1 Pipeline

1.1 MIPS

The MIPS architecture is based on the SIMD model, the instruction set (ISA) is based on 32-bit format instruction.

ALU Instructions: `op $x,$y,$z`

Instr. Fetch & PC Increm.	Read of Source Regs. `$y` and `$z`	ALU OP (`$y op $z`)	Write Back of Destinat. Reg. `$x`

Load Instructions: `lw $x,offset($y)`

Instr. Fetch & PC Increm.	Read of Base Reg. `$y`	ALU Op. (`$y+offset`)	Read Mem. `M($y+offset)`	Write Back of Destinat. Reg. `$x`

Store Instructions: `sw $x,offset($y)`

Instr. Fetch & PC Increm.	Read of Base Reg. `$y` & Source `$x`	ALU Op. (`$y+offset`)	Write Mem. `M($y+offset)`

Conditional Branch: `beq $x,$y,offset`

Instr. Fetch & PC Increm.	Read of Source Regs. `$x` and `$y`	ALU Op. (`$x-$y`) & (`PC+4+offset`)	Write PC

Figure 1: MIPS instructions

The CPU, central process unit, amounts to a control unit plus a data path. Communication between the CPU and the memory in the computing infrastructure happens through the control, data and address buses.

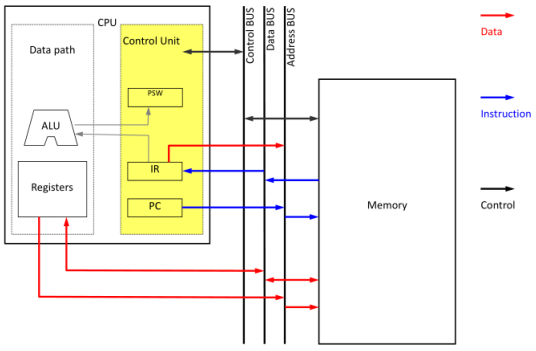

Figure 2: Computing infrastructure

Execution of MIPS instructions Every instruction in MIPS subset can be implemented in at most 5 clock cycles as follows:

IF Instruction Fetch Cycle

Send the content of Program Counter register to Instruction Memory and fetch the current instruction from Instruction Memory. Update the PC to the next sequential address by adding 4 to the PC (since each instruction is 4 bytes).

ID Instruction Decode and Register Read Cycle

Decode the current instruction (fixed-field decoding) and read from the Register File of one or two registers corresponding to the registers specified in the instruction fields.

Sign-extension of the offset field of the instruction in case it is needed.

EX Execution Cycle
The ALU operates on the operands prepared in the previous cycle depending on the instruction type:

Register-Register ALU Instructions: ALU executes the specified operation on the operands read from the RF

Register-Immediate ALU Instructions: ALU executes the specified operation on the first operand read from the RF and the sign-extended immediate operand

Memory Reference: ALU adds the base register and the offset to calculate the effective address.

Conditional branches: Compare the two registers read from RF and compute the possible branch target address by adding the sign-extended offset to the incremented PC.

ME Memory Access

Load instructions require a read access to the Data Memory using the effective address

Store instructions require a write access to the Data Memory using the effective address to write the data from the source register read from the RF

Conditional branches can update the content of the PC with the branch target address, if the conditional test yielded true.

WB Write-Back Cycle

Load instructions write the data read from memory in the destination register of the RF

ALU instructions write the ALU results into the destination register of the RF.

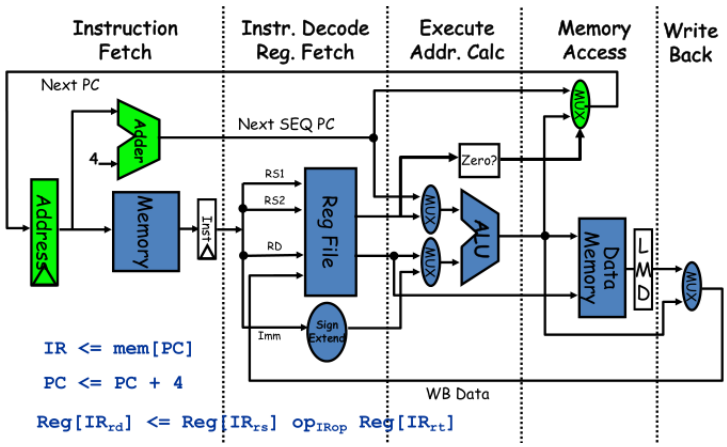

Figure 3: MIPS data path

In the single cycle implementation of MIPS the length of the clock cycle is defined by the critical path given by the load instruction: T = 8ns. Each instruction is executed in a single clock cycle of 8ns.

Instruction Type	Instruct. Mem.	Register Read	ALU Op.	Data Memory	Write Back	Total Latency
ALU Instr.	2	1	2	0	1	6 ns
Load	2	1	2	2	1	8 ns
Store	2	1	2	2	0	7 ns
Cond. Branch	2	1	2	0	0	5 ns
Jump	2	0	0	0	0	2 ns

Figure 4: Instruction latency

In the multi-cycle implementation the instruction is distributed on multiples cycle (5 for MIPS).

