Coupling Latency-Insensitivity with Variable-Latency for Better Than Worst Case Design: A RISC Case Study

Mario R. Casu, Stefano Colazzo, Paolo Mantovani
Dipartimento di Elettronica, Politecnico di Torino
C.so Duca degli Abruzzi, 24, I-10129, Torino, Italy
mario.casu@polito.it

ABSTRACT

The gap between worst and typical case delays is bound to increase in nanometer scale technologies due to the spread in process manufacturing parameters. To still profit from scaling, designs should tolerate worst case delays seamlessly and with a minimum performance degradation with respect to the typical case. We present a simple RISC core which tolerates worst case extra latency using the *Latency-Insensitive Design* approach coupled to a *Variable-Latency* mechanism. Stalls caused by excessive delay, by data and control hazards and by late memory access are dealt with in a uniform way. Compared to a pure worst-case approach, our design method permits to increase the core clock frequency by 23% in a 45 nm CMOS technology, without area and power penalty.

Categories and Subject Descriptors

B.5.1 [**Design**]: REGISTER-TRASNFER-LEVEL IMPLEMENTATION—*Control design, Data-path design*

General Terms

Design

Keywords

Latency-Insensitive Design, Variable-Latency

1. INTRODUCTION

The high variability of process technology parameters – like threshold voltages and transistor channel lengths – is a design challenge in nanometer technologies. Designers who take large margins to ensure operation in the so-called *worst case* process corner – worst combination of technological parameters producing the longest delays – are bound to waste the advantages of scaling. On the other hand, a *typical case* approach likely does not meet the design yield requirements. To solve the dilemma, researchers in the field of microarchitecture design are investigating in two directions which have in common the aim to design for the typical instead of the worst case. A first approach, whose foremost representative is Razor [1][2], tolerates variability through error detection and correction. The other technique uses *Variable-Latency* (VL) functional units that operate mostly at one clock cycle per operation, and schedule execution in two cycles when critical paths are activated [3].

This work belongs to the second class of methods. We designed a processor core compatible with the MIPS R2000 instruction set (albeit with no support for floating point) whose control is designed to tolerate variable-latency operations. We developed the core at register-transfer level and coded it in synthesizable VHDL. The pipeline interlock is based on the *Latency-Insensitive Design* (LID) approach which uses a synchronous handshake among the pipeline stages [4] and makes it simple and convenient to embed VL functional units. The main contributions of this work are:

- A fine-grain stage-by-stage pipeline interlock, enabled by the LID approach, allows to support in a uniform way pipeline stalls caused by control and data hazards, late memory access and stalls caused by VL execution.
- A specific synthesis method increases the clock frequency by 23% compared to a standard approach and limits VL penalties just to the worst process corner. No performance degradation ensues in all other corners.
- A small software built-in self-test (BIST) executed once after reset and that makes use of standard instructions checks if the core can run at full speed or if it needs two cycles when a critical instruction is executed.

Other relevant works reported MIPS implementations, like the one by Martin *et al.* who presented an asynchronous delay-insensitive MIPS R3000 in [5]. Delay-insensitive design differs from LID, despite the similarity in nomenclature, under various aspects, but primarily because of the more radical asynchronous style of the first compared to the usual synchronous choice of the second which makes it more suited to a standard design flow. As far as we know this is the first reported implementation of a synchronous latency-insensitive processor with variable-latency functional units compliant with a well-known instruction set architecture.

The paper is so organized. We provide a short background about related work in Section 2. LID and VL units are the subject of Section 3 and 4, respectively. Logic synthesis methodology and results on a CMOS 45 nm technology target are discussed in Section 5. Section 6 describes the simulations used to verify design correctness and to evaluate performance. Section 7 concludes the paper.

GLSVLSI'11, May 2–4, 2011, Lausanne, Switzerland.

