Advanced Computer Architecures notes

Elia Ravella

April 19, 2021

Contents

Part I
Introduction

1 Taxonomy of Computer Architectures

1.1 Flynn Taxonomy

Flynn defined four categories to group computing architectures. This so called taxonomy helps system architects to distinguish among possible models of processors or system themselves.

1. SISD: Single Instruction Single Data, uniprocessor systems;
2. MISD: Multiple Instructions Single Data, no commercial application;
3. SIMD: Single Instruction Multiple Data, low overhead, used in custom integrated circuits;
4. MIMD: Multiple Instruction Multiple Data, off the shelf micros

SISD is the simplest and most diffuse model of computation, one data stream in input and a *single* instruction executed per istant of time. SIMD introduces parallelism in a vector fashion: multiple processes execute a single instruction (the *same* instruction) on different data elements. Another type of parallelism: MIMD. Just SIMD with multiple different instructions per instant.

Part II
Performance and Cost

2 Performance

2.1 Evaluating performance

We need to distinguish between how to measure performance of a computer architecture and what impacts it without affecting performance. The amount of memory used is not a performance evaluator, but it affects performance in some way. The power consumed, the latency and the execution time are. To evaluate the performance of a CA I must use all the performance indicator and find the right tradeoff. Also, context! A CA is inserted in a context that bounds its scope, so the environment itself shapes the way performance is evaluated
Every performance indicator says something about the CA. So, the whole system must be evaluated with a *whole set* of indexes that each tell a different thing about the system itself. This provides a full view of the system strenghts and weaknesses and highlights the things to optimize.

2.2 Quantifying the Design Process

"The frequent case" case: when designing a CA, the "frequent case" represents the set of conditions met *the most times* during a standard computation run. Optimizing the standard case means optimizing *singnificately* all the system behaviour.

Amdahl's Law Amdahl's law describes the concept of the speedup as a function of execution time. The set of formulas that describes the law is

$$EXTIME_{new} = EXTIME_{old} \times [(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}] \tag{1}$$

$$SPEEDUP_{overall} = \frac{EXTIME_{old}}{EXTIME_{new}} = (1 - Fraction_{enhanced} + \frac{Fraction_{enhanced}}{Speedup_{enhanced}})^{-1} \tag{2}$$

and calculates the speedup of a system taking into account all his part, the speedupped one and the invariate one.

2.3 Indexes on performance

Overview of performance indicators and measurements.

- Instruction Count: the effective number of instructions executed. This is influenced also by the program we're trying to execute.
- Tyme per Cycle: the clock rate. OBVIOUSLY influenced by the hardware architecture.
- MIPS and MFLOPS indicates already a very *program bound*-like perfomance indicator. They represents the number of integer/floating point operations that the architecture can carry out in a second. They're very synthetic indexes.
- CPI - Average CPI (clock per instruction) represents obviously yhe amount of clock cicles that are necessary to carry out a specific operation. The element that most impacts the CPI is the ISA: the alphabet we use shapes the words we can create.

Notice how execution time is not present. That's because *reducing execution time is the goal*, so execution time does not modify performance, it just *represents* it.

2.4 Benchmarking

Benchmarking is "running a fake program with the sole purpose to evaluate a system and find bottlenecks". It's a technical test. Workloading a CA means feeding it with a real work load to test what is a response to a non-syntethic stimulus.
Benchmarking must be

1. representative: we need to benchmark the right aspects of the architecture we're testing.

2. updated: old benchmark does not test the architecture well, because of the very architecture is changed and the benchmark is *not anymore representative* because it's *old*.

2.5 Energy, Power

Energy and power consumption are a "cost - like" performance indicator. The less the power, the less the cost of keep the architecture running, and tha's easy. Also, reducing the power consumed means *reducing the power needed from a battery* and this impacts a lot in the mobile scenario. Energy per task is a good indicator for energy consumption. The power used by an architecture is influenced by a ton of factors, not least the TDP (thermal design power). Also architecture-bound aspects (clock rate, voltages, busses) influence power consumption *heavily*.

Part III
Pipelining

3 MIPS architecture

Reference architecture for this course is MIPS.
MIPS is a RISC architecture (Reduced Instruction Set Computer) this means that the only possible operation (that are very optimized) are basic operations. This kind of architecture is the opposite of CISC (Complex ISC) where more complex operation are supported at chip level.
MIPS is a LOAD/STORE architecture. This means that data cannot come into the ALU from anywhere besides the CPU general purpose registers. MIPS uses 32-bit instructions that can be

- Register Register operations
- Register Immediate operations
- Branch operations
- Jump/Call operations

Operand lenght is 6-bit. We have 64 possible operations.

3.1 Assembly basics

We'll see three classes of MIPS instructions:

- ALU: add "reg target" "reg source" "reg source", or sub "reg target" "reg source" "immediate"
- LOAD/STORE: lw "reg target" "offset register" (load word)
- Branches/Jumps: classic bne or bge or jmp operations

3.2 MIPS instruction execution

A fetch - decode - execute cycle in this architecture works this way:

1. Fetching the instruction means sending the PC content to the memory in order to access the memory location where the instruction is. Then, the PC is updated (PC + 4, each instruction take 32 bits).

2. Decoding the instruction: in this phase a fixed-field operation is put in place to deconde the instruction itself, and the registers needed for the computation are read.

3. Execute operation: the ALU calculates the result of the instruction supplied.

4. Memory Access and Write Back phases: in these two phases the CPU takes care of the memory access (LOAD or STORE instructions) and the registers update.