The basic cycle is smaller of 2 ns, the instruction latency is 10 ns.

The multi-cycle execution can be sequential of pipelined.

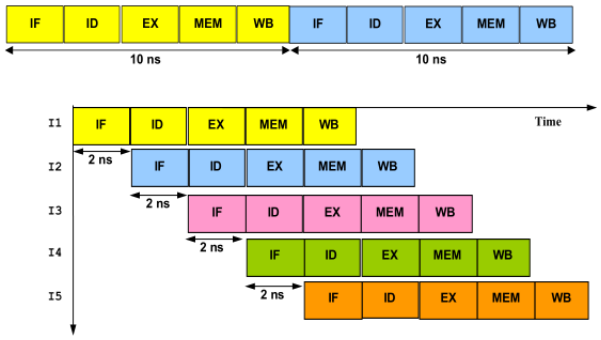

Figure 5: Sequential vs pipelined execution

Latency Each instruction is worsened from 8ns to 10ns.
Throughput Is improved by four times, 1 instruction each 8ns to 1 instruction each 2ns.

The Register File is used in WB and ID stages. In the **optimized pipeline** the RF write in the first half of the clock cycle and the RF read occurs in the second half of the clock cycle avoiding a possible read after write hazard which otherwise would've required a stall. It is safely to assume that the optimized pipeline will be the standard pipeline throughout the course.

2 Hazards

Dependencies are created at code-level and can be translated into conflicts when the compiler translate the code into instructions.

A hazard is created whenever there is a conflict (and therefore a dependency) between instructions, and these instructions are close enough that the overlap caused by pipelining would change the order of access to the operands involved in the dependence.

Hazards prevent the next instruction in the pipeline from executing during its designated clock cycle, reducing the performance from the ideal speedup gained by pipelining.

2.1 Structural hazards

Use of the same resource, such as the ALU, from different instructions simultaneously. There cannot be structural hazards in the MIPS architecture:

Instruction Memory separated from Data Memory, thus allowing the fetch of instructions and the reading of register during the same clock cycle.

Register File (RF) can be accessed twice in the same clock cycle: write access on the rising edge of the clock and read access on the falling edge by another instruction.

2.2 Data Hazard

Attempt to use a result before it is ready, three different types:

RAW, read after write

WAR, write after read

WAW, write after write

The RAW hazard is the only possible hazard in the MIPS architecture since all five stages are accessed in the same order in and between instructions, this is due to the fact that in the MIPS architecture there cannot be simultaneous access to the same resource and the access to the resources follow the order of the instructions.

For this reason write backs are in-order and therefore WAW cannot happen, and the reading of registers of a previous instruction happens always before the write back of a subsequent instruction precluding the WAR hazards.

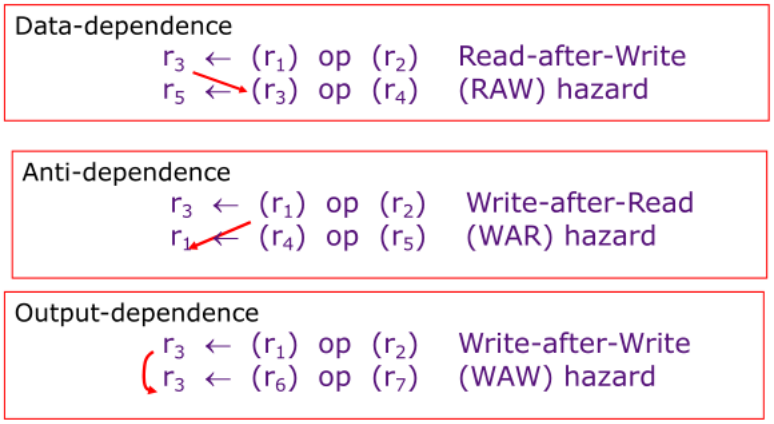

Figure 6: Data hazards

RAW hazards can be dealt with at compile-time (design solution) or at run-time (hardware solution).

Compilation Techniques

→ Nop instructions, insertion of no operation

→ Instruction Scheduling, to avoid that correlating instructions are too close the compiler tries to insert independent instructions among correlated instructions.

Scheduling is a reordering of independent instructions without changing the overall result; when the compiler does not find independent instructions, it inserts nops.

Hardware Techniques

→ Insertion of stalls (or bubbles) in the pipeline

→ Data Forwarding (or Bypassing)

Forwarding data means shortcutting data from one resource of the CPU as output to another resource of the CPU as input using one clock cycle for the propagation, this means using temporary result stored in the pipeline instead of waiting for the write back of results in the RF. Forwarding costs are additional multiplexers to allow to fetch inputs from the pipeline.

In the MIPS architecture the common paths for data forwarding are EX-EX, MEM-EX, MEM-MEM.

