is predicted by the outcome predictor in the IF stage as *not taken*, then the PC is incremented as usual. If the prediction is correct, there will be no penalty, otherwise we need to flush the instruction fetched after the branch with a one-cycle penalty.

If the branch is predicted by the BOP in IF stage as *taken*, the target buffer gives the predicted target address. Similarly to the previous case, if the prediction is correct, there will be no penalty, otherwise we need to flush the instruction fetched after the branch with a one-cycle penalty.

2.4.1 Branch History Table (1-bit)

The *branch history table* (BHT) is a table containing 1 bit for each branch that says whether the branch was recently taken or not taken. The behavior is controlled by a **finite state machine** with only 2 states to remember the last direction taken by the branch.

The table is indexed by the lower portion k-bit of the address of the branch instruction, to keep the size of the table limited. For locality reasons, we would expect that the most significant bits of the branch address are **not** changed. The table has **no tag check** (every access is a hit). The prediction bit may have been put there by another branch with the same low order address bits, but this doesn't matter, as the **prediction is just a hint**.



A **misprediction** occurs either when:

- The prediction is incorrect for that branch;
- The same index has been referenced by two different branches, and the previous history refers to the other branch (this can occur because there is no tag check). To reduce this problem, it is enough to reduce the number of rows in the BHT (that is, to increase k), or to use a hashing function.

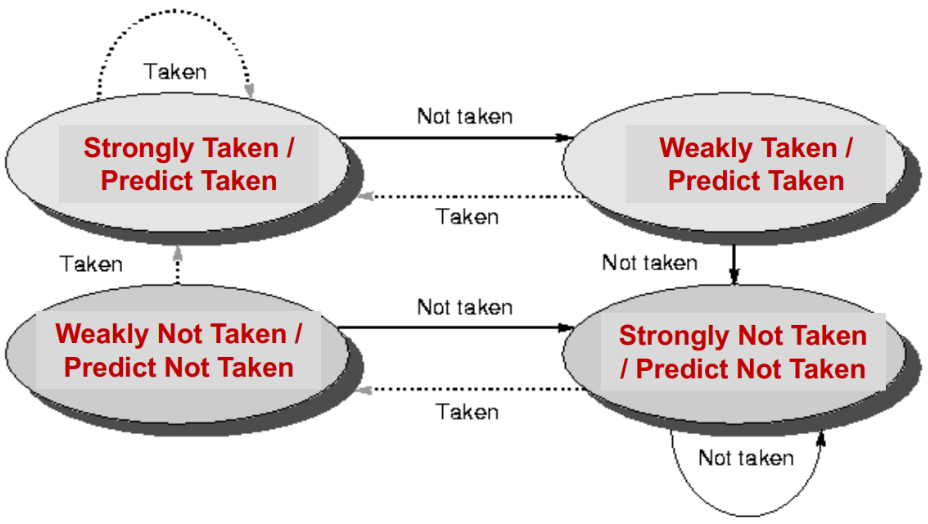
In a *loop*, for example, a branch is almost always T, and then NT once at the exit of the loop. Therefore, the 1-bit BHT causes 2 mispredictions:

1. At the **last iteration**, since the prediction bit is T, while we need to exit from the loop

2. When we re-enter the loop, at the **first iteration** we need the branch to stay in the loop, while the prediction bit was flipped to NT on previous execution of the last iteration of the loop

2.4.2 Branch History Table (2-bit)

We can use a 2-bit BHT to encode 4 states, in order to change the prediction only after 2 mispredictions. The finite state machine then becomes

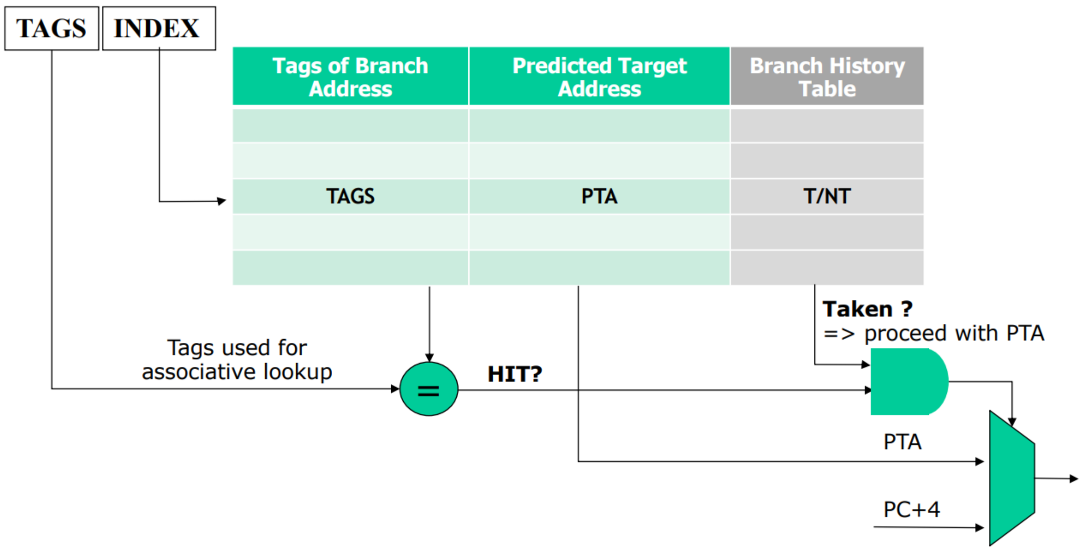

Taken
Strongly Taken / Predict Taken
Not taken
Weakly Taken / Predict Taken
Taken
Taken
Not taken
Weakly Not Taken / Predict Not Taken
Not taken
Strongly Not Taken / Predict Not Taken
Taken
Not taken

Coming back to the previous example, in the last loop iteration, we mispredict the branch but we do not need to change the prediction. When re-entering the loop, the branch is correctly predicted as taken.

2.4.3 Branch target buffer

The *branch target buffer* is a cache storing the **predicted target address** for the taken-branch instructions. This predicted target address is expressed as PC-relative.

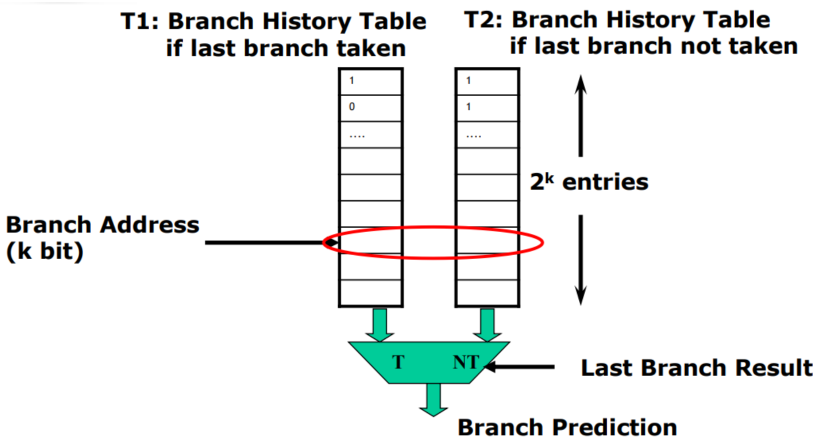
The branch target buffer is designed as a direct-mapped cache placed in the IF stage by using the address of the fetched branch instruction to index the cache. Then, **tags** are used for the associative lookup. The branch target buffer is used in combination with the branch history table in the IF stage.



2.4.4 Correlating branch predictors

The 2-bit BHT uses only the recent behavior of a single branch to predict the future behavior of that branch. We can consider use the following idea: the behavior of recent branches are **correlated**, meaning that the recent behavior of **other branches** (rather than just the current branch) can also influence the prediction of the current branch.

This type of branch predictors that use the behavior of other branches to make a prediction are called **correlating predictors**.


T1: Branch History Table if last branch taken
T2: Branch History Table if last branch not taken
1
0
....
1
1
....
2k entries
Branch Address (k bit)
T
NT
Last Branch Result
Branch Prediction

We record if the most recently executed branches have been *taken* or *not taken*. The branch is predicted based on the previous executed branch by selecting the appropriate 1-bit BHT:

- One prediction is used if the last branch executed was *taken*;
- Another prediction is used if the last branch executed was *not taken*.

Normally, the last branch executed is *not* the same instruction as the branch being predicted (although this can occur in simple loops with no other branches in the loop).

In general, an (m, n) correlating predictor records the last m branches to choose from 2^n BHTs, each of which is an n-bit predictor.

The branch prediction buffer can be indexed by using a concatenation of low-order bits from the branch address with m-bit global history (i.e., global history of the most recent m branches). For example, a 2-bit BHT predictor with no global history is simply a (0, 2) predictor.

2.4.5 Two-level adaptive branch predictors

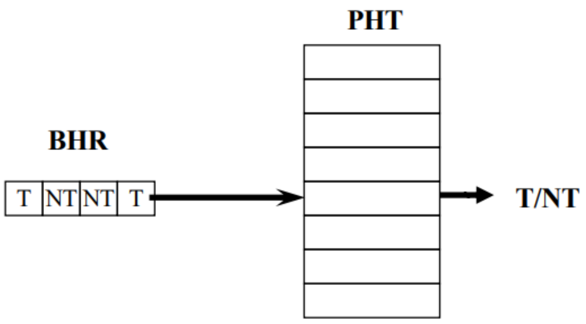
The first level history is recorded in one or more k-bit shift register, called **Branch History Register (BHR)**, which records the outcomes of the k most recent branches (used as **global** history).

The second level history is recorded in one or more tables, called **Pattern History Table (PHT)** of 2-bit saturating counters (used as a **local** history).

The BHR is used to index the PHT to select which 2-bit counter to use. Once the 2-bit counter is selected, the prediction is made using the same methos as in the 2-bit counter scheme.