3.3 MIPS Chip Architecture

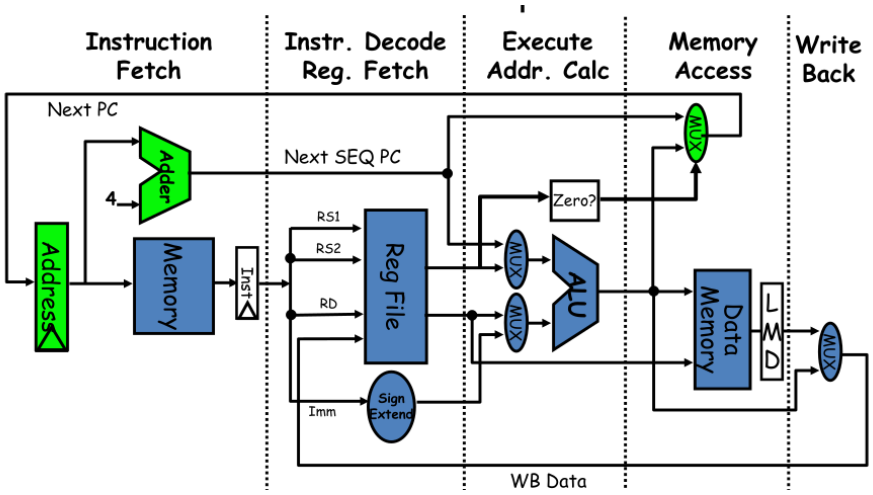


Figure 1: The MIPS data path with pipeline stages highlighted

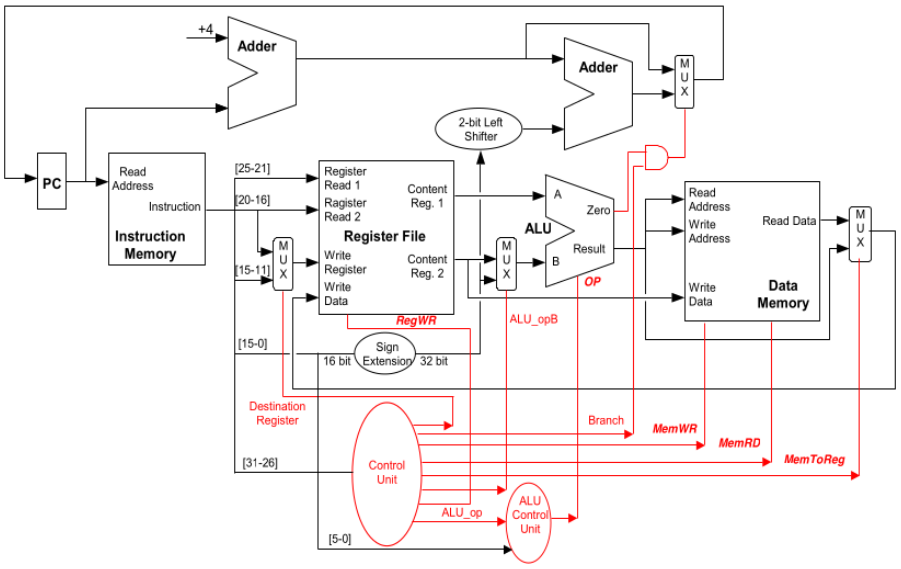


Figure 2: The MIPS full CPU, data path integrated with control unit

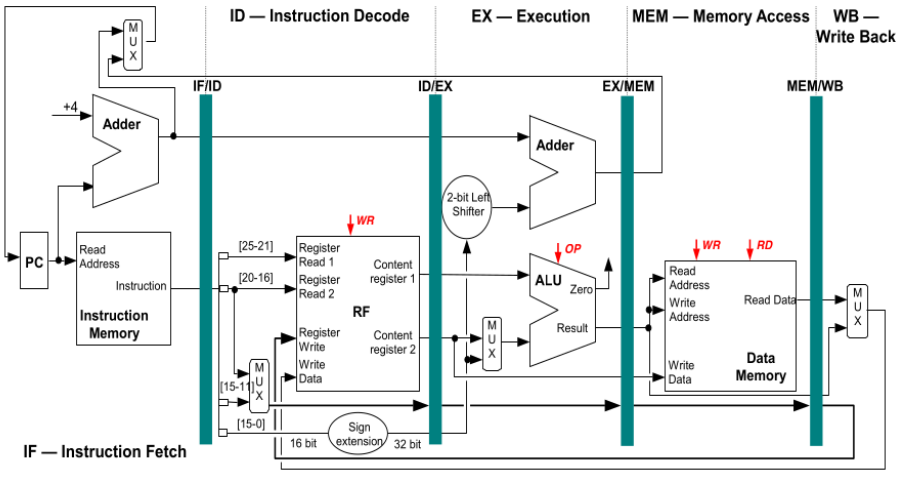


Figure 3: Pipelined MIPS

4 Data and Control

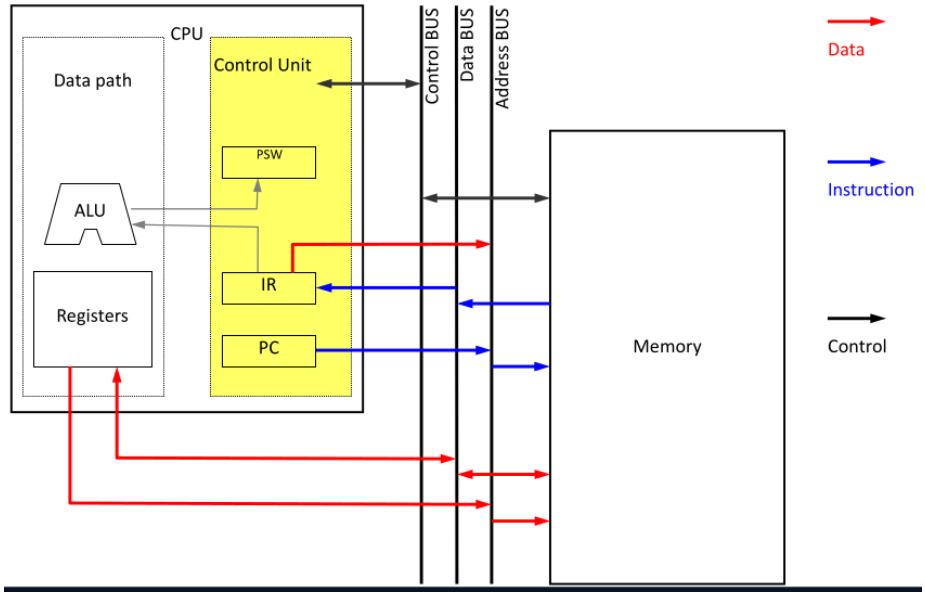
Every computer architecture is composed of 2 main parts: the one that computes data (datapath) and the one that controls the execution (control unit).