Note that WB-ID is not a data path, data is written and accessed in the same clock cycle inside the Register File. MEM-ID path is not useful if we divide clock cycle in rising and falling edge (optimized pipeline).

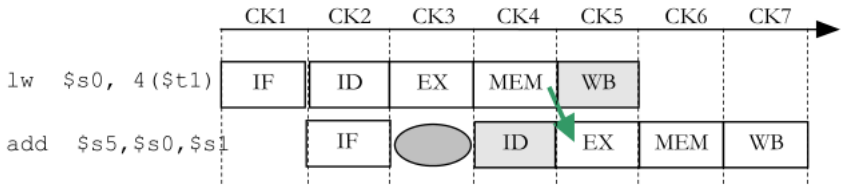

Figure 7: Load/Use hazard requires a stall to use the MEM-EX path

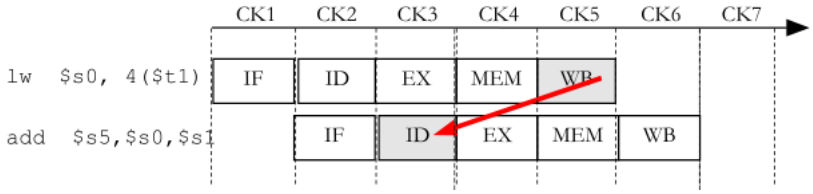

Figure 8: Load/Use without forwarding needs two stalls

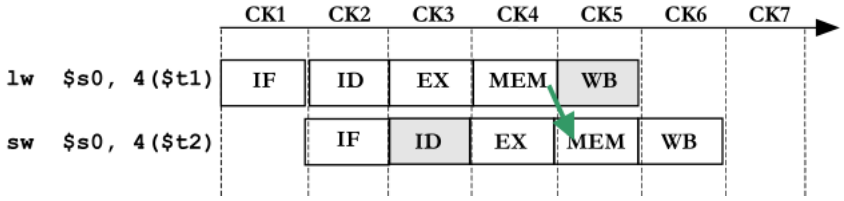

Figure 9: Load/Store hazard uses the MEM-MEM path

2.3 Control hazards

Attempt to make a decision on the next instruction to execute, before the condition is evaluated. A MUX is responsible for the value of PC, which is based on ALU output: until the ALU doesn't evaluate the condition the PC is not known, and we can't know which is the next instruction.

The PC is available at the MEM stage.

Generally true statements are executed in the waiting for the evaluation, as if the condition were true and no jump was needed: just add to the PC plus one, and then see if the branch is taken. For this particular reason it is suggested to write inside the `IF` statements the code that most probably is going to be executed.

In case we wait for the evaluation of the condition the stalls required are two in case of forwarding, three without forwarding.

An additional solution would be the early evaluation of the PC in the ID stage instead of the EXE stage, in this new pipeline the PC adder is anticipated and only one stall would be required (with forwarding) to fetch the correct instruction. This solution bring an issue in presence of a RAW hazard: anticipating the PC adder means that we have to add a stall in order to solve the conflict.

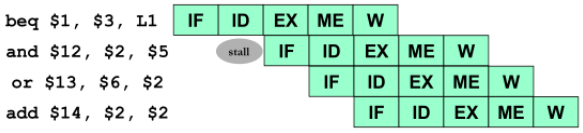

Figure 10: Early evaluation of the PC

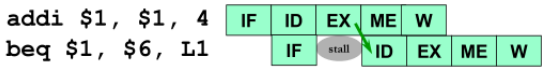

Figure 11: PC early evaluation issue

3 Branch prediction

When a branch is taken the **branch target address** (BTA) is stored in the PC instead of the address of the next instruction in the sequential instruction stream. The branch outcome and Branch Target Address are ready at the end of the EX stage, conditional branches are solved when PC is updated at the end of the ME stage. Control hazards reduce the performance from the ideal speedup gained by the pipelining since they can make it necessary to stall the pipeline.

$$\text{Pipeline Speedup} = \frac{\text{Pipeline Depth}}{1 + \text{Pipe Stall Cycles per Instruction due to Branches}}$$

$$= \frac{\text{Pipeline Depth}}{1 + \text{Branch Frequency} \times \text{Branch Penalty}}$$

Figure 12: Performance impact by control hazards

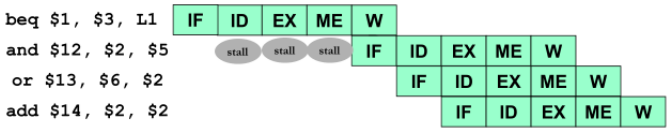

Figure 13: Branch stalls without forwarding

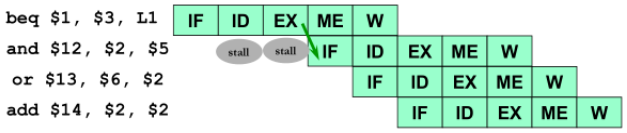

Figure 14: Branch stalls with forwarding

In order to reduce the speed penalty cause by the branch hazards we can use prediction techniques in order to guess the next instruction to be executed.