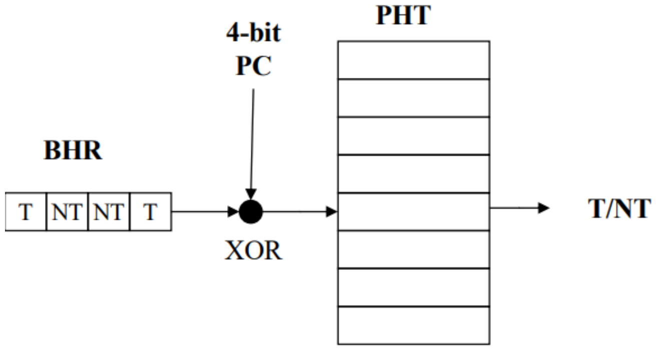
2.4.6 Global adaptive predictor

The Global Adaptive (GA) predictor uses the correlation between the current branch and the other branches in the global history to make the prediction: a PHT (local history) indexed by the content of BHT (global history).



2.4.7 GShare predictor

It is a variation of the GA predictor where we want to correlate the BHR recording the outcomes of the most recent branches (global history) with the low-order bits of the branch address. **GShare**: we make the XOR of 4-bit BHR (global history), with the low-order 4-bit of PC (branch address) to index the PHT(local history)



2.5 Performance of branch prediction techniques

The *performance* of a branch prediction technique depends on:

- **Accuracy** - Measured in terms of percentage of incorrect predictions given by the predictor.
- **Cost** - Measured in terms of time lost to execute useless instructions (misprediction penalty) given by the processor architecture. The cost increases for deeply pipelined processors.
- **Branch frequency** - Given by the application. the importance of accurate branch prediction is higher in programs with higher number of branch instructions.

3 Instruction Level Parallelism

Pipelining overlaps the execution of instructions, exploiting the **instruction level parallelism**. The goal is to maximize the throughput, instructions per clock (IPC), and minimize the clocks per instructions (CPI). Pipelining improves instructions throughput, but not the latency of the single instruction.

Determining **dependencies** among instructions is critical to define the amount of parallelism existing in a program. If two instructions are *dependent* on each other, they cannot be executed in parallel, they must be executed in order, or only partially overlapped. There are three different types of dependencies in a code:

1. **True data dependencies** - Instruction j is dependent on some data produced by a previous instruction i;
2. **Name Dependencies** - Two instructions use the same register or memory location;
3. **Control dependencies** - Seen in Section 2. They impose the order of instructions.

3.1 Name dependencies

A **name dependency** occurs when two instructions use the same register or memory location, but there is no flow of data between the instructions associated with that name.

Name dependencies are *not* true data dependencies, since there is no value (no data flow) being transmitted between the two instructions, here it's just a **register reuse**.

Consider instruction I_i that precedes instruction I_j in the program order. There are two types of name dependencies:

- **Anti-dependencies** - When I_j writes in a register or memory location that instruction I_i reads, it can generate a **Write After Read (WAR) hazard**. Original instructions ordering must be preserved to ensure that I_i reads the previous value.

$$\begin{aligned} I_i :&\quad r_3 \leftarrow r_1 \text{ operation } r_2 \\ I_j :&\quad r_1 \leftarrow r_4 \text{ operation } r_5 \end{aligned}$$

- **Output dependencies** - When I_i and I_j write in the same register or memory location, it can generate a **Write After Write (WAW) hazard**. Original instructions ordering must be preserved to ensure that the value finally written corresponds to I_j.

$$\begin{aligned} I_i: &\quad {\color{red}r_3} \leftarrow r_1 \text{ operation } r_2 \\ I_j: &\quad {\color{red}r_3} \leftarrow r_6 \text{ operation } r_7 \end{aligned}$$

.

We can solve name dependencies with **register renaming**: if the register used can be changed, then the instructions do not conflict anymore. This can be easily done if there are enough registers available in the ISA (instruction set architecture). It can be either done statically by the compiler, or dynamically by the hardware.

On the other side, dependencies through memory locations are more difficult to detect (*memory disambiguationi* problem), since two addresses may refer to the same location but can look different.

A dependency of this kind can potentially generate an hazard, but the number of stalls to eliminate the hazard are part of the pipeline architecture (dependencies are a property of the *program*, while hazards are a property of the *architecture*).

In order for a program to be correct, it is necessary to respect these two critical **properties** (normally preserved by maintaining both data and control dependencies during scheduling):

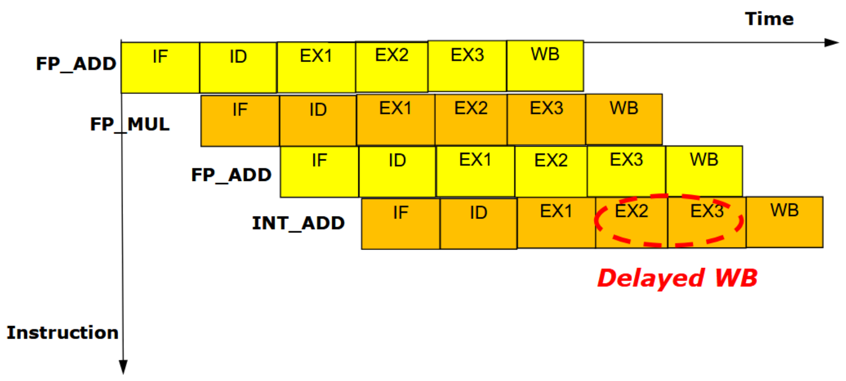
1. **Data flow**: the actual flow of data values among instructions that produces the correct results and consumes them;
2. **Exception behavior**: preserving this property means that any changes in the ordering of instruction execution must not change how exception are raised in the program.

3.2 Multi-cycle pipelining

We will uses these basic assumptions:

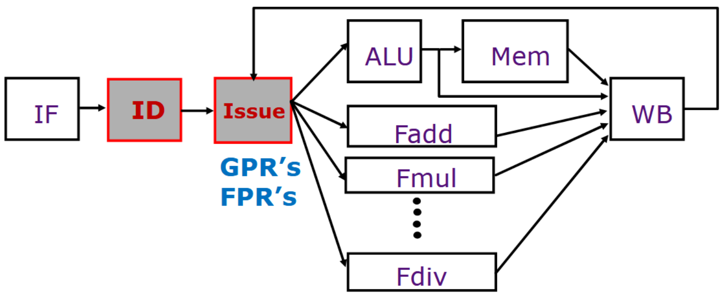
- We consider **single-issue** processors (one instruction issued per clock);
- Instructions are **issued in-order**;
- The **execution stage** might require multiple cycles latency, depending on the operation type;
- **Memory stages** might require multiple cycles access time due to instruction and data cache misses.

3.2.1 Multi-cycle in-order pipeline



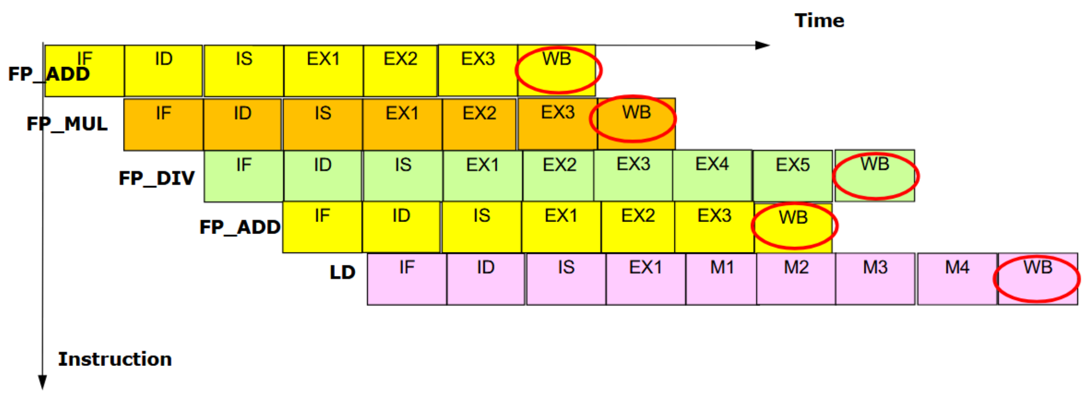
We se that, with this technique, the instructions are issued in-order, and committed in-order. This avoids the generation of WAR and WAW hazards, and preserves the precise exception model.

3.3 Multi-cycle out-of-order pipeline



We can see the following characteristics:

- The ID stage is split in 2 stages: *instruction decode* (ID) and *Issue* (read registers);
- There are multiple functional units with variable latency;
- There exist multi-cycle floating-point instructions with long latency;
- The memory systems have variable access time, multi-cycle memory accesses due to data cache misses (unpredictable statically);
- No more commit point: out-of-order commit (instructions are still issued in-order). There is the need to check for WAR and WAW hazards and imprecise exceptions.

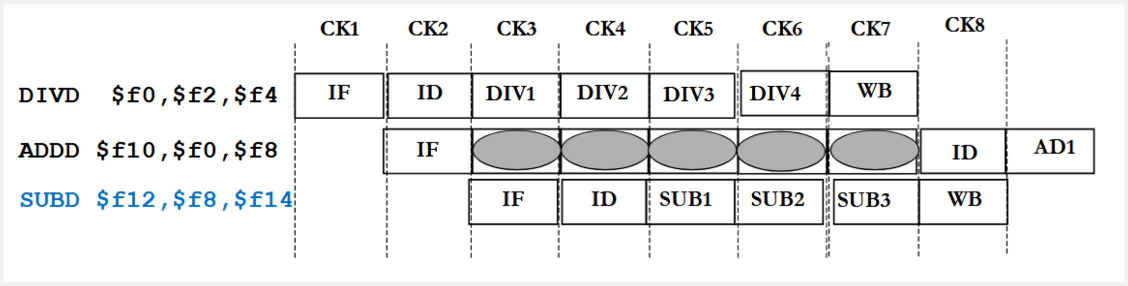


3.4 Dynamic scheduling

Hazards due to true data dependencies that cannot be solved by forwarding cause **stalls** of the pipeline. This means that no new instructions are fetched nor issued, even if they are not data dependent.

The **solution** is to allow data independent instructions behind a stall to proceed. The hardware manages dynamically the instruction execution to reduce stalls: an instruction execution begins as soon as their operands are unavailable. This generates out-of-order execution and completion.

Example



Here ADDD *stalls* for RAW hazard on $f0 (waiting many clock cycles for DIVD commit), SUBD would stall even if not data dependent on any previous instruction.

Instead, we to enable SUBD to proceed. This generates *out-of-order execution*.

3.5 Imprecise exceptions

An exception is **imprecise** if the processor state when an exception is raised does not look exactly as if the instructions were executed in-order.