Datapath The datapath runs the house. All data pass through this module, and the registers here are used to pipeline operation.

Control Unit The CU guides the computations. It takes "run time decisions" in order to pilot execution of intructions. The registers here are architecture-

specific.

Memory And memory? "Slow primary memory" is connected with the CPU (so DP + CU) by means of buses. The structure is similar to:



Things to be noted:

1. the datapath does not connect to the control path
2. the control unit does not write on the data bus

5 Hazards

Pipelining is cool, parallele, throughput enhancing and overall faster. Does she have drawbacks? YES, hazards. Both the ID and the WB phases in the pipeline we have an access to the registers, that can overlap. This is usually overcome assigning (for example) rising edge of the clock to reading and falling edge to writing.
But what are hazards? An hazard is generated by dependency between two instructions. Hazards can be classificated in

1. Structural hazards: the registers example. Different phases of the same pipeline access simoultaneously the same resource.
2. Data hazards: attempting to *reading the future*: accessing a piece of data that's not be computed yet.
3. Control hazards: making a decision basing on a condition *not yet evaluated*.

5.1 Structural Hazards

No structural hazards in the reference architecture (edge - based synchronization)

5.2 Data Hazards

Utilizing a piece of data that is not ready is called the condition of data hazard (or data conflict). This is generally generated by two (or more) consecutive instructions that generate and use the same value.
We can further classify the types of data hazards:

Read after Write conflicts are generated by two consecutive operations, the first writes a value and the second reads it. In a pipelined environment, this can cause problems. This is the most common hazard.
This kind of conflicts can be handled at compilation time (the compiler itself detects a dependence of this kind and inserts delays in the execution ("bubbles" or "stalls") of the instructions to distanciate "enough" the read and write operation. The cost is incremented overhead and increased execution time. This approach can also be implemented *at architecture level*. The difference is that compiler will delay instructions with NOP instructions, while at architecture level clock cycle are just "skipped".
Another way to resolve this kind of conflicts is to reschedule instructions (taking advantage od the dependencies between them) in order to

1. distanciate enough the write-read conflicts
2. not inserting dead times

Forwarding, instead, is the technique that "bypasses" the WB phase and makes available the result of the ALU evaluation directly in the pipeline. So additional EX-EX, MEM-EX, MEM-ID paths are introduced in the architecture. This approach does not cover certain type of conflicts, like load - use hazards. This conflict, however, is the only one possible in the reference architecture: all the other data hazards are solved with the insertion of forwarding paths.

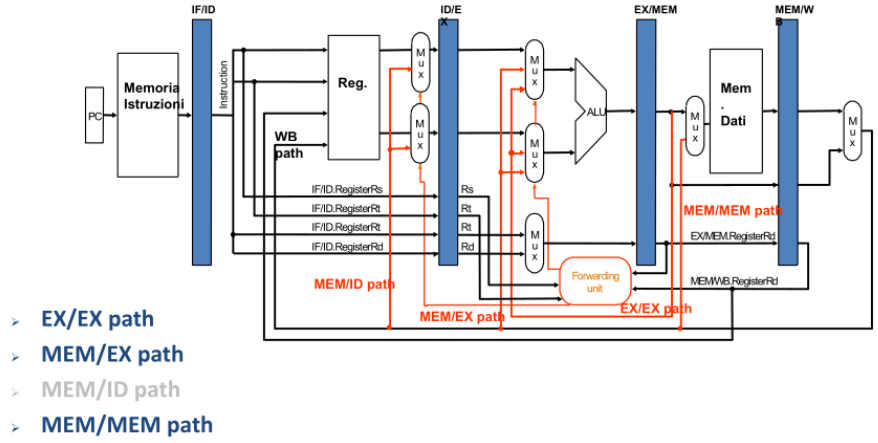


Figure 4: MIPSarchitecture with forwarding paths

Write after Write It's a overwriting conflict. This is a problem when a WB stage of a instructions happens *after* the WB phase of a *following* instruction. This conflict can't happen in the reference architecture: all instructions takes 5 clock cycles.

Write after Read In a out-of-order scenario we can have a write operation of an instruction happens *after* the read operation of an instruction *preceding* her. This, again, is only possible when instructions can take more stages at executing than the instructions following them, so not possible in the reference architecture.

5.3 Control Hazards

Conditional instructions at assembly level are jumps (oversimplifying). The jump target address is computed just after the condition is evaluated (it can be PC + 1 or "TargetAddress").
In the reference pipeline the PC is updated in the last pipeline stage, BUT we can skip the MEM phase because the PC is updated in the MEM phase itself. This kind of instructions (beq, bne) terminates after the MEM stage. This means that further instructions must wait for the MEM phase of the conditional statement in order to be fired, so *at least 4 cycles*.
The standard procedure to deal with this kind of instructions structure is to *preload in the pipeline the "true" branch*, and then flush it if the condition is not met. Also, data forwarding reduces the number of stalls to be introduced: EX-ID path, for example, reduces the number of stalls from 3 to 2.