3.1 Static branch prediction

Static branch prediction are at compile-time:

- Branch always taken - No jump

- Branch always not taken
  In theory always jump, but in reality we have to calculate the BTA before jumping and at this point there is no prediction to make (in the MIPS pipeline).

- Backward taken, forward not taken
  Based on the assumption that backward jumps (while and for loops) are taken and forward jumps (if statements) are not taken.

- Profile driven prediction
  We are going make several runs of our program and the choice is based on statistical data for each branch.

- Delayed Branch
  The compiler statically schedules and independent instruction in the branch delay slot, which is executed whether the branch is taken. There are three ways in which the branch delay slot can be scheduled:

  From before, the instruction in the delay slot is always executed.

  From target, this strategy is preferred when the branch is taken with high probability.

  From fall-through, this strategy is preferred when the branch is not taken with high probability. In this case the branch delay slot is scheduled from the not-taken fall-through path.

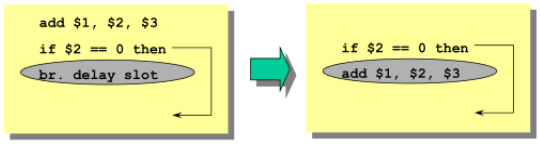

Figure 15: Branch delayed - from before

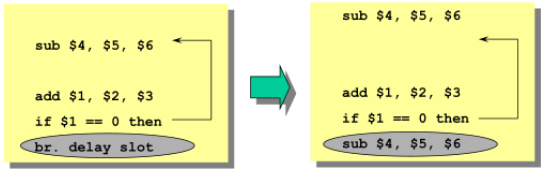

Figure 16: Branch delayed - from target

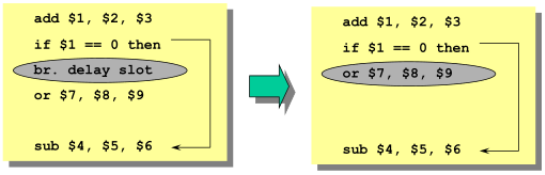

Figure 17: Branch delayed - from fall-through

3.2 Dynamic branch prediction

Dynamic branch prediction happens at run-time by the hardware, and is based on two interacting mechaninsms:

- Branch Outcome Predictor
  To predict the direction of a branch (i.e., taken or not taken).

  Branch History Table

- Branch Target Predictor
  To predict the branch target address in case of taken branch.

  Branch Target Buffer

3.2.1 Branch Target Buffer

The Branch Target Buffer (BTB or Branch Target Predictor) is a cache storing the predicted branch target address for the next instruction after a branch. We access the BTB in the IF stage using the instruction address of the fetched instruction (a possible branch) to index the cache.

Figure 18: BTB entry

3.2.2 Branch History Table

The Branch History Table (BHT or Branch Prediction Buffer) is indexed by the last k-bits of the branch address, meaning that there are at most 2^k entries, with possible collisions, and each entry contains n-bits that says whether the branch was recently taken.

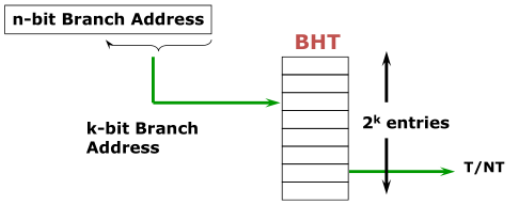

Figure 19: Branch history table

The 1-bit history table tells us if the last time the branch was taken or not taken (e.g., 0 and 1). Differently from the profile drive prediction the BHT is updated whenever the branch is encountered.

For example:
0, prediction not taken, outcome not taken - no update
0, prediction not taken, outcome taken - update 0 to 1
0, prediction taken, outcome not taken - update 1 to 0

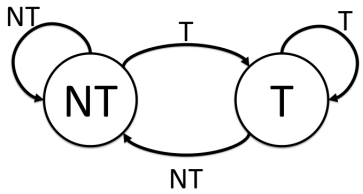

Figure 20: 1-bit BHT automata

A misprediction occurs 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. To solve this problem it is enough to increase the number of rows in the BHT or to use a hashing function.

Among the n-bits BHT the 2-bit history table is the best one because it is still small in size, it allows for speedy updates maintaining the prediction accuracy high, but differently from the 1-bit the prediction must miss twice before it is changed. In a loop branch we do not need to change the prediction at the last iteration, thus increasing the speed in case of nested loop such as the ones used to inspect matrices.