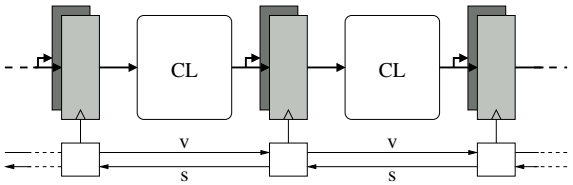

Figure 1: Latency-insensitive pipeline with local stall. (CL stands for combinational logic.)

2. BACKGROUND AND RELATED WORK

Latency-Insensitive (LI) design was introduced by Carloni *et al.* to help synchronous circuits tolerate excessive latency coming from wires [4]. Synchronous processes exchange data by means of a handshake protocol that utilizes validity and stall bits. If some of the inputs to a block are not valid, because data are late due to an excessive wire delay, the block gets clock-gated. It will be enabled back when the data will finally arrive. At the same time the other valid inputs are stalled. Stall signals are not broadcast by the protocol but are instead locally back-propagated, block by block. As a result, extra buffers are needed to save incoming new data that otherwise will be lost.

This principle can be adapted to a standard pipelined microarchitecture. Every pipeline register stores only valid data and is clock-gated if receives invalid tokens or a stall signal. In the latter case a new incoming datum is stored in an ancillary register. The idea of such fine-grain pipeline interlock was discussed first in [6] but was not further extended to a real design case. That paper also proposed a smart implementation of the double buffer by allowing a separate use of the two latches of a master-slave pair.

Figure 1 depicts a hypothetical LI pipeline of combinational logic (CL) stages separated by pipeline registers and controlled by a stall logic. Stalling events arising locally are progressively propagated upstream (s=1) and progressively invalidate (v=0) downstream stages, resulting in a local clock gating. Two registers per stage are needed to save an incoming valid datum while the current one is stopped.

Stalling events occur because of data and control dependencies or because of excessive latency in computation or in memory access. The different sources of stalls are handled in the same way by the LI control using forward valid (v) and backward stop signals (s). Cortadella *et al.* in [7] briefly hint at how a LI protocol (called "elastic" in [7]) can be coupled with VL units, but did not elaborate further. In essence, a late unit invalidates the forward validity bit and stops the new incoming data at the same time. When computation is done, stall gets released and validity asserted again.

Literature cited so far focused on the control substrate that can be used to build an efficient latency-tolerant pipeline. Other works concentrated on the variable-latency datapath. Relevant examples are [3] for a methodology that isolates critical paths and predicts their activation, [8] for variable-latency register files and floating-point units, [9] for a technique that pairs random logic with extra gates to evaluate whether computation requires one or two cycles to complete.

A number of papers addressed how to tolerate pipeline errors. The Razor approach, proposed to detect errors caused by low voltage in a dynamic voltage scaling setting [1], proved useful also to tolerate process variations [2]. In case of error, [1] proposes a global stall, something that we avoid by means of the LI protocol. [2] avoids stall broadcast but the error gets propagated downward and taken care of just prior to register file commit. Finally a rollback re-executes the failing instruction which may have issued long before error handling, with consequent waste of time and energy.

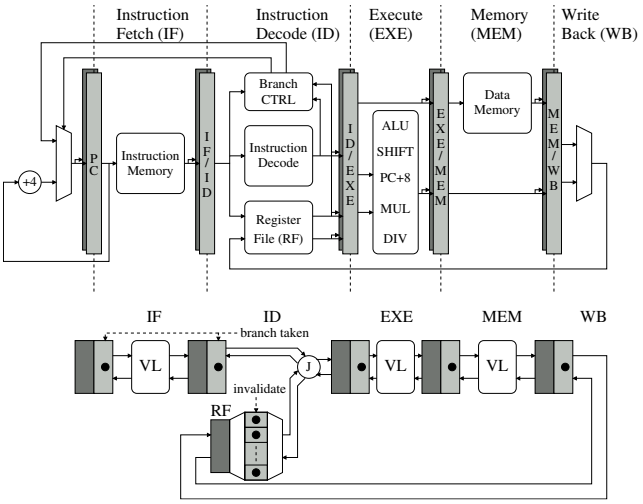

Figure 2: MIPS latency-insensitive pipeline and abstract representation of the control with token flow.