Early Evaluation What if we anticipate the condition evaluation *the most that we can*? We can move an ALU for conditions in the second stage of the pipeline (ID) in order to recompute faster the condition. This reduces the number of bubbles needed to 1.
This technique combined with the "betting on the true branch" reduces the

scope of the pipeline flush needed in the case the bet result is bad. BUT early evaluation introduces data hazards when the same registers must be accessed by two consecutive instructions where the second one is a branch. Also, this approach does not free us from stalls: in fact, we still need to insert a single stall before the IF of the next instruction.

5.3.1 Branch Prediction

Ok, so we reduced the flush scope of a prediction to one instruction (with the early evaluation of branch condition). Can we go further? Yes, we can pre-load a branch of the conditional execution in order to anticipate the jump.
These techniques can be classified in two major families, static and dynamic, if they're fixed or acts basing on the data they've got.

Static Branch Prediction Techniques

- Branch Always Not Taken: the next instruction to be fetched is PC + 4. Easy.
- Branch Always Taken: next instruction to be fetched is the one computed by ID stage.
- Backward Taken Forward Not Taken: this techniques "try to predict basing on the direction of the jump". So the assumption is "forward jumps are conditions, we enters them; backward jumps are loops, we always take them".
- Profile Driven Prediction: statistic approach. Thismethod can use compiler hints.
- Delayed Branch: rescheduling. This approach takes an instruction "near" the branch instruction and reschedule it after the branch one, because either if the branch is taken or not *no penalty clock cycles* are to be taken. Obviously, the moved instruction must not be data dependent from the condition or the branch itself. There are three possible ways in which the *branch delay slot* (the first instruction after the branch instruction) can be filled:
  - From before: the instruction to fill the delay slot is taken from before the branch instruction
  - From target: the instruction is taken from the target of the branch (useful when loops are common: it's preloading the loop body)
  - From fall-through: the instruction to fill the delay slot is taken from the taken forward path

Dynamic Branch Prediction Techniques How can we enhance the "branch bet" choice at runtime? Basing on the data we gather at runtime, change the predicted branch. This is implemented with an associative table that links *PC bits with target addresses.* The whole predictor mechanism can be described by two parallel computations:

Branch Outcome Prediction BOP is the procedure that selects the actual label to jump to. If we're using a one-bit predictor, the behaviour is schematized by a super simple FSA that acts this way:

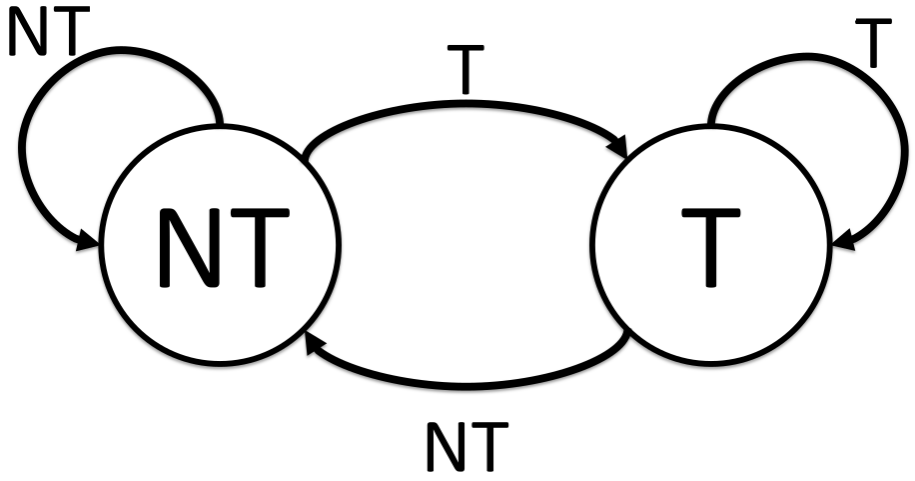
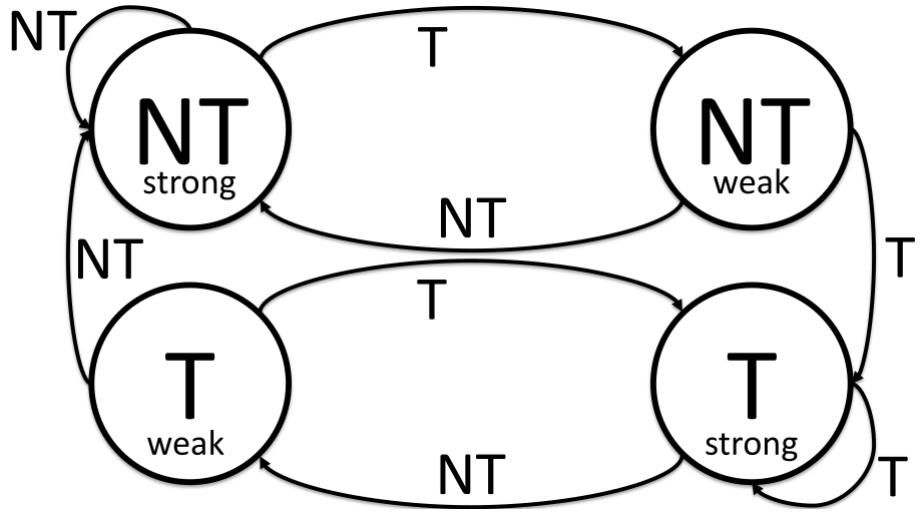