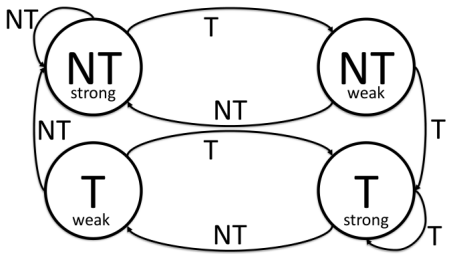

Figure 21: 2-bit BHT automata

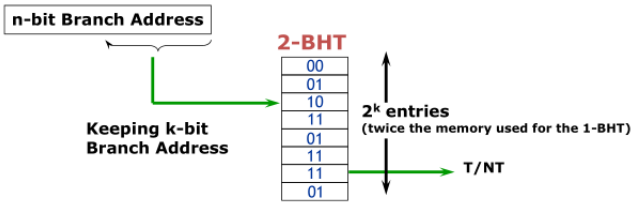

Figure 22: 2-bit BHT

3.2.3 Correlating Branch Predictor

The idea behind is that the behaviour of recent branches are correlated, that is the recent behaviour of other branches can influence the prediction of the current branch. In general (m, n) correlating predictor records last m branches to choose from 2^m BHTs, each of which is a 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).

Example: a (2, 2) correlating predictor with 64 total entries; 6-bit index composed of: 2-bit global history and 4-bit low- order branch address bits.

The (2,2) correlating branch predictor outperforms other predictors such as the 2-bit BHT.

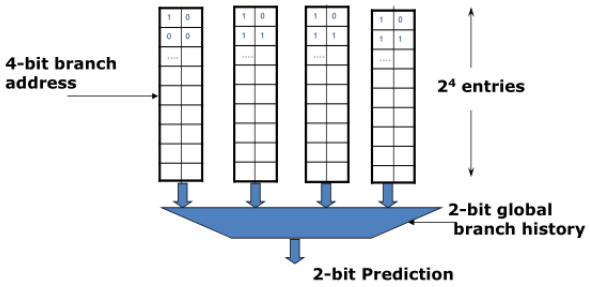

Figure 23: (2,2) correlating branch predictor

4 Instruction level parallelism

ILP: potential overlap of execution among unrelated instructions, possible if:

- no structural hazards
- no raw, waw, war stalls
- no control hazards

$$\text{Pipeline CPI} = \text{Ideal pipeline CPI} + \text{Structural Stalls} + \text{Data Hazards Stalls} + \text{Control Stalls}$$

In the MIPS architecture the only possible structural hazards, WB-ID, where a register in the RF can be both written and read, is solved by splitting the clock cycle in rising edge (WB) and falling edge (ID).

ILP	**PP**
overlap individual machine operations	separate processor getting separate chuncks of the program
transparent to the user	nontransparent to the user
goal: speed up	goal: speed up and quality up

Table 1: ILP vs Parallel Processing

4.1 Complex Pipeline

WAR and WAW were not possible with ADD operation with integers, but MULT and DIV operations use floating point numbers. A new stage introduced, the ISSUE stage, that allows for high performance in presence of:

- long latency or partially pipeline floating-point units
- multiple function and memory units
- memory systems with variable access time
- precise exception

New hazards arise from variable latencies of different FUs:

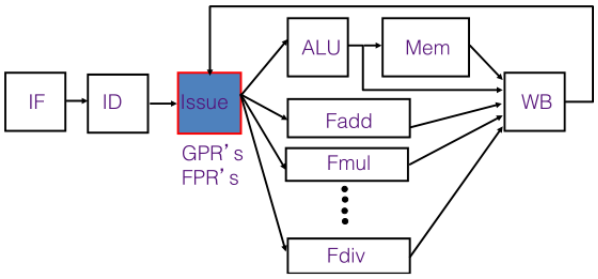

Figure 24: Complex pipeline

- Structural
  - EX stage if some FPU or memory unit is not pipelined and takes more than one cycle.
  - WB stage due to variable latencies of different functional units
- Data
  - WAW due to variable latencies of the FUs

A solution would be delay the WB in order to have the same latency for all instructions but that would slow down too much single cycle integer operations without forwarding.

When it is safe to issue an Instruction? Things to consider before dispatching an instruction:

- the required FU is available
- the input data is available
- it is safe to write the destination
- there is not a structural conflict at the WB stage

Assumptions Of a general complex pipeline architecture:

- all functional units are pipelined
- registers are read in ISSUE stage: we consider that a register is written (WB) in the first half of the clock cycle and read (IS) in the second half of the clock cycle

- no forwarding
- ALU operations take 1 clock cycle
- FP ALU operations take 2 clock cycles
- memory operations take 2 clock cycles
- write back unit has a single port
- instructions are fetched, decoded and issued in order
- an instruction will only enter the ISSUE stage if it does not cause a WAR or WAW hazard
- only one instruction can be issued at a time, and in the case multiple instructions are ready, the oldest one will go first

To reach higher performance more parallelism must be achieved, this cannot be done augmenting the CPI of the ideal pipeline because it could create further problems with hazards.

Dependences must be detected and solved, and instructions must be ordered (scheduled) so as to achieve highest parallelism of execution compatible with available resources.

Three types of dependencies:

- data dependencies, RAW
- control dependencies
- name dependencies, two types:
  - antidependencies, WAR
  - output dependencies, WAW

  Generated by the lack of registers.

Dependencies are a property of the program, while hazards are a property of the pipeline.