In principle, error detection could be coupled with LI control, but we preferred to prevent errors with a VL mechanism rather than correcting them. The reason is the complexity of the sequential circuit that substitutes standard pipeline registers for error detection and of the corresponding timing constraints [2]. Moreover, it is not well suited to soft, synthesizable cores that use standard-cell libraries, like ours.

3. MICROARCHITECTURAL DESIGN

The pipeline of our MIPS is shown on top of Figure 2. It is a classic in-order five stage design (IF, ID, EXE, MEM and WB) with no bypass loops. We kept it extremely simple compared to a modern microarchitecture so as to focus on latency-insensitive and variable-latency features rather than on *instructions per cycle* (IPC) performance optimization.

Every pipeline register is paired with its companion buffer that saves the incoming datum if a stall signal arises from the subsequent stage. For reasons of clarity the diagram omits control and protocol signals. Branch and jump instructions are resolved in the decode stage. The execution unit is made of five parallel units, an arithmetic and logic unit (ALU), an adder that increments the program counter (PC+8) to compute the link address, a shifter, an internally pipelined multiplier to support integer 32 and 64-bits multiplications, and a multicycle combinational divider that completes operation in a fixed nine clock cycle latency.

The bottom of Figure 2 illustrates an abstract model of the pipeline LI control. The rectangles represent the pipeline register pairs. At reset, the light gray ones contain valid data, represented by black tokens in figure. The dark gray elements, initialized with "bubbles" (i.e. no tokens), store valid tokens when stops occur. Tokens propagate along the "valid" paths (forward arrows) whereas stall events assert "stop" signals (backward arrows). The J element is a *join*