Imprecise exceptions can occur *out-of-order* because:

- The pipeline may have *already* completed instructions that are *later* in the program order than the instruction causing the exception.
- The pipeline may have *not yet* completed some instructions that are *earlier* in the program order than the instruction causing the exception.

Imprecise exception make it difficult to restart execution after handling.

3.6 Multiple-issue processors

A scalar pipeline (the one seen up until now) is limited to a $CPI_{\text{ideal}} = 1$. That is, it can never fetch and execute more than *one instruction per clock* (single-issue). It can be even worse due to *stalls* added to solve hazards.

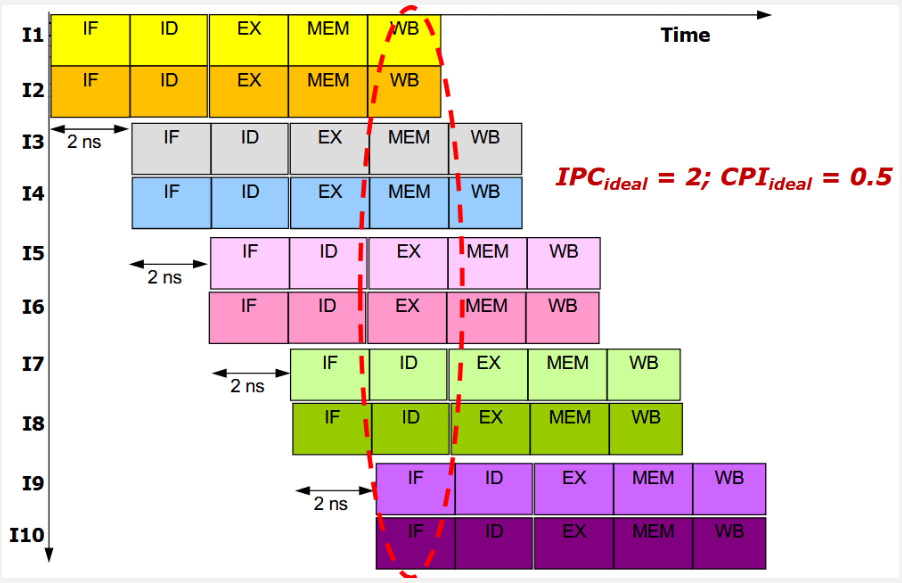
To reach higher performance, we need **multiple-issue**. It means fetching and executing *more than one* instruction per clock.

Instruction dependencies must be detected and solved: instructions must be *re-ordered* (**scheduling**) to achieve the highest instruction level parallelism given the available resources:

$$IPC_{\text{ideal}} > 1 \implies CPI_{\text{ideal}} < 1$$

Example: dual-issue pipeline

With a 2-issue 5-stage pipeline there are 2×5 different instructions overlapped:



3.6.1 Dynamic scheduling

Dynamic scheduling can be applied to both *single-issue* scalar processor and *multi-issue* processors (*superscalars*). However, multiple-issue pipelines have disadvantages:

- Very complex logic and cost to check and manage dependencies at runtime (i.e., to decide with instructions can be issued at every clock cycle).
- Cycle time limited by scheduling logic (dispatcher and associated dependency-checking logic).
- It does not scale well. It is almost impractical to make the issue-width greater than 4.

3.6.2 VLIW (Very Long Instruction Word) and static scheduling

VLIW processors issue a fixed number of instructions formatted either as one large instruction, or as a fixed instruction packet with parallelism among instructions explicitly indicated by the instruction.

VLIW relies on the compiler to schedule the code for the processor. This makes it similar to static scheduling, although static scheduling issues a varying (rather than a fixed) number of instructions per clock.

The advantages of VLIW increases as the maximum issue rate grows. Static scheduling, on the other hand, is typically used for narrow issue width, as their advantages diminish as the issue width grows.

These are the main disadvantages of static scheduling:

- **Unpredictable branch behavior**: the code parallelization is limited to basic blocks;
- **Unpredictable cache behavior**: variable memory latency for hits/misses;
- **Complexity of compiler technology**: the compiler needs to find a lot of parallelism in order to keep the multiple functional units of the processors busy;
- **Code size explosion** duo to insertion of NOPs;
- **Low code portability and binary code incompatibility**: consequence of the larger exposure of the microarchitecture (implementation details) at the compiler in the generated code;
- **Low performance portability**.

3.7 ILP Limitations

The **issue width** is the number of instructions that can be issued in a single cycle by a *multiple-issue* processor.

When superscalars were invented, 2- and rapidly 4-issue width processors were created. However, due to the intrinsic level of parallelism of a program, it is hard to decide which 8, or 16, instructions can execute every cycle.

The solution is to *introduce more levels of parallelism.*

4 Scoreboard dynamic scheduling algorithm

In the case of the simple scalar pipeline, hazards due to true data dependences that cannot be solved by forwarding cause the stall of the pipeline. No new instructions can be fetched nor issued even if there are not data dependencies in the new instructions.

The solution is to allow data independent instructions behind a stall to proceed. The hardware rearranges dynamically the instruction execution to reduce stalls. This causes out-of-order execution and out-of-order commit.

The **scoreboard** is a dynamic scheduling algorithm (implemented at the hardware level). We will make use of the following basic assumptions:

- We consider a single-issue processor;
- In-order issue;
- Instruction execution begins as soon as operands are ready, whenever not dependent on previous instructions (no RAW hazards);
- There are multiple pipelined Functional Units with variable latencies;
- Execution stage might require multiple cycles, depending on the operation type and latency;
- Memory stage might require multiple cycles access time due to data cache misses.

This means that there is out-of-order execution and out-of-order commit, which introduces the possibility of WAR and WAW hazards.

4.1 Scoreboard basics

Scoreboard allows data independent instructions behind a stall to proceed, not waiting for previous instructions.

In the time between when an instruction begins execution and when in completes execution, the instruction is in execution. The scoreboard pipeline allows multiple instructions in execution at the same time, meaning that it requires multiple pipelined functional units.

4.2 Scoreboard pipeline stages

Scoreboard divides the ID stage in two stages:

1. **Issue** - Decode instructions and check for structural hazards;
2. **Read Operands** - Wait until not dependent on previous instructions and no data hazards, then read operands.

Scoreboard allows instructions to execute whenever both stages 1 and 2 hold, not waiting for prior instructions to complete.

Scoreboard keeps track of dependencies and the state of parallel ongoing operations. Instructions pass through the issue stage *in-order*, but they can be stalled or bypass each other in the read operand stage (out-of-order read operands). Then instructions enter *execution out-of-order* and have different latencies, which implies *out-of-order commit.*

There are multiple instructions in the execution phase, meaning that there are multiple pipelined execution units. Also, there is no register renaming, as this would be a compile time technique.

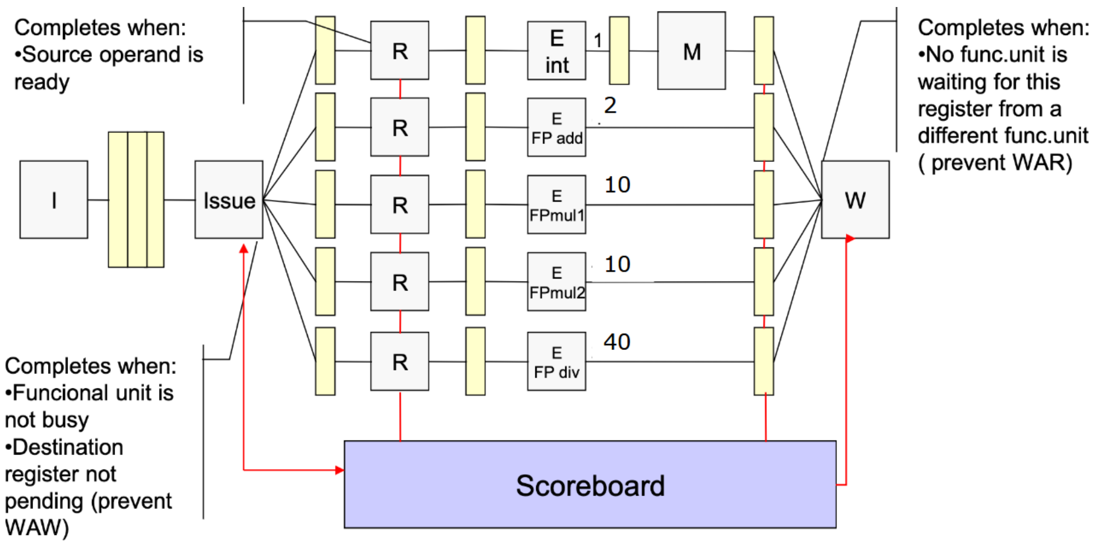
The fact that there is out-of-order commit means that *WAR and WAW hazards can occur.* To *solve WAR* hazards, we read registers only during the Read Operands stage, and we stall Write Back until the previous registers have been read. To *solve WAW* hazards, we detect the WAW hazard and stall the issuing of new instructions until previous instruction causing WAW completes.

4.3 Scoreboard scheme

Any hazard detection and resolution is **centralized** in the scoreboard. Every instruction goes through the scoreboard, where a record of data dependencies is constructed. The scoreboard then determines when the instruction can read its operand and begin execution (check for RAW).

If the scoreboard decides that the instruction cannot execute immediately, it monitors every change and decides *when* the instruction can execute. The scoreboard also controls *when* the instruction can write its result into the destination register (check for WAR and WAW).

The idea of the scoreboard is to keep track of the status of instructions, functional units, and registers.


Completes when:
•Source operand is ready
Completes when:
•No func.unit is waiting for this register from a different func.unit (prevent WAR)
Completes when:
•Funcional unit is not busy
•Destination register not pending (prevent WAW)
I
Issue
R
E int
E FP add
E FPmul1
E FPmul2
E FP div
1
2
10
10
40
M
W
Scoreboard

The scoreboard control has **4 stages**:

1. **Issue** - Decode instruction and check for structural hazards and WAW hazards. The instructions are issued in program order (for hazard checking).
    - If a functional unit (FU) for the instruction is available (*no structural hazard*) and no other active instruction has the same destination register (*no WAW hazard*), then the scoreboard issues the instruction to the FU and updates its data structure.
    - If either a *structural hazard* or a *WAW hazard* exists, then the instruction issue stalls, and no further instruction will issue until these hazards are solved.
2. **Read Operands** - Wait until there are no RAW hazards, then read operands. Check for structural hazards in reading ports of RF.
   A source operand is available if:
    - no earlier issued active instruction will write it; or
    - a functional unit is writing its value in a register.