The main techniques to eliminate dependencies are:

- register renaming
- scheduling
  - static, compiler

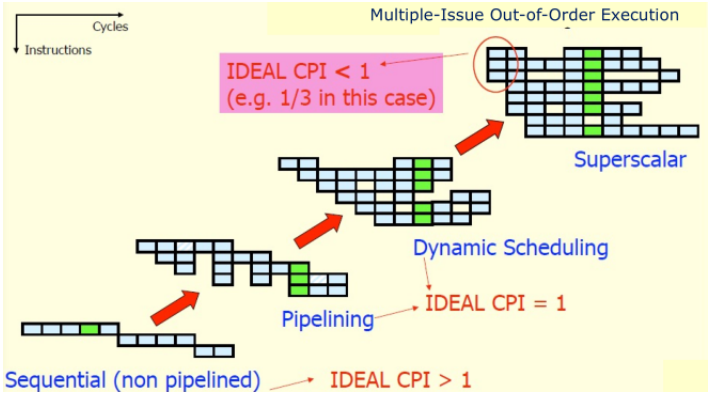

Figure 25: Instruction Level Parallelism

– dynamic, hardware

Steps to exploit more ILP in terms of hardware optimization:

→ sequential

↪ pipeline
single-issue in-order-execution

↪ dynamic scheduling
single-issue out-of-order execution

↪ superscalar
multiple-issue out-of-order execution

5 Static scheduling

Compilers can use sophisticated algorithms for code scheduling to exploit ILP. The amount of parallelism available within a **basic block**, a straight-line code sequence with no branches in except to the entry and no branches out except at the exit, is quite small (statistically it varies from 4 to 7 instructions).

Data dependence can further limit the amount of ILP we can exploit within a basic block to much less than the average basic block size.

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

Static detection and resolution of dependencies are accomplished by the compiler, dependencies are avoided by code reordering.

The output of the compiler is a reordered dependency-free code, which can be processed, for example, by the VLIW (Very Long Instruction Word) processors.

Limits of Static Scheduling

- unpredictable branches
- variable memory latency
- code size explosion
- compiler complexity

Till know we had a single-core multi-cycle parallelism, thanks to the single-issue pipelined architecture. The single-issue architecture means we cannot aim to have an ideal CPI bigger than one.

Multi-issue architectures are the next step to improve the CPI:

- Superscalar
- VLIW

5.1 VLIW architectures

Each instruction word contains more than one operation, the processor can initiate multiple operations per cycle specified completely by the compiler. This leads to a low hardware complexity:

- no scheduling

- reduced support of variable latency instructions
- single control flow (1 PC)
- explicit parallelism

An instruction can be a set of operations that are intended to be issued simultaneously, the VLIW has multiple operations packed into one instruction, and each type of operation has its own slot.

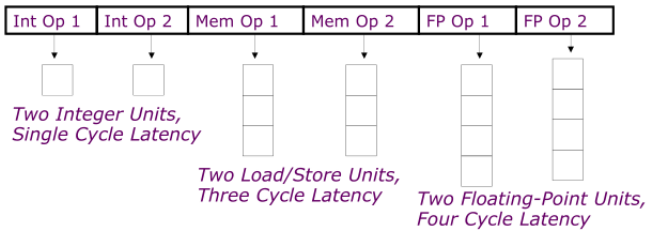

Figure 26: Very Long Instruction Word

This type of architecture requires guarantees of:

- parallelism within an instruction does not generate RAW data hazard
- no data use before the data is ready, no data interlocks

Compiler responsibilities:

- maximize parallel execution
  - exploit ILP and LLP (Loop Level Parallelism)
  - it is necessary to map the instructions over the machine functional units
  - this mapping must account for time constraints and dependencies among the tasks
- guarantee intra instruction parallelism
- avoid hazards (no interlocks), typically separates operations with explicit nops
- minimize the total execution time

Pros:

simple HW

good compilers can effectively detect parallelism

Cons:

- huge number of registers to keep active the FUs
  - needed to store operands and results
- large data transport capacity between
  - FUs and register files
  - Register files and Memory
- high bandwidth between i-cache and fetch unit
- large code size
  - use of (big) nops in the VLIW
  - unpredictable branches have different optimal schedules that varies with branch path
- knowing branch probabilities requires significant extra steps in build process due to profiling

An example of scheduling of a loop in VLIM architecture:

operations	**ls, sd**	**add, bne**	**fadd**
clock cycles	3	1	4

Many blank lines in the scheduler corresponds to nops, leading to $\frac{1 \text{ FP ops}}{8 \text{ cycles}} = 0,125$, which is far from our ideal CPI > 1. To increase parallelism we use loop unrolling.