Figure 5: The NT stands for Not Taken, and T for Taken

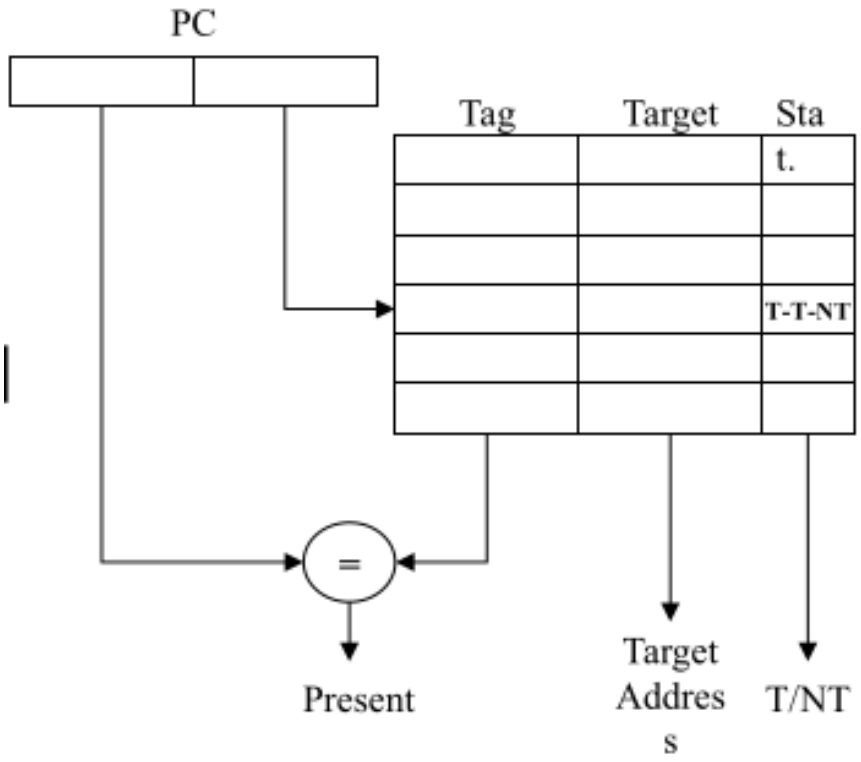
This models the behaviour of the predictor of *which branch is to be taken.* Important remark: the BHT FSA may *overlap predictions* (in the case for example of nested loops) if the prediction is based on different loops. An enhanced model use a 2-bit register to switch state:



We could also add a third, fourth bit to the memory of the FSA, but sperimental data show that a 2-bit BHT predictor is the best tradeoff between complexity and accuracy.
Another approach is to lenghten the scope of the prediction: what if two threads of execution have a similar pattern in branches, but they're "unlucky"? We

could add a dedicated branch predictor that matches previous behaviours with the current one to look for patterns and better decide which path to take. This "multilevel predictors" relies on 2 factors: the lenght of the *behaviour branch* to be analyzed **and stored** and the number of bits used to discrimine the branch taken (so as the simple predictor: in fact, simple FSA predictors are (0, 2) correlating predictors).
Correlating predictors, in practice, memorize the "story" of a execution path and then feed this story to a normal FSA predictor to compute the outcome.
Even more formally, a (m, n) predictor uses the behaviour of the last m branches to choose from 2^m branch predictors, each of which is a $n - bit$ predictor for a single branch. From the implementation point of view, the history of branches is memorized in a shift register of lenght m.

Branch Target Predictor Simply enough, the BTP is a cache that stores the predicted address to jump to if a branch is taken. It's structured as an hashmap of jump addresses $\rightarrow$ PC-relative jump targets. Every associative entry must also have some validity bits, in order to signal wheter the branch is "in use" or not. Usually, these bits are signaled by the Branch Outcome Predictor.



Part IV
Parallelism

6 Instruction Level Parallelism

"Potential overlapping of of execution among unrelated instructions". This definition of parallelism includes pipelined architectures: in fact, at every clock cycle, a pipeline is executing a number of instructions that's up to the number of stage it's composed of. *Actual* parallelism can be achieved by parallel architecuters (so hardware that is designed to handle multiple instructions per clock cycle) such as superscalar architecures and VLIW.
ILP is *not* parallel processing! The first is just pipeline operations to enhance throughput, the second is a non-user-transparent way of *executing programs*.
ILP is possible $\Leftrightarrow$ there are no WAR, WAW, RAW conflicts, neither strtuctural hazards or control stalls.
At first, we must introduce the Complex Pipeline in order to understand the aim of the procedures to implement ILP.

6.1 Complex Pipeline

The complex pipeline has multiple path between the DECODE step and the WRITE BACK phase. This means that *right after* the decode phase an additional phase (called "issue") that send the operation to the correct pipeline must be introduced. The complex pipeline *does not create real parallelism between instructions* because in any case the EXE stage will not be shared among instructions. It just introduces the problem of having to deal with *multiple data path* that each introduce a potential critical path to the architecture. Each one of those additional datapath can have a different clock count (an integer operation could take less than a "load word" operation, in terms of clock cycles) and thus enhance throughput. This, however, could lead to problems in the fetch phase of operations.
A possible solution (in fact, the superscalar once) is to load two instructions per clock cycle, and send them down different paths if they're "different" (basing on the dedicated hardware for each). The additional assumption is, every alternative EXE path is delayed in order to have a homogeneous clock cycles number before the WB stage.