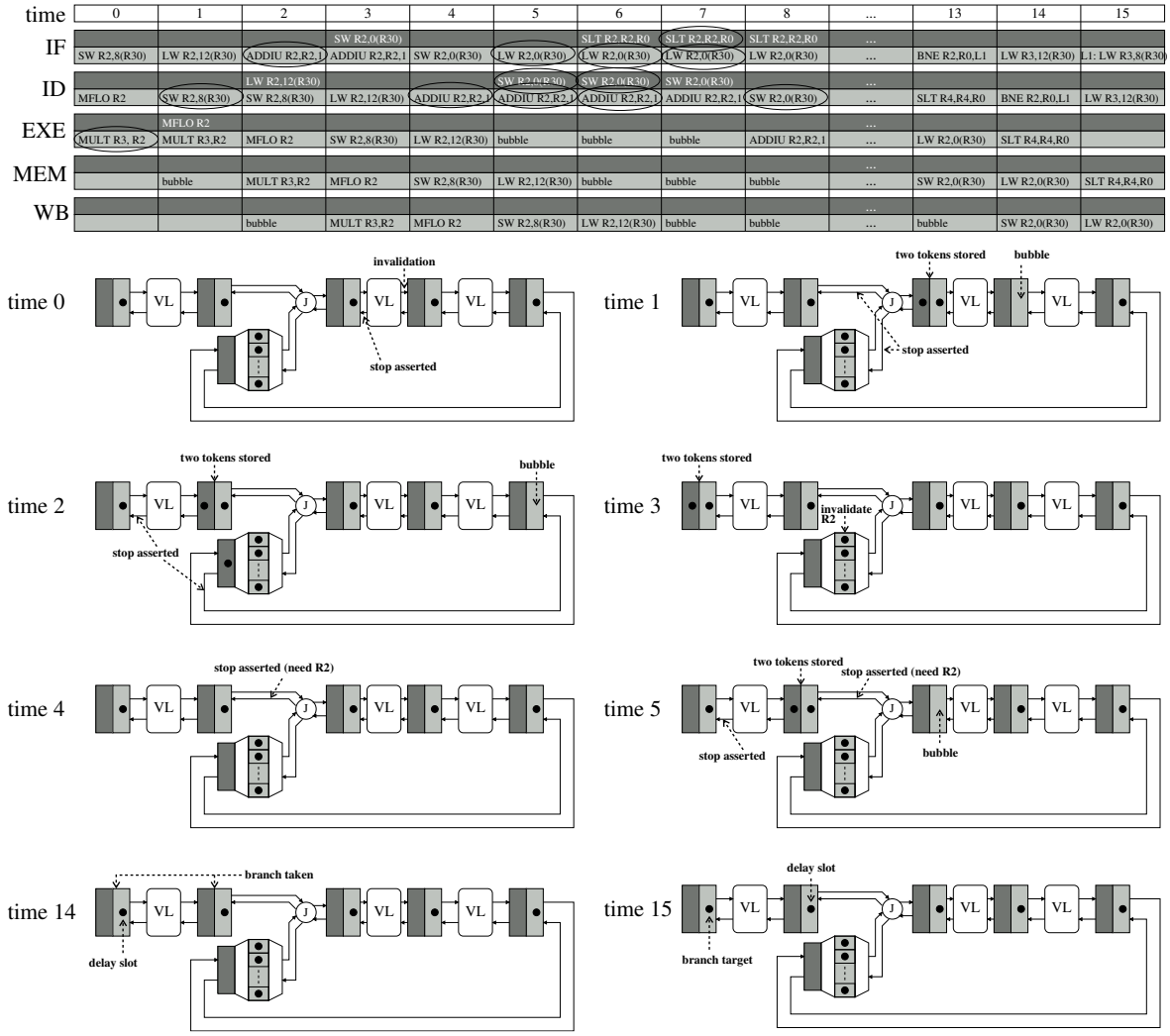

Figure 3: Execution of assembly segment and corresponding token view. Circled instructions are stalled.

controller that validates its output when both its inputs are valid and stalls the valid one if the other is not valid [7]. In this case, both the decoded instruction and the required registers must be valid. Each of the 32 registers of the register file (RF) has a corresponding token. A load instruction, or in general any instruction that updates a register, first *invalidates* the RF destination token. When such instruction reaches the WB phase, it validates again the token of the destination register. This automatically solves any *load-use* data dependency: Any other valid instruction attempting to read that register before its token gets validated generates a pipeline stall through the join controller. We remark that such pipeline interlock solves all data dependencies without the need for compiler intervention.

The dashed arrows labeled *branch taken* in Figure 2 indicate the actions subsequent to the decoding of a branch instruction. The program counter is updated with the branch target, if taken. The token of the IF/ID pipeline register contains the so-called *delay slot*, that is the instruction that immediately follows a branch or jump.

Figure 3 shows some situations of stalling events to help understand the behavior of the LI control of the pipeline. The top part reports a sequence of instructions that flow throughout the various pipeline stages in clock cycles 0-15. Circled instructions are stalled. The bottom part is the corresponding token view for cycles 0-5 and 14-15. Three relevant situations are reported. A case of stall caused by excessive computation latency (stall event at time 0); a case of stall created by load-use data dependency (event at time 4 and also at time 8); a situation of branch taken (time 14).

At time 0, *multiply* instruction `MULT R3,R2` in EXE stage gets stalled because of an excessive latency. A stop event at time 0 stalls the instruction and invalidates the result. This corresponds to the bubble inserted in MEM pipeline stage at time 1. The incoming *move* instruction `MFLO R2`, decoded at time 0, gets queued in the secondary ID/EXE register (dark gray) at time 1 because the primary (light gray) is full (two tokens stored in token view at time 1). Also, the stop signal flows backward and at time 1 stalls the just fetched *store* instruction `SW R2,8(30)` that is ready to be decoded. From time 1 to 2 the stop signal flew backward and now stalls *addition* instruction `ADDIU R2,R2,1` in fetch stage while two tokens are now stored in IF/ID register pair

and another token is stored in the ancillary dark register of the RF. At time 2 a stop goes also backward from RF to MEM and meets a bubble. Stopping an invalid datum is useless, hence the stop does not further propagate.