   When the source operands are available, the scoreboard tells the FU to proceed to read the operands from the RF and begin execution.

RAW hazards are solved dynamically in this step: there is *out-of-order* reading of operands, and instructions are sent into execution *out-of-order* (no data forwarding).

3. **Execution** - The FU begins execution upon receiving operands. When the result is ready, it notifies the scoreboard that the execution has been completed.
   FUs are characterized by *variable latencies* to complete execution. The load/store latency depends on data cache HIT/MISS times. Therefore, there is *out-of-order execution*.

4. **Write result** - Check for WAR hazards on destination, check for structural hazards in writing RF, and finish execution. Once the scoreboard is aware that the FU has completed execution, the scoreboard *checks for WAR hazards*:

   - If none, it writes the results.
   - If there is a WAR, then the scoreboard stalls the completing instruction.

5 Tomasulo dynamic scheduling algorithm

The Tomasulo algorithm is another dynamic scheduling algorithm. It enables instructions execution behind a stall to proceed. It works by introducing the **implicit register renaming** to avoid WAR and WAW hazards.

WAR hazards are avoided by **executing an instruction only when its operands are available**, which is exactly what the simpler scoreboard approach provides. WAR and WAW hazards, which arise from name dependences, are eliminated by register renaming. **Register renaming** eliminates these hazards by textbfrenaming all destination registers, including those with 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.

Example

Consider the following code:

```
DIV.D F0, F2, F4
ADD.D F6, F0, F8     # RAW F0
S.D F6, 0(R1)        # RAW F6
MUL.D F6, F10, F8    # WAW F6 with ADD.D, WAR F6 with S.D
```

Static register renaming introduces a register `S` to avoid WAR and WAW hazards (statically by the compiler):

```
DIV.D F0, F2, F4
ADD.D S, F0, F8     # RAW F0
S.D S, 0(R1)        # RAW S
MUL.D F6, F10, F8
```

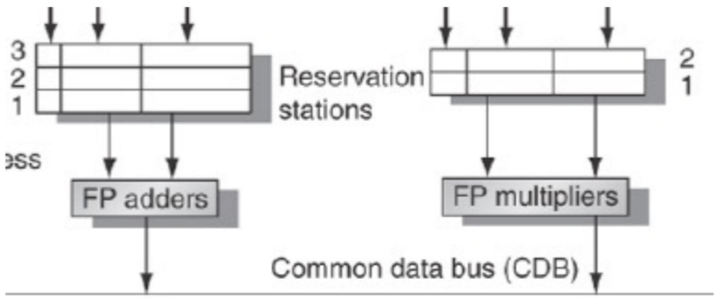
Implicit register renaming uses the *reservation station* `RS1` to avoid WAR and WAW hazards (dynamically by the hardware):

```
DIV.D F0, F2, F4
```

```
    ADD.D RS1, F0, F8      # RAW F0
    S.D RS1, 0(R1)         # RAW RS1
    MUL.D F6, F10, F8
```

5.1 Tomasulo basic concepts

Tomasulo introduces some FU buffers called **Reservation Stations** (RSs) in front of the FUs to keep pending operands. The control logic and RSs are **distributed** with the function units (instead of being centralized like the scoreboard).



Registers in instructions are replaced by their *values* or *pointers to reservation stations* to enable *implicit register renaming*. This avoids WAR and WAW hazards by renaming results by using RS numbers instead of RF numbers. Since there are more reservation stations than registers, Tomasulo can perform optimizations that the compiler can't.

The basic idea is that the **results are passed to the FUs from Reservation Stations** (not through registers), via a **common data bus that broadcasts the results to all FUs and to store buffers** (like a sort of forwarding). The *store buffers* are treated as a sort of RSs as well.

5.2 Reservation station components

Each reservation station has the following fields:

- **Tag** - Identifies the reservation station.
- **Busy** - Indicates that this reservation station is occupied.
- **OP** - The type of operation to perform on the source operands.
- **Vj, Vk** - The values of the source operands.
- **Qj, Qk** - Pointers to to the reservation station that will produce **Vj**, **Vk** (a zero value means that the source operand is already available in **Vj** or **Vk**).

Note that either V-field or Q-field is valid for each operand.

5.3 Register file and Store Buffers

Each entry in the RF and in the store buffers have a **value (Vi)** and a **pointer (Qi)** field. The value (Vi) field holds the register/buffer content. The pointer (Qi) field corresponds to the number of the reservation stations producing the result to be stored in this register.

If the pointer is zero means that the value is available in the register/buffer content (no active instruction is computing the result)

5.4 Load/store buffers

Load/Store buffers have a **busy** and **address** field.

The address field holds info for memory address calculation for load/stores. Initially it contains the instruction offset (immediate field). After address calculation, it stores the effective address.

Store instructions in the **store buffers** wait for the value given by the RF or the FUs to be sent to the memory unit.

5.5 Stages of the Tomasulo algorithm

The Tomasulo algorithm comprises the following stages:

1. **Issue** (in-order):
    - Get an instruction **I** from the head of instruction queue (maintained in FIFO order to ensure **in-order issue**).
    - **Check if there is an empty reservation station**, otherwise the instruction **stalls**.
    - If the operands are not ready in the RF, keep track of the FU that will produce them (**Q pointers**). This step **renames registers**, eliminating WAR and WAW hazards.
        - **WAR resolution**: if **I** writes **Rx**, read by an instruction **K** already issued, **K** knows already the value of **Rx** read in the reservation station buffer, or knows what instruction (previously issued) will write it. So the RF can be linked to **I**.
        - **WAW resolution**: since we use in-order issue, the RF can be linked to **I**.
2. **Start Execution** (out-of-order):
    - Execution begins when both operands are ready, which resolves RAW hazards. Execution also requires the FU to be available, checking for structural hazards in the FU. By delaying the execution until operands are available, RAW hazards are avoided at this stage.
    - If operands are not ready, the instruction monitors the Common Data Bus for results.
        - Note that several instructions could become ready in the same clock cycle for the same FU. We need to check if the execution unit is available. This is a critical choice for loads/stores to be kept in program order.
        - Note also that usually RAW hazards are shorter because operands are given directly by the reservation station without waiting for the RF write back.

- **Load and stores** have a *two-step* execution process:
   (a) **First step**: compute the effective address when base register is available, and place it in the load/store buffer.
   (b) **Second step**: loads in the Load Buffer as soon as the memory unit is available, and stores in the Store Buffer wait for the value to be stored before being sent to the memory unit.
- To **preserve the exception behavior**, no instruction can initiate execution until all branches preceding it in the program order have completed. This restriction guarantees that an instruction that generates an exception really would have been executed. If branch prediction is used, the CPU must confirm prediction correctness before beginning the execution of next instructions

3. **Write result** (out-of-order):
   - When it is available, write the result on *Common Data Bus* and from there into *Register File* and into *all reservation stations (including store buffers)* waiting for this result.
   - *Stores* also write data to the memory unit during this stage (when the memory address and result data are available).
   - Then, *mark the reservation station as available.*

Reservation stations offer a sort of *data forwarding.*

5.6 ReOrder Buffer

5.6.1 Hardware-based speculation

We introduce the concept of hardware-based (HW) speculation, which extends the ideas of dynamic scheduling beyond branches.

HW-based speculation combines **3 key concepts**:

1. **Dynamic branch prediction**;
2. **Speculation** to enable the execution of instructions before the branches are solved by undoing the effects of mispredictions;
3. **Dynamic scheduling** beyond branches.

To suppor HW-based speculation, we need to enable the execution of instructions *before* the control dependences are solved. In case of **misprediction**, we need to undo the effects of an incorrectly speculated sequence of instructions. Therefore, we need to separate the process of execution completion from the commit of the instruction result in the RF or in memory.

The solution is to introduce an HW buffer, called **ReOrder Buffer**, to hold the result of an instruction that has finished execution, but not yet committed in RF/memory.

5.6.2 ReOrder Buffer (ROB)

ReOrder Buffer has been introduced to support **out-of-order execution** but **in-order commit**.

A buffer to hold the results of instructions that have finished execution *but not yet committed*. And a buffer to pass results among instructions that have started *speculatively* after a branch.

The ROB is also used to pass results among dependent instructions to start execution as soon as possible. *The renaming function of Reservation Stations is replaced by ROB entries.*

Uses **ReOrder Buffer numbers** instead of reservation station numbers as pointers to pass data between dependent instructions. Reservation Stations now are used only to buffer instructions and operands to FUs (to reduce structural hazards).

ROB supplies operands to other RSs between execution complete and commit (as a sort of forwarding). Once the instruction commits, the result will be written from the ROB to the register file. A mispredicted branch flushes the ROB entries marked as speculative.

The instructions are written in ROB in strict program order. When an instruction is issued, an entry is allocated to the ROB **in-order**. Each entry must **keep the status of an instruction**: issued, execution started, execution completed, ready to commit. Each entry includes a **speculative status flag**, indicating whether the instruction is being speculatively executed or not.

An instruction is **ready to commit (retire)** if and only if:

- Its execution has already completed, AND
- It is not marked as speculative, AND
- All previous instructions have already retired.

The ROB is a **circular buffer** with two pointers:

1. **Head pointer** indicating the instruction that will **commit** (leaving ROB) *first.*
2. **Tail pointer** indicating the next free entry.

5.6.3 ROB content

Each entry in ROB contains the following fields:

- **Busy field** - Indicates whether ROB entry is busy or not.
- **Instruction type field** - Indicates whether instruction is a branch (no destination result), a store (memory address destination), or a Load/ALU (register destination).
- **Destination field** - Supplies register number (for loads and ALU instructions) or memory address (for stores) where results should be written.
- **Value field** - Used to hold the value of the result until instruction commits.
- **Ready field** - Indicates that instruction has completed execution, value is ready.
- **Speculative field** - Indicates whether instruction is executed speculatively or not.