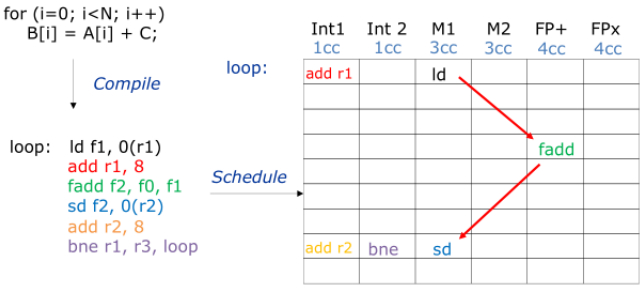

Figure 27: Loop execution on VLIM

5.2 Loop unrolling

The idea is that we can extend the loop body as to include a finite number of subsequent iterations of the loop, increasing the amount of available ILP. Unrolling simply replicates the loop body multiple times, adjusting the loop termination code, by doing so the loop *overhead* decrease relatively to the loop *body*.

$$\text{Loop} = (\text{loop prolog} + 4 \times \textit{loop body} + \text{loop epilog}) \times \tfrac{n}{4} \text{ iterations}$$

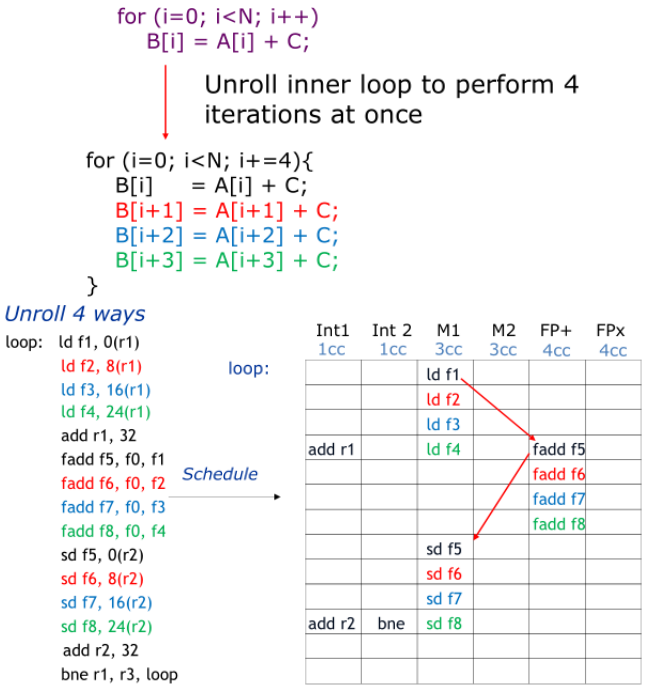

Figure 28: Loop unrolling

Note that non FP operations can have multiple schedule locations, e.g., the `ld f2` could be scheduled also in the first cycle to M2, but if does not decrease the CPI, it is recommended using a new instruction. In this case we end up with $\frac{4 \text{ FP ops}}{11 \text{ cycles}} \approx 0,36$, more than doubling the previous result.

The **limits** of loop unrolling:

code size

number of register

That is, we have an upper bound to the number of replications of the loop body, which has to be considered constant in regard to the $\theta(n)$ loop iterations: we reduced the *prolog* and *epilog* of the loop by a constant.

We can optimize the results of loop unrolling with **software pipelining**, further increasing the ILP.

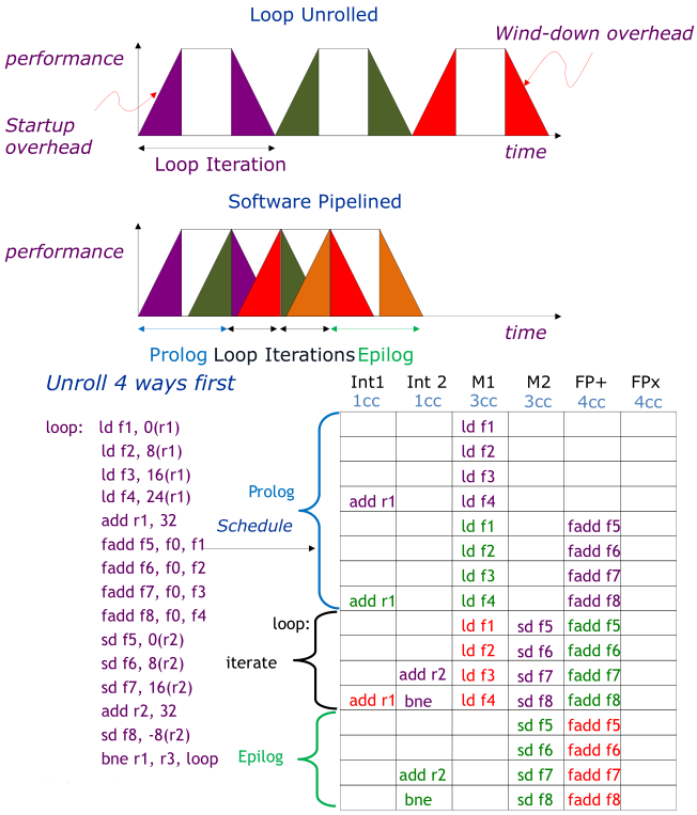

Figure 29: Loop unrolled vs Software pipelined

$$\text{Software Pipelined Loop} = \text{prolog} + \textit{iterations} + \text{epilog}$$

In this case we have just one prolog and one epilog for all the loop iterations and can be considered $O(1)$ with respect to the n iterations. For this reason we consider only the *iterate* part for our performance evaluation which is $\frac{4 \text{ FP ops}}{4 \text{ cycles}} = 1$.