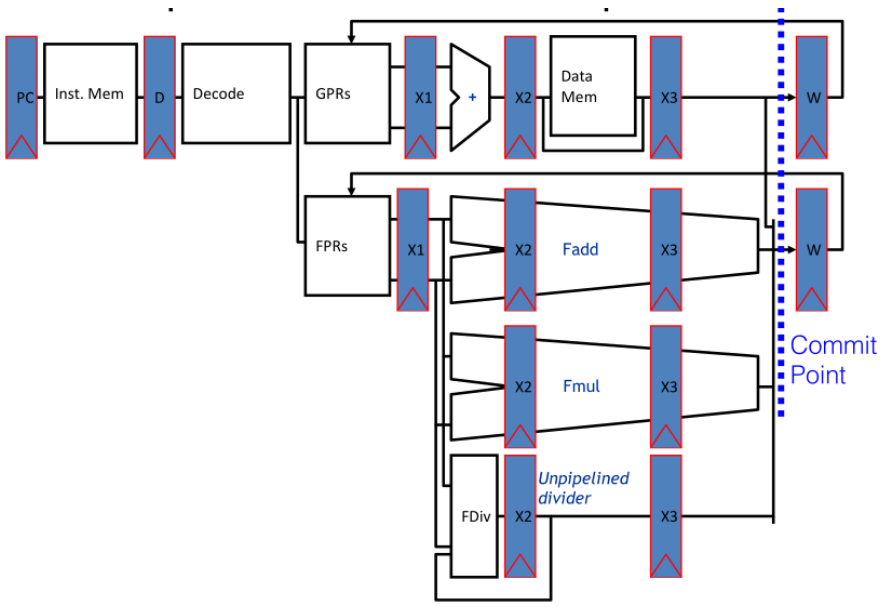


Figure 6: Here we see how the three different EXE stages are delayed-stalled all to match the longest data path (the FAdd, FMul one)

6.2 VLIW

Very Long Instruction Word allows actual parallelism between instructions. This is done with the aim to raise the CPI above one ⇒ having multiple instructions completed *simultaneously*. The idea is to delegate the scheduling of the operations *entirely* to the compiler, and enable the processor to fetch more than one operation per cycle. To do so, an additional distinction between instrutions and operations is needed:

- Operation is the basic unit of computation
- Instruction is a batch of operations; all the operations in the batch can be executed simultaneously

The compiler now has to arrange operations in these batches (the VLIW instructions) that (due to their dimension) justify the name "Very Long Instruction". The Fetch, Decode, Mem and Write Back phases are unchanged in these architectures, juste the EXE block is replicated and parallelized.

6.2.1 Enforcing ILP

In a regular program, it's difficult to find parallel-prone batches of instruction. The compiler has to figure out how to enhance code translation in order to fully use the parallel architecture underneat.

Loop Unrolling A simple example is the way loops that access sequential (or random access) data structure, just as a simple foreach, can be "unrolled", so more iteration of it can be executed at the same time. N.B.: this must be compatible with the conflicts in the code!

Software Pipelining In order to enhance Loop Unrolling performance, again rescheduling comes in hand. In which way? Basically "anticipating the next loop iteration" at a scheduling level. This is made by anticipating operations *in order to free slots in the pipeline that can be used to startup next iterations.* It's probable that some "compensating instructions" must be added.
We can see Software Pipelining as a Loop Unrolling *enhanced* because it's just LU that pays the "startup" and "wind down" operation *just one time per loop* instead of *once per iteration.*

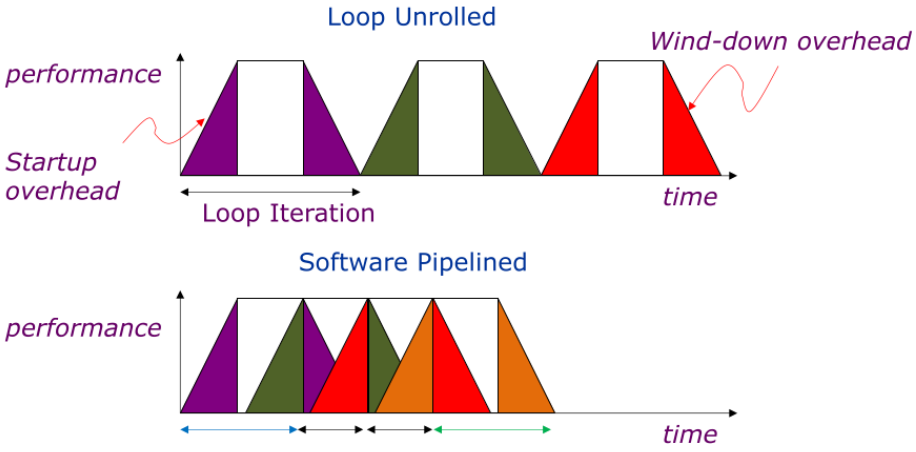


Figure 7: Visualization of the improvement of Software Pipelining

REMEMBER: SP is a *compiler* solution, that just *exploits efficently* a VLIW architeture.

Trace Scheduling A trace is a loop free sequence of basic blocks (of instructions) embedded in the control graph. It's an "execution path" for the application. The idea behind trace scheduling is to execute application and profile the traces (against fixed input) to find the *most probable to be executed.* Then, consider the entire string of "probable blocks" as a single one, in order to apply heavy rescheduling to the whole operations set, to enhance performance.

Messing with Code Compensating code (like decreasing a counter afterwards having anticipated the increment) can be super cool when it comes to rescheduling operations in order to exploit hardware parallelism. BUT it must be handled with care, just like rescheduling *itself.* There are some well established techniques:

- Speculative execution: "move around" code that we *think* (probabilistically) can be executed.
- Predicated execution: we use the parallelization at the hardware level to simultaneously execute different branches of a conditional block and then *validating* only the right one.

7 Dynamic Scheduling

So, dynamic scheduling is the procedure to "rearranging instructions/operations" in order to fully exploit the hardware. Keep in mind that we're talking about *hardware solutions* to reschedule operations, so the additional hardware needed to analyze instructions, look for dependencies and dispatch executions increases (significantly) the overall complexity of the architecture.