5.7 Speculative Tomasulo architecture with reorder buffer

Pointers are directed towards the ROB entries instead of reservation stations. **Implicit register renaming** by using the ROB entries to eliminate WAR and WAW hazards and enable dynamic loop unrolling.

A register or memory location is updated only when the instruction reaches the *head of ROB*. By using ROB to hold the result, it is easy to undo speculated instructions on mispredicted branches, and manage exceptions precisely.

There are 4 phases of the Speculative Tomasulo Algorithm:

1. **Issue** - Get instruction from the instructions queue.
   - If there are one RS entry free and one ROB entry free, the instruction is issued:
     - If its operands are available in RF or ROB, they are sent to RS
     - Number of ROB entry allocated for first results is also sent to RSs (it will be used to tag the result when it will be placed on the CDB)
   - If the RS is full or/and ROB is full, the instruction stalls
2. **Execution started**
   - When both operands are ready in the RS (**RAW solved**), the instruction starts to execute on operands (**EX**)
   - If one or more operands are not yet ready, check for RAW to be solved by monitoring the CDB for the result
   - For a **store**: only the base register needs to be available (at this point, only the effective address is computed)
3. **Execution completed and write result in ROB**
   - Write result on Common Data Bus to all awaiting RS and ROB value field
   - Mark RS as available
   - For a *store*: the value to be stored is written in the ROB value field, otherwise monitor CDB until value is broadcast.
4. **Commit** - Update RF or memory with ROB result.
   - When instruction is at the head of ROB and the result is ready, update RF with result (or store to memory), and remove instruction from ROB entry.
   - Mispredicted branch flushes ROB entries (sometimes called "graduation").

Note that there are **3 different possible commit sequences**:

- **Normal commit**: instruction reaches the head of the ROB, result is present in the buffer. Result is stored in the Register File, instruction is removed from ROB.
- **Store commit**: as above, but the result is stored in memory, rather than in the RF.

- **Instruction is a mispredicted branch**: speculation was wrong, so ROB is flushed ("graduation"), and execution restarts at the correct successor of the branch. If the branch was instead correctly predicted, the branch is completed.

5.7.1 ReOrder Buffer: summary

- Only retiring instructions can *complete*, i.e., update architectural registers and memory;
- ROB can support both speculative execution and precise exception handling;
- Speculative execution: each ROB entry includes a *speculative status flag*, indicating whether the instruction has been executed speculatively.
- Finished instructions cannot retire as long as they are in speculative status;
- *Interrupt handling*: exceptions generated in connection with instruction execution are made *precise* by accepting exception request only when instruction becomes "ready to retire" (exceptions are processed *in order*).

6 Very Long Instruction Word (VLIW) architectures

VLIW architectures are a type of static multiple-issue processor designed to exploit Instruction Level Parallelism (ILP). Instead of hardware dynamically checking dependencies and scheduling instructions, VLIW processors rely on the compiler to statically identify and package parallel operations into a single, very long instruction word, also known as a bundle.

The **single-issue packet (bundle)** represents a wide instruction with multiple independent operations per instruction, named **Very Long Instruction Word**. The compiler identifies *statically* the multiple independent operations to be executed *in parallel* by the multiple functional units.

The compiler solves *statically* the *structural hazards* for the use of HW resources and the *data hazards*, otherwise the compiler inserts NOPs.

The long instruction (bundle) has a fixed set of operations (*slots*). For example, a 4-issue VLIW has a bundle that contains up to 4 operations (corresponding to 4 slots).

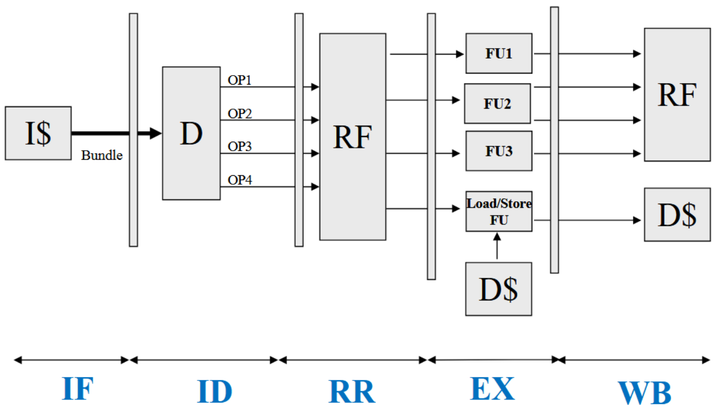
6.1 VLIW Processors

There is a **single PC** to fetch a long instruction (bundle). Only one branch for each bundle to modify the control flow.

There is a **shared multi-ported register file**: if the bundle has 4 slots, we need 2×4 read ports and 4 write ports to read 8 source registers per cycle, and to write 4 destination registers per cycle.

To keep the FUs busy, there must be enough parallelism in the source code to fill in the available 4 operation slots. Otherwise, **NOPs are inserted**.

If each slot is assigned to a Functional Unit, the decode unit is a simple decoder and each operation is passed to the corresponding FU to be executed.



If there are more parallel FUs than the number of issues (slots), the architecture must have a **dispatch network** to redirect each operation and the related source operands to the target FU.

6.1.1 Data dependencies

True, anti, and output data dependencies are solved **statically** by the compiler by considering the FU latency. **To solve RAW hazards in VLIW processors**:

- The compiler during the scheduling phase **statically** reorders instructions (not involved in the dependencies).
- Otherwise, the compiler introduces some **NOPs**.

Operation latencies and **data dependencies** must be exposed by the compiler. Otherwise, correct instruction execution is compromised. This is true even in case of a pipelined multiplier.

WAR and *WAW* hazards are statically solved by the compiler by correctly selecting temporal slots for the operations, or by using Register Renaming. *Structural hazards* are also solved by the compiler.

The compiler can also provide useful *hints* on how to statically predict branches. However, a **mispredicted branch** must be solved dynamically by the hardware **flushing** the execution of the speculative instructions in the pipeline.

To keep *in-order execution*, the Write Back phase of the parallel operations in a bundle must occur at the same clock cycle, to avoid structural hazards accessing the RF and WAR/WAW hazards.

Operations in a bundle are constrained to the **latency** of the longer latency operation in the bundle. Otherwise, we get out-of-order execution and we need to check the RF write accesses and WAR/WAW hazards.

6.1.2 Register pressure

Register pressure refers to the fact that the multicycle latency of the operations, together with the multiple issue of instructions, generates an increment of the number of registers occupied over time.

The **register renaming** used to solve WAR and WAW hazards by the compiler also increases the register pressure.

6.1.3 Dynamic events

The compiler at static time does not know the behavior of some **dynamic events** such as:

- **Data Cache Misses**: stalls are introduced at runtime (latency of a data cache miss is known at compile time).
- **Branch Mispredictions**: need of flushing at runtime the execution of speculative instructions in the pipeline.

6.1.4 Statically scheduled processors

Compilers can use sophisticated algorithms for code scheduling to exploit ILP. It detects whether instructions can be run in parallel based on hardware resource availability and data dependencies.

Consider a **basic block**, a straight-line sequence of code with only one entry point and one exit point. This means that if the first instruction in a basic block is executed, all subsequent instructions in that block are guaranteed to be executed exactly once and in order (without any branches). The *problem* is that the amount of parallelism within a basic block is small.

Data dependencies can further limit the amount of ILP we can exploit within a **basic block** to much less than the average block size (true data dependencies force sequential execution of instructions).

to obtain substantial performance enhancements, we must *exploit ILP across multiple basic blocks* (i.e., across branches).

6.2 Main advantages of VLIW

Good performance through extensive **compiler optimizations** to schedule the code statically (exploiting the intrinsic program parallelism). The compiler can analyze the code on a **wider instruction window** than the hardware. The compiler has more time to analyze data dependencies and to extract more parallelism.

There is also a **reduced hardware complexity** because the complexity is moved to the compiler level. A small die area means cheaper processor cost and lower power consumption. It is also easily extendable to a large number of FUs. Moreover, instructions have fixed fields, therefore we can use a simpler decode logic.

6.3 VLIW code scheduling

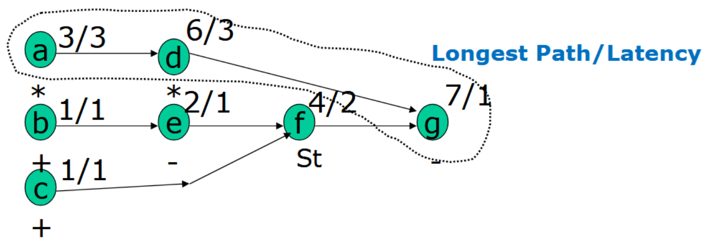
The main goal is to statically reorder instructions in object code so that they are executed in a minimum amount of time, and in a semantically correct order.

First, decompose the code in **basic blocks** (sequence of operations with no branches). The code in a basic block has:

- **One single entry point**: no code line within the basic block is the destination of a branch/jump anywhere in the program.
- **One single exit point**: only the last instruction can cause the program to begin executing code in a different basic block.

6.3.1 Dependence graph

A **dependence graph** captures true, anti, and output dependencies between instructions. Anti and output dependencies are named dependencies due to variables/registers reuse.



Longest Path(i) = Max (LP(Pred(i))) + Latency(i)

Longest Critical Path = Max (LP(i)) = 7

- Each node represents an **operation** in the basic block;
- There is an edge from a node i to another node j in the graph **if** the results of the operation i are used by the operation j (node i is a *direct predecessor* of node j);
- Each node is annotated with: the **longest path** to reach it (i.e., the node latency).

The **longest critical path** is the longest path in the dependency graph.

The **scheduling problem** consists of assigning the cycle of execution to each operation in the graph. The simplest type of scheduling occurs when we want to execute the code with the minimum amount of time, given the latency of each node, and we don't care about the number of resources required.

We have a list of nodes (**Ready List**) whose predecessors are all scheduled and their operands are ready. We then simply assign each node in the Ready List as soon as possible (**ASAP algorithm**) without any constraint on the resources.

6.3.2 Lint-based scheduling algorithm

This is a **resource-constrained** scheduling algorithm.