Note: in case of short loops, loop unrolling looses performance due to the costs of starting and closing the iterations.

5.3 Trace Scheduler

Loop unrolling is useful but does not cover other type of branches that limit the basic **block size** in control-flow intensive irregular code. In these cases is difficult to find ILP in individual basic blocks.

Trace scheduling focus on traces:

- a loop-free sequence of basic blocks embedded in the control flow graph
- it is an execution path which can be taken for some set of inputs
- the chances that a trace is actually executed depends on the input set that allows its execution

Note: a trace may include branches but not loops.

Algorithm Idea, some traces are executed much more frequently than others:

- pick string of basic blocks, a trace, that represents most frequent branch path
- use profiling feedback or compiler heuristics to find common branch paths
- schedule whole "trace" at one
- add fixup code to cope with branches jumping out of trace

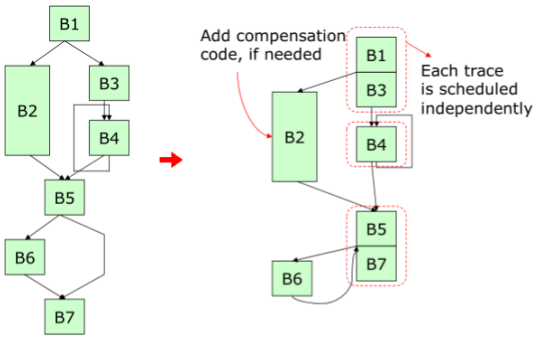

Figure 30: Trace scheduling

For example we suppose that {B1, B3, B4, B5, B7} is the most frequently executed path, therefore the traces are {B1, B3}, {B4} and {B5, B7}.

Note: Traces are scheduled as if they were basic blocks, by removing control hazards we increase ILP.

Problems Compensation codes are difficult to generate, especially entry points (of a different path). In addition to need of compensation codes there are restrictions on movement of a code in a trace:

- **dataflow** of the program must not change, it is guaranteed to be correct by maintaining:
  - data dependencies
  - control dependencies
- the exception behaviour must be preserved

Solutions There are two approaches to eliminate control dependency:

- use of **predicate** instructions (Hyperblock scheduling)
- use of **speculative** instructions (Speculative Scheduling), and speculatively move an instruction before the branch.

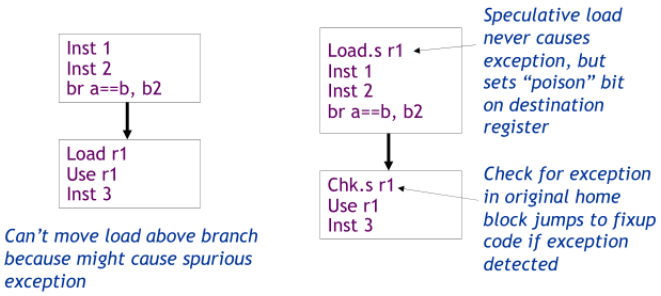

Figure 31: Speculative instruction

Rotating Register Files

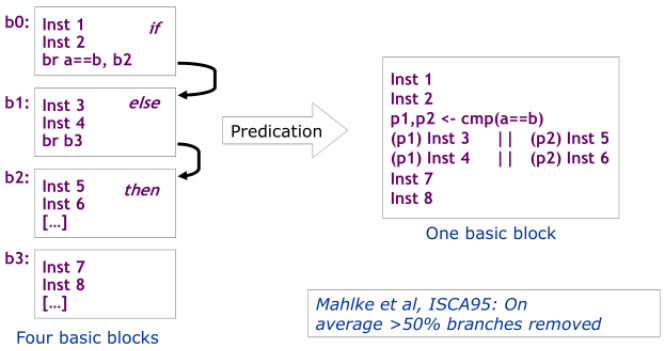

Figure 32: Predicated execution

5.4 Dynamic scheduling

The hardware reorders the instruction execution to reduce pipeline stalls while maintaining data flow and exception behavior. • Main advantages: – It enables handling some cases where dependences are unknown at compile time – It simplifies the compiler complexity – It allows compiled code to run efficiently on a different pipeline. • Those advantages are gained at a cost: – A significant increase in hardware complexity, – Increased power consumption – Could generate imprecise exception

Basically: Instructions are fetched and issued in program order (in- order-issue) • Execution begins as soon as operands are available – possibly, out of order execution – note: possible even with pipelined scalar architectures. • Out-of order execution introduces possibility of WAR, WAW data hazards. • Out-of order execution implies out of order completion unless there is a re-order buffer to get in-order completion

Contents