7.1 Scoreboard

7.1.1 Scoreboard Data Structures

In order to properly manage the complex pipeline and "give the right timing" to each instructions, functional unit and register, the scoreboard architecture makes use of additional data structures. The most important is a "hash map"-like stucture that links every functional unit to a set of flags and pointers, in detail:

1. A *busy* flag that signals wheter the FU is in use
2. A *op* vector that describes the available operation in that FU. This is also used to *tell multi functional units which is the desired operation*
3. Three pointers to the registers (one for the destination, two for the sources, obviously)
4. Two pointers to the functional units that *should produce the desired value* (this pointer is used to detect RAW conflicts) and that is linked to the source/destination register pointer (in fact, if we call F_i and F_j the two source registers, these two pointers would be Q_i, Q_j). More than pointers, this are FU ids, like "integer module 1" or "adder 2"
5. Two additional flags (R_j, R_k) that signals *when* F_j, F_i are ready. These two bits are used *alongside the other flags and pointers* to signal if the instruction is waiting for data, but their meaning changes depending on the type of instruction and the phase of it

This hashmap (which index is the FU) is called *functional unit status* and works with an additional array of signals the status of all registers, the *register result status*.

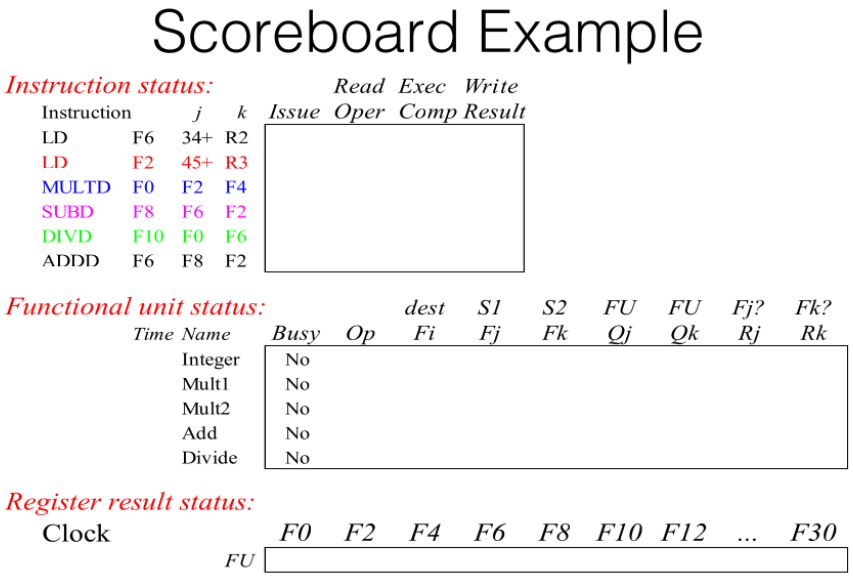


Figure 8: The scoreboard blank data structures

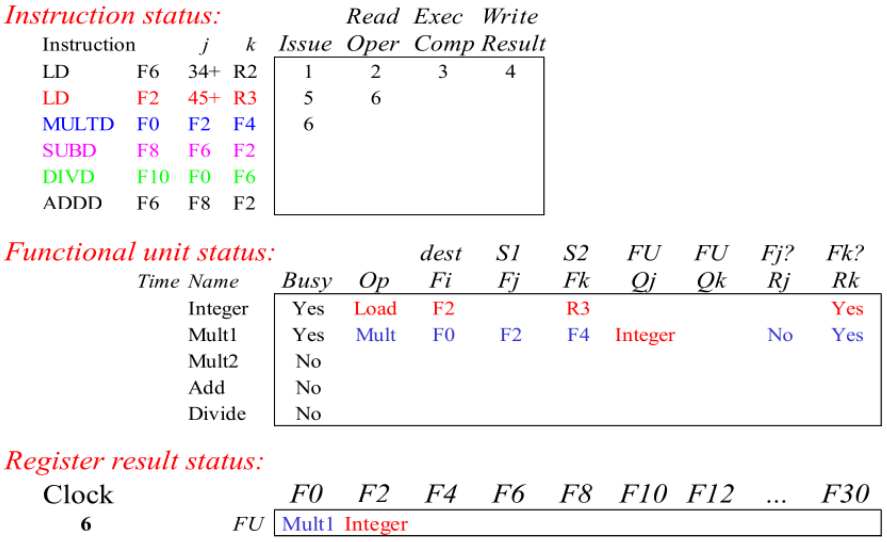


Figure 9: The scoreboard data structures executing an instruction

7.1.2 Scoreboard - The Algorithm

Environment: complex pipeline with "ISSUE" stage, we need to implement out of order execution. The basic solution is to split the ID stage in two, the first that analyzes the instruction and the second that *computes the operands and detects hazards*. An instruction is allowed to fire if there are no hazards. **So if**

instruction 2 follows instruction 1 and the latter is blocked due to an hazard *but 2 is free to fire*, 2 will execute normally.
That means that a scoreboard dynamic scheduled pipeline is

- In order issued
- out of order executed
- out of order committed (writebacked)

The Scoreboard pipeline is ultimately divided in 4 main stages:

1. The issue stage decodes the instruction and detects the hazards. Instructions are issued in program order. Stalling this phase is used to avoid WAW and structural hazards, and it's done if
   - There's not a functional unit ready for this type of operation
   - There's a conflict in the target register

   If no hazards are detected, the Read Operands stage is entered
2. Due to the fact that there's no possible forwarding in this pipeline, scoreboard hardware needs to tell each FU when it can read from registers. This is the stage to be stalled in order to avoid RAW hazards. An instruction is stalled here if one (or both) of its R_i is set to false.
3. Execution stage: the FU gets the operands when the scoreboard enables the Read Operands stage "right behind" the exe stage, and this stage notifies the scoreboard when the result is ready. The execution phase can take up different amount of clock cycles based on the kind of operation performed.
4. Accessing to the registers in order to write back the results of instructions execution. This stage can be stalled if a WAR is detected.

Ultimately, the Scoreboarding approach removes RAW hazards, by *issuing an operation only when its operands are available*, and WAR hazards, by *stalling the writeback phase if needed*. WAW hazards are still to be taken care of.