Before scheduling begins, operations on top of the graph are inserted in the **ready set**. An operation is in the **ready set** if all of its predecessors have been scheduled and if the operands are ready.

Starting from the first cycle, for each cycle, try to schedule operations (nodes) from the **ready set** that can fill the available resource slots.

When more nodes are in the ready set, select the node with the highest **highest priority** (longest path to the end of the graph).

6.3.3 Local and global scheduling techniques

To exploit all the possible parallelism within each basic block, then the compiler must try to expand basic blocks and schedule instructions across basic blocks.

Local scheduling techniques operate within a single basic block (loop unrolling, software pipelining). **Global scheduling** techniques operate across basic blocks (trace scheduling, superblock scheduling).

Loop unrolling

The compiler must test if loop iterations are **independent** to each other. The compiler can increase the amount of available ILP by unrolling a loop: the loop body is replicated multiple times (depending on the *unrolling factor*), adjusting the loop termination code.

Pros:

- **Loop overhead** (number of counter increments and branches per loop) is minimized;
- Extends the **length of the basic block**, so the loop exposes more instructions that can be effectively scheduled to minimize NOP insertions.

Cons:

- Increases the **register pressure** (number of allocated registers) due to the need of register renaming to avoid name dependences;
- Increases the **code size and instruction cache misses**.

Loop-carried dependences

Loop-level analysis involves determining what data dependences exist among the operands across the iterations of a loop.

Loop-carried dependences occur when *data accesses* in later iterations are dependent on data values produces in earlier iterations.

Software pipelining

We can reorganize the loop in anew loop so that each new iteration executes instructions (stages) chosen from different iterations of the original loop. this technique is called **software pipelining**. It can be seen as a sort of symbolic loop unrolling.

Trace scheduling

We try to find parallelism across conditional branches (global code scheduling). It is composed in two steps:

1. **Trace Selection**: find the most likely sequence of basic blocks (trace) of long sequence of straight-line code (statically-predicted or profile-predicted).
2. **Trace Compaction**: squeeze the trace into few VLIW instructions. Need book-keeping (compensation) code in case the prediction is wrong.

This is a form of compiler-generated speculation. The compiler must generate **fixup** code to handle cases in which the trace is wrong (misprediction). Needs extra registers and time to undo badly predicted instructions.

Superblock scheduling

It is an extension/optimization of trace scheduling.

A **superblock** is a group of basic blocks with a single entrance and multiple control exits. Superblocks are constructed by profiling the application and by duplicating tails (blocks after an entrance in the trace).

Advantages:

- Optimization is simpler because there are no side entrances.
- We need to create compensation code only for the exits and not for the entrance.

7 Memory hierarchy

Programmers want unlimited amounts of memory with low latency. However, fast memory technology (SRAM) is more expensive per bit than slower memory (SRAM).

The *solution* is to organize memory system into **hierarchy**:

- The entire addressable memory space is available in larger and slower memory;
- There are incrementally slower and faster memories, each containing a copy of a subset of the memory below it.

Temporal and spatial locality ensures that nearly all references can be found in smaller memories. This gives the illusion of a large, fast memory being presented to the processor.

7.1 Classifying cache misses

There are three major categories of cache misses:

1. **Compulsory misses** (or **cold start misses**) - The first access to a block is not in the cache, so the block must be loaded in the cache from the MM. There are compulsory misses even in an *infinite cache* (they are independent of the cache size).
2. **Capacity misses** - If the cache cannot contain all the blocks needed during the execution of a program, *capacity misses* will occur due to blocks being replaces and later retrieved. Can be *reduced* with larger cache size.
3. **Conflict misses** - If the block-placement strategy is set associative or direct-mapped, conflict misses will occur because a block can be replaces and later retrieved when other blocks map to the same location in the cache.

Can be *reduced* with larger cache or increasing associativity (by definition, fully associative caches avoid all conflict misses).

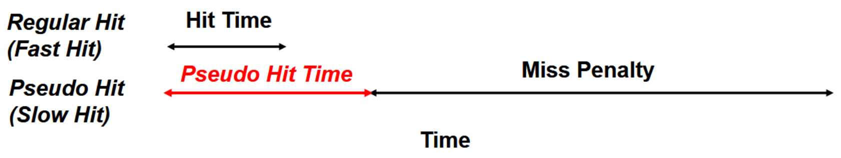
7.2 Improving cache performance

Average Memory Access Time:

$$\text{AMAT} = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty}$$

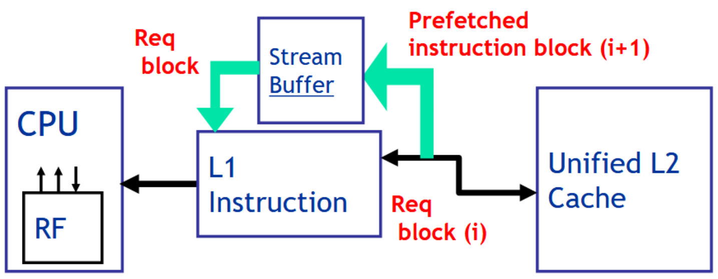
7.2.1 How to reduce the miss rate

- **Larger cache size** - (*Drawbacks*: increases hit time, area, power consumption, and cost).
- **Larger block size** - Reduces the miss rate up to a point when the block size is too large with respect to the cache size (*Drawbacks*: larger block increase miss penalty, and reduce the number of block, so increase conflict misses if the cache is small),
- **Higher associativity** - Decreases conflict misses (*Drawbacks*: the complexity increases hit time, are, power consumption, and cost).
- **Victim Cache** - A small associative cache used as a buffer to place data discarder from cache to better exploit temporal locality.
  It is placed between cache and its refilling path towards the next lower-level in the hierarchy.
  Victim cache is checked on a miss to see if it has the required data before going to lower-level memory. If the block is found in victim cache, the victim block and the cache block are swapped.
- **Pseudo-Associativity** and **Way Prediction**:
  - **Way Prediction** - In 2-way associative caches, use extra bits to predict, for each set, which of the two ways to try on the next cache access (predict the way to pre-set the mux):
    * If the way prediction is correct $\implies$ Hit time
    * If way misprediction $\implies$ **Pseudo** hit time in the other way and change the way predictor
    * Otherwise go to the lower level of hierarchy (Miss penalty)
  - **Pseudo-Associativity** - In direct mapped caches, divide the cache in two banks in a sort of associativity:
    * If the bank prediction is correct $\implies$ Hit time
    * If misprediction on the first bank, check the other bank to see if it's there. If so, we have a **pseudo-hit** (slow hit) and change the bank predictor
    * Otherwise go to the lower level of hierarchy (Miss penalty)



- **Hardware pre-fetching of instructions and data** - The basic idea is to exploit locality, pre-fetch next instructions (or data) before they are requested by the processor. Pre-fetching can be done in cache or in an *external stream buffer*.
- **Instruction pre-fetching**: fetch two blocks on a miss, the requested block (i) and the next consecutive block ($i+1$). The requested block is placed in the cache, while

the next block is placed in the *instruction stream buffer*. If miss in cache, but hit in stream buffer, move the stream buffer block into cache and pre-fetch the next block $(i + 2)$.



- **Data cache block pre-fetching**: relies on *extra memory bandwidth* that can be used without penalty (*Drawback*: if pre-fetching interferes with demand misses, it can lower performance).
- **Software pre-fetching data** - the compiler can help in reducing useless pre-fetching (*compiler-controlled pre-fetching*): compiler inserts pre-fetch LOAD instructions to load data in registers/cache before they are needed.
- **Compiler Optimizations** - The basic idea is to apply profiling on software applications, then use profiling info to apply code transformations: reorder instructions in memory so as to reduce conflict misses.
  Managing Data:
  - **Merging arrays**: improve *spatial locality* by single array of compound elements vs 2 arrays (to operate on data in the same cache block).
  - **Loop interchange**: improve *spatial locality* by changing loops nesting to access data in the order stored in memory (re-ordering maximized re-use of data in a cache block).
  - **Loop fusion**: improve *spatial locality* by combining 2 independent loops that have same looping and some variables overlap.
  - **Loop blocking**: improve *temporal locality* by accessing sub-blocks of data repeatedly (instead of accessing by entire columns or rows).

7.2.2 How to reduce the miss penalty

- **Read priority over write on miss** - The idea is to give higher priority to read misses over writes to reduce the miss penalty. The **write buffer** must be properly sized, larger than usual to keep more writes in hold.
  - *Drawback*: this approach can complicate the memory access because the **write buffer** might hold the updated value of a memory location needed on a read miss $\implies$ RAW hazards through memory.
  - **Write through** with write buffers might generate RAW conflicts with main memory reads on cache misses.

    * Check the contents of the write buffer on a read miss: *if there are no conflicts*, let the memory access continue sending the read miss before the write.

    * Otherwise, the read miss has to wait until the write buffer is empty, but this might increase the read miss penalty.

- **Sub-block placement** - Don't have to lead the full block on a miss, instead **move sub-blocks**. We need valid bits per sub-blocks to indicate validity. *Drawback*: it's not exploiting spatial locality enough.

- **Early restart and critical word first** - Usually the CPU need just one word of the block on a miss. The idea is that we *don't wait for the full block* to be loaded before restarting the CPU:

  - **Early restart**: request the words in *normal order* from memory, but as soon as the requested word of the block arrives, send it to the CPU to let the CPU continue execution, while filling in the rest of the words in the cache block.

  - **Critical word first** (or requested word first): request the missed word first from memory and send it to the CPU as soon as it arrives to let the CPU continue the execution, while filling the rest of the words in the cache block.

- **Non-blocking caches (hit under miss)** - Allows data cache to continue to supply cache hits during a previous miss (hit under miss). *Hit under miss* reduces the effective miss penalty by working during a miss instead of stalling CPUs on misses: requires *out-of-order* execution CPU: the CPU needs to not stall on a cache miss.

- **Second Level Cache** - A second level cache (L2) is introduced. The L1 cache is small enough to match the fast CPU clock cycle, while the L2 cache is large enough to capture many accesses that would go to main memory, reducing the effective miss penalty. The L2 cache hit time is not tied to the CPU clock cycle.

$$\begin{aligned} & \text{AMAT} = \text{Hit Time}_{\text{L1}} + \text{Miss Rate}_{\text{L1}} \times \text{Miss Penalty}_{\text{L1}} \\ \text{where} \quad & \text{Miss Penalty}_{\text{L1}} = \text{Hit Time}_{\text{L2}} + \text{Miss Rate}_{\text{L2}} \times \text{Miss Penalty}_{\text{L2}} \end{aligned}$$

- **Merging write buffer** - The goal is to reduce stalls due to full write buffer. Each entry of the write buffer can merge words from different memory addresses.

7.2.3 How to reduce the hit time

- **Small and simple L1 caches** - Direct-mapped caches can overlap tag compare and transmission of data. Lower associativity reduces power because fewer cache lines are accesses.

- **Avoiding address translation** - Every time a process is switched logically, we must flush the cache, otherwise we get false hits (with a time cost to flush). Dealing with *aliases* allows to have two different virtual addresses that map to the same physical address. To avoid virtual address translation during indexing of cache: if index is physical part of address, can start tag access in parallel with address translation so that can compare to physical tag.

- **Pipelined writes** - The idea is to Tag Check and Update Cache Data as separate stages: current write tag check and previous write cache updated. The *delayed write buffer* must be checked on reads, either complete write or read from write buffer.

8 Multithreading

8.1 What is a thread

A thread is a lightweight process with its own instructions and data. The threads can be created either **explicitly** by the programmer, or **implicitly** by the OS. The amount of computation assigned to each thread is the **grain size** (can go from a few instructions to hundreds or thousands).

8.2 Hardware multithreading

If multithreading is implemented at the hardware level, there is non need for context switch with the intervention of the operating system. Multithreading is manages directly by the processor architecture, using additional resources.

When the processor switches between different threads, the **state** of each thread must be preserved, while the processor switches. We need **multiple Register Files and multiple Program Counters**. Other parts, such as the functional units, can be **shared** among threads.

The processor must **duplicate** the independent state of each thread: separate copy of the register set and separate PC for each thread. Multithreading allows multiple threads to **share** the functional units of a single processor. The memory address space can be **shared** through the virtual memory mechanism. The HW must support the ability to change to a different thread relatively quickly (more efficiently than a process context switch).

There are different types of multithreading for a superscalar processor:

- **Coarse-grained multithreading**: when a thread is stalled, perhaps for a cache miss, another thread can be executed.
- **Fine-grained multithreading**: switching from one thread to another thread on each instruction.
- **Simultaneous multithreading**: multiple thread are using the multiple issue slots in a single clock cycle.

8.3 Superscalar with no multithreading

The use of issue slots is limited by a lack of ILP. Multiple-issue processors often have more functional units parallelism available than a single thread can effectively use by ILP.

A long stall, such as an instruction cache miss, can leave the entire processor idle for some clock cycles. The basic idea is that, in the empty slots (due to a long stall of the single black thread), put another thread.

8.4 Coarse-grained multithreading

Long stalls (such as L2 cache misses) are hidden by switching to another thread that uses the resources of the processor. This reduces the number of idle cycles, but:

- Within each clock, ILP limitations still lead to empty issue slots
- When there is one stall, it is necessary to empty the pipeline before starting the new thread
- The new thread has a pipeline start-up period with some idle cycles remaining and loss of throughput.
- Because of this start-up overhead, coarse grained MT is better for reducing penalty of high-cost stalls, where pipeline refill time is much less than stall time.

8.5 Fine-grained multithreading

The basic idea is that at each clock cycle we must switch to another thread, in a sort of round-robin among active threads. It switches between threads on each instruction, skipping any thread that is stalled at that time, eliminating fully empty slots:

- The processor must be able to switch threads on every cycle.
- It can hide both short and long stalls, since instructions from other threads are executed when one thread stalls.
- It slows down the execution of individual threads, since a thread that is ready to execute without stalls will be delayed by another thread.
- Within each clock, ILP limitations still lead to empty issue slots.

8.6 Simultaneous multithreading (SMT)

The threads in an SMT design are all sharing just one processor core, and just one set of caches. This has major performance downsides compared to a true multiprocessor.

On the other hand, applications which are limited primarily by memory latency (but not memory bandwidth), such as database systems, benefit dramatically from SMT, since it offers an effective way of using the otherwise idle time during cache misses.

Thus, SMT presents a very complex and application-specific performance scenario.

9 Data-level parallelism in SIMD and Vector architectures

9.1 ILP, DLP

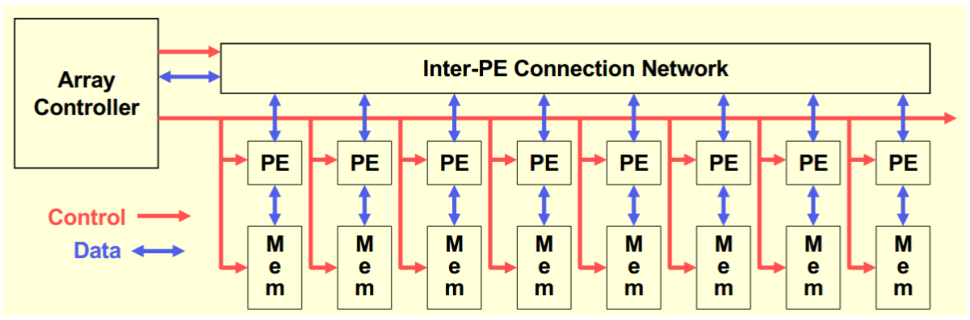
Instead of going in the direction of complex, out-of-order **ILP** processor, and in-order processor can achieve the same performance, or more, by exploiting **DLP** (**Data-level parallelism**): same instruction to manage parallel data, and with more energy efficiency.

9.2 SIMD architecture

SIMD (Single Instruction stream, Multiple Data stream) architectures can exploit significant **data-level parallelism**. SIMD is more efficient than MIMD (Multiple Instruction streams, Multiple Data streams), as they only need to fetch one instruction per data operation.

Also, SIMD allows programmers to **continue to think sequentially** and achieve **parallel speedups** (compared to MIMD that requires parallel programming).

There is a central controller that sends the same instruction to multiple processing elements (PEs) for multiple data.



- Only requires one array controller
- Only requires storage for one copy of the sequential program
- All parallel computations are fully synchronized

The PEs are synchronized with a **single Program Counter**. Each Processing Element (PE) has its own set of data, use different sets of register addresses.

The cost of control unit is shared by all execution units. Only one copy of the code in execution is necessary.

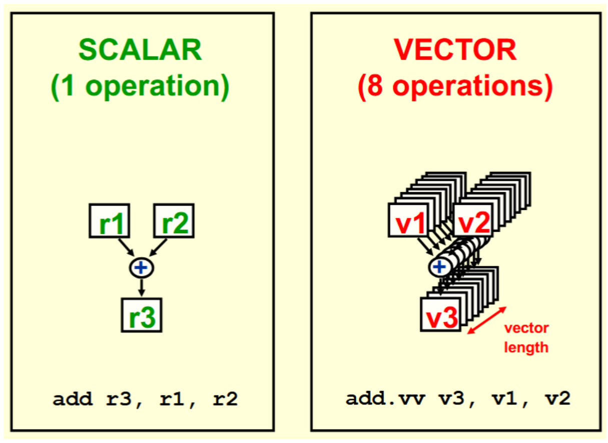
9.3 Vector architectures

They are a variation of SIMD machines. The basic idea is to lead **sets** of data elements into **vector registers**, operate on **vector registers**, and write the results back into memory.

A single instruction operates on **vectors of data**:

- Synchronized processing units: **single Program Counter**
- Register-to-register operations
- Used to hide memory latency (memory latency occurs one per vector load/store vs. one per element load/store).
- Leverage memory bandwidth.

Vector processors have operations that work on linear arrays of data: ”vectors”.



Each vector has 8-elements with n-bits/element. Each element is independent to any other.

9.4 Properties of vector processors

- Each result is **independent** of previous result, this implies:
  - $\Longrightarrow$ Long pipeline, the compiler ensures no dependences
  - $\Longrightarrow$ High clock rate
- Vector instructions access memory with known pattern:
  - $\Longrightarrow$ Highly interleaved memory
  - $\Longrightarrow$ Amortize memory latency of over 64 elements
  - $\Longrightarrow$ No data cache required (but there is still instruction cache)
- Reduces the number of branches and branch problems in pipelines
- Single vector instruction implies lots of work:
  - $\Longrightarrow$ Fewer instruction fetches

9.5 Components of vector processors

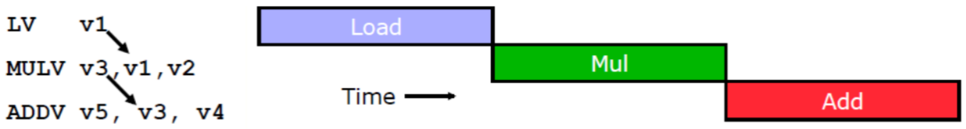
- **Vector Register File**: fixed length register bank holding multiple-element vectors
  - Has at least 2 read and 1 write ports
  - Typically 8-32 vector registers, each holding 64-128 bit elements
- **Vector Functional Units (FUs)**: fully pipelined, start new operation every clock
  - Typically 4 to 8 FUs
- **Vector Load-Store Units (LSUs)**: fully pipelined unit to load or store a vector, may have multiple LSUs
- **Scalar Registers**: single element for FP scalar or address
- **Cross-bar** to connect FUs, LSUs, registers

9.6 Operation chaining

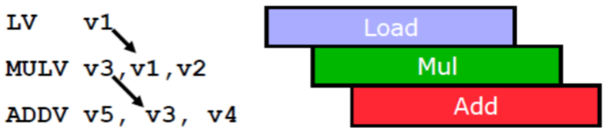
The results from each FU are forwarded to the next FU in the chain. The concept of forwarding is extended to vector registers: a vector operation can start as soon as each element of its vector source operand become available.

Even though a pair of operations depend on one another, *chaining allows the operations to proceed in parallel on separate elements of the vector*. In this way, we don't need to wait anymore for the last element of a load to start the next dependent instruction.

Without chaining, we must wait for the last element of one instruction to be written before starting the next dependent instruction.



With chaining a dependent operation can start as soon as each element of its vector source operand become available.