7.2 Tomasulo Algorithm

Tomasulo algorithm is a dynamic algorithm that enables out of order execution and out of order completion for instructions (issuing is still handled in order). The goal of this algorithm is the same of scoreboard: high performance without compiler intervention. The main difference between Tomasulo and Scoreboard is how the control logic is handled: in Scoreboard is centralized, while in Tomasulo approach are distributed and *nearer* to the functional units. Operand buffers are called *reservation station*. Tomasulo's architectures resolves WAW and WAR hazards efficiently by exploiting *register renaming*.

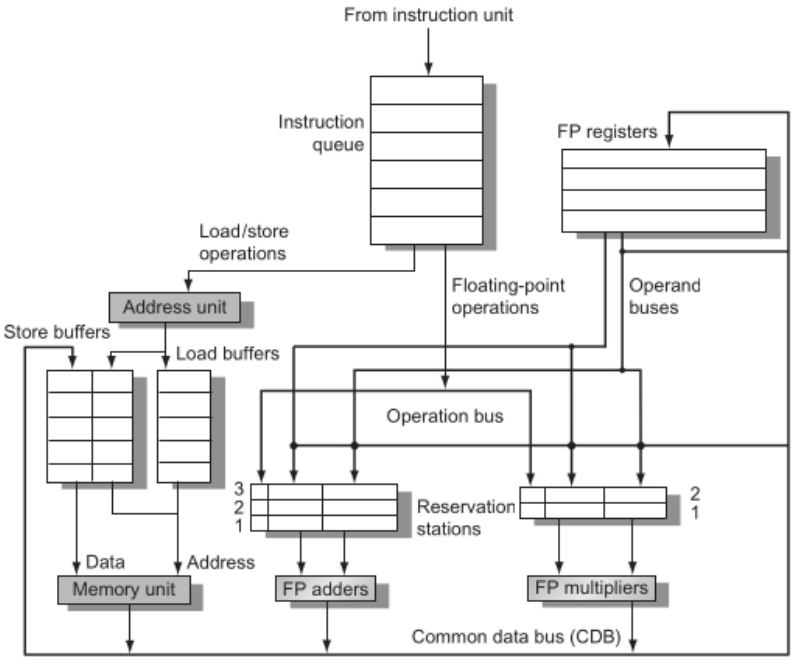


Figure 10: Basic Tomasulo architecture

7.2.1 The reservation stations

The reservation stations are the core of the Tomasulo mechanism. They're the buffer between the register file and the actual functional units, and *embody the register renaming approach* that lies under the Tomasulo algorithm. Also, proxies are introduced for the "load from memory" operations (load buffers) and for the write back operations (store buffers).
The reservation stations can access a common data bus (the hearth and nervous system of the Tomasulo architecture) that makes available / read the values broadcasted from Reservation Stations.

7.2.2 Tomasulo data structures

Tomasulo decentralized approach does not affect much the kind of information needed in solving this problem. In fact, the *very same information* is just scattered near the functional units, inside the reservation station. The reservation station, in fact, are made up of:

- A string indicating the type of operation needed
- V_j and V_k the values of the source operands (value = copies)
- Q_j and Q_k pointers to the reservation station that produces V_j and V_k. Can be used to *bypass* V_j and V_k
- a "busy" flag

7.2.3 Three stages Tomasulo architecture

The Tomasulo pipeline is divided in

ISSUE stage Get the instruction from the queue and if a RS is ready, *set the right values of it*, that can be either in the registers or in a issued computation; in the latter case, the pointers to the FU that will produce the right value are given to the issued intruction. If there's no station available, the operation is stalled: we're in presence of a structural hazards. In this step registers are renamed, avoiding WAR and WAW hazards.

EXECUTION stage If an operand is not ready *it will arrive from the shared data bus*. In this stage operands are "waited" monitoring the CDB, and each RS and FU knows which other will produce the desired value because of the ISSUE stage providing thi information earlier.

WRITE stage In this phase the result of the FU computations are broadcasted on the common data bus and stored in the RS that requested it and in the Store Buffers, in order to be stored in memory or in the RF eventually.

Part V
Interrupts and Exceptions + Handling

8 Interrupts

An interrupt is a "signal" that alter the normal control flow of a program, *at runtime*. An interrupt must be handled by a dedicated *interrupt handler* and then the architecture should be able to *resume the normal execution of the program*. An interrupt can come from the program that's executing as well as the external environment. An ACA definition of an interrupt is "an event that requests the attention of the processor". We can characterize interrupts in a super complicated way:

1. A/Synchronous
2. User requested or coerced
3. User maskable or unmaskable
4. Within or between instructions
5. Resume/Terminate events

8.1 Asynchronous Interrupts

Hardware signals, timer expirates are all async interrupts. They're all caused from external sources instead of the currently running program. These kind of

interrupts can be handled at the end of the execution of the current instruction, and thus they're far easy to manage wrt sync interrupts.

The Interrupt Handler When an external interrupt is invoked (an I/O change, for example) the CPU should switch the context of the execution from the currently executing program to the code of the interrupt handler, the piece of code that carries out the managing of the event. So all the instruction up to I_i, so I_{i-1}... must be completed (this defines a precise interrupt, also[1]), the PC must be saved and the execution must be transferred to a dedicated kernel-level interrupt handler.

8.2 Synchronous Interrupts

Synchronous interrupts (also called *exceptions*) are interrupts and errors generated by the curently running program. Undefined opcode, program error (zero division) or misaligned memory are all interrupts that come from the running program itself. These exceptions are rare, but they're problems, and they must be handled carefully.
A synchronous interrupt is caused by a particular instruction (so generally is a *within instruction* kind of interrupt) and generally cause the instruction itself (if not the entire routine) must be restarted.

[1] An interrupt is said to be precise if an istant can be identified so that all instruction before that moment can commit their state and no subsequent (to the interrupt) instruction modifies the saved state. Precise interrupts are desiderable because (obviously) *execution can be resumed easily* and because they generate a "safe state" to proceedback from.