Convoys

Convoys are a set of vector instructions that could potentially execute together partially overlapped (no structural hazards). Sequences with read-after-write dependency hazards can be in the same convoy via **chaining**.

Chimes

Chime is a timing metric corresponding to the unit of time to execute one convoy. m **convoys** execute in m **chimes**. Simply stated, for a vector of length n, and m convoys in a program, $n \times m$ clock cycles are required.

Chime approximation ignores some processor-specific overheads.

9.7 Vector length control

The **Maximum Vector Length (MVL)** is the physical length of vector registers in a machine (64 in VMIPS). What to do when the vector length in a program is not exactly 64?

1. **Vector Length smaller than 64** - There is a special register, called vector-length register (**VLR**). The VLR controls the length of any vector operation (including vector load/store). It can be set to any value **smaller** than MVL.
2. **Vector Length unknown at compile time** - Restructure the code using a technique called **strip mining**:

- Code generation technique such that each vector operation is done for a size less than or equal to MVL.
- Sort of loop unrolling where the length of the first segment is $(n \mod MVL)$, and all subsequent segments are of length MVL.

9.8 Vector mask registers

When there is an IF statement inside the FOR loop code to be vectorized, the loop cannot normally be vectorized. Use a **vector mask register** to "disable" some elements:

- The vector mask uses a Boolean vector of length MVL to control the execution of a vector instruction.
- When vector mask registers are enabled, any vector instruction operates ONLY on the vector elements whose corresponding masks bits are set to 1.

The cycles for non-executed operation elements are lost, but the loop can still be vectorized.

9.9 Memory banks

Memory system must be designed to support **high bandwidth** for vector loads and stores.

Spread accesses across **multiple banks**:

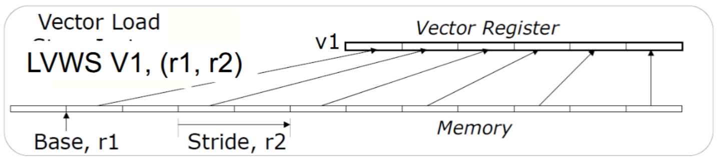
- Control bank addresses independently
- Load or store non sequential words
- Support multiple vector processors sharing the same memory
- Startup time: time to get first word from memory to registers

9.10 Stride

When a **matrix** is allocated in memory, it is linearized and laid out in row-major order in C $\implies$ the elements in the columns are not-adjacent in memory. To handle non-adjacent memory elements, we use the **stride**: the distance separating memory elements that are to be gathered into a single register.

When the elements of a matrix in the inner loop are accessed by column, the are separated in memory by a **stride** equal to the row size times 8 bytes per entry.

We need an instruction **LVWS** to lead elements of a vector that are non-adjacent in memory from address R1 with stride R2.



10 GPGPU Computing

A Graphics Processing Unit (GPU) is a full device equipped with a highly parallel microprocessor (many-core) and a high bandwidth private memory:

- High streaming throughput
- Fine-grain SIMD parallelism
- Low-latency floating point (FP) computation

Born in response to the growing demand for high-definition 3D rendering graphic applications, as they are specialized for parallel intensive computation. All operations are performed in parallel by the GPU using a large number of **threads** processing all data independently.

The target of a GPU is the **throughput**, not **latency** (as for CPUs).

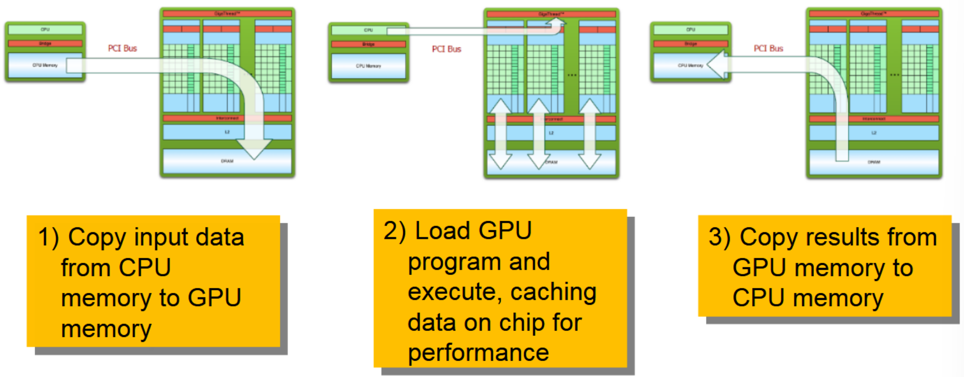
Many scientific/technical applications process large data sets and use data-parallel programming models to speed up the computation. GPU graphics API were not suitable to scientific applications, but the idea remained valid $\implies$ General Purpose GPU computing (GPGPU)

10.1 GPGPU programming model

The GPU (device) serves as a **coprocessor** for the CPU (host):

- CPU and GPU are separate devices, with separate memory addresses
- The GPU has its own high-bandwidth memory
- CPU and GPU should work together for maximum performance

Serial parts of a program run on the CPU (host), but **computation-intensive** and **data-parallel** parts are offloaded to the GPU (device). Data is moved between device memory and host memory when needed.


PCI Bus
PCI Bus
PCI Bus
1) Copy input data from CPU memory to GPU memory
2) Load GPU program and execute, caching data on chip for performance
3) Copy results from GPU memory to CPU memory

The data movement between CPU and GPU is the main bottleneck:

- Low bandwidth with respect to internal CPU and GPU since it exploits PCI Express
- Relatively high latency
- Data transfer can take more than the actual computation

This is especially a problem when porting CPU applications to GPGPU:

- Ignoring or not treating data movement seriously destroys GPU performance benefits.
- Some programming solutions may hide/automate data transfer

10.2 CUDA basics

CUDA is a parallel computing architecture and programming model that expose the computational horsepower of **NVIDIA GPUs**. It has its own C/C++/Fortran compiler and enables GPGPU computing with minimal effort:

- Write a program for one thread
- Instantiate it on many parallel threads
- Low learning curve, no knowledge of graphics is required

The programming model evolves to reflect the underlying hardware architecture.

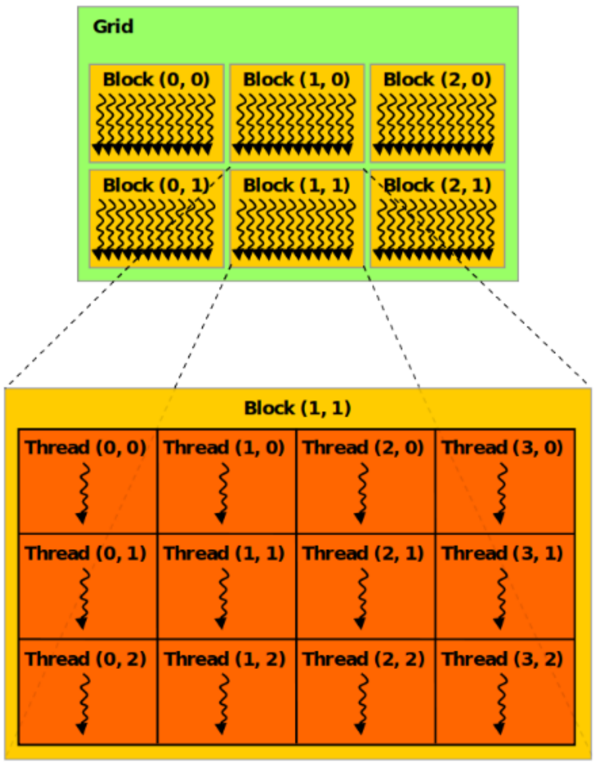
A function which runs on a GPU is called **kernel**:

- When a kernel is launched on a GPU, thousands of threads will execute its code
- The programmer decides the number of threads
- Each thread acts on different data elements independently

10.3 CUDA thread hierarchy

- Threads are grouped together in blocks
- Blocks are grouped together in grids
- Blocks and grids are organized as N-dimensional (up to 3) arrays

Threads that belong to the same block can cooperate through shared memory (SM). IDs are used to identify threads and blocks.



Dividing threads into blocks improves scalability.

10.4 Warp scheduling

Blocks are divided in warps. **Warps** are scheduling units on the SM. All threads in a warp execute the same instruction (SIMD):

- Control unit for instruction fetch, decode, and control is shared among multiple processing units.
- Control overhead is minimized.

10.5 Control divergence

Control divergence occurs when threads in a warp take different control flow paths by making different control decisions (is/else, different number of loop iterations).

The execution of threads taking different paths are serialized in current GPUs:

- The control paths taken by the threads in a warp are traversed one at a time
- During the execution of each path, only the threads taking that path will be executed in parallel

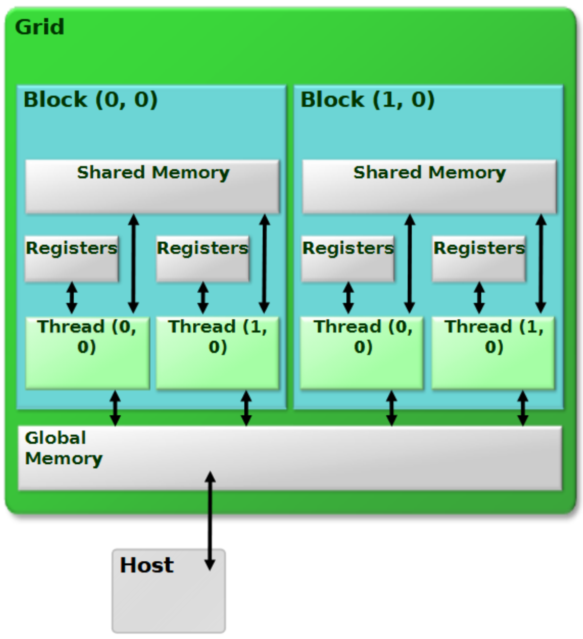
10.6 CUDA memory hierarchy

Three types of memory are available, matching the hardware architecture:

- Per-thread Private Local Memory (Registers)

- Per-block Shared Memory (Cache)
- Global Memory (off-chip DRAM)

Host can transfer data to/from global memory.



10.7 GPU execution model

A GPU kernel is invoked:

- Each thread block is assigned to a SM
- When a block completes, the runtime system assigns another one to the SM
- Threads within a block are divided in warps
- The scheduler selects warps for execution on the SM cores