When `ADDIU R2,R2,1` finally reaches ID stage at time 4, a data dependency occurs. Previously decoded instruction `LW R2,12(R30)`, now in EXE, invalidated at time 3 the token of destination register R2: It will be only updated when the store commits. Hence, the addition wanting to read R2 is stalled and restarts only at time 7 when R2 is available again. Thereby two instructions get stored in IF/ID pair at time 5 and two in PC at time 6. At time 8 an analogous case of data dependency occurs to instruction `SW R2,0(R30)`.

At time 14, conditional *branch if not equal* instruction `BNE R2,R0,L1` is decoded. If R2 register content differs from R0, the branch target labeled `L1`, which corresponds to *load* instruction `LW R3,8(R30)`, is fetched. The delay slot instruction, `LW R3,12(R30)`, is executed even if the branch is taken.

4. VARIABLE-LATENCY UNITS

To take advantage of typical silicon and get more performance than a standard worst-case approach, it's necessary that in the worst scenario only a few, predictable critical paths exist to which the VL mechanism can be applied. If they are rarely activated, and so the extra clock cycle is not frequently used, the performance penalty will be limited. For simple RISC processors like ours critical paths are in EXE, due to long carry propagation in adders or multipliers, and in the branch-loop path from DEC to PC.

As for the EXE paths, regular average-case input patterns rarely activate them [10][11]. For the multiplier, since 64-bits multiplications are not frequent, we allow the higher 32-bits part of the result to be computed in three clock cycles instead of two (the multiplier has an internal pipeline register). MIPS multiplications write their result into a special register from which data can be read with a special move instruction. Then, once the VL is configured, `MULT` or `MULTU` instructions are detected by the EXE dispatcher which stalls the pipeline locally by asserting the backward stop signal and injecting a bubble forward, but only if a move from special register immediately follows. If not, no stalls occur. Another stall case arises in the very uncommon case in which a 32-bits `MUL` follows a 64-bit multiplication. 32-bits `MUL`'s are instead not subject to extra latency.

For the 32-bits adder/subtracter in ALU and the PC+8 adder, we used a different technique. We cannot schedule an extra clock cycle latency based on the instruction only, as we did for the multiplier, otherwise the penalty would be intolerable, given the high rate of occurrence of these instructions. We thus designed a Brent-Kung parallel-prefix adder [12] with a few extra gates that detect the critical paths activation. In a nutshell, we enable variable-latency at present clock cycle if carry bit C_{15} changed compared to the previous cycle (an xor gate and a flip-flop are used for this purpose) and if it will be propagated at least until the 23rd sum bit (the logic AND of propagate bits P_{22} down to P_{16} is the condition we use). This condition covers the true critical path (which goes down to sum bit 31st) and some others less critical but close to it. We compared our adder with the one available through the Synopsys DesignWare library and the worst propagation delay turned out to be only 20 ps slower in a 45 nm CMOS technology, an acceptable penalty.

For the optimization of the branch-loop, we selected the register file (RF) as the component of this path to which VL could be efficiently applied. By allowing half of the registers in RF to be accessed in two instead of one cycle in the worst case we can actually shorten the access time to the other half, because this type of relaxed constraint helps the logic synthesizer find a shorter delay. Then, when a branch instruction attempts to read registers from R16 to R31, an extra stall is generated and the branch completes in two clock cycles. The choice of slowing down registers R16:R31 comes from an analysis of some program benchmarks. We found out that such registers are seldom used for branch instructions, much less than registers R0:R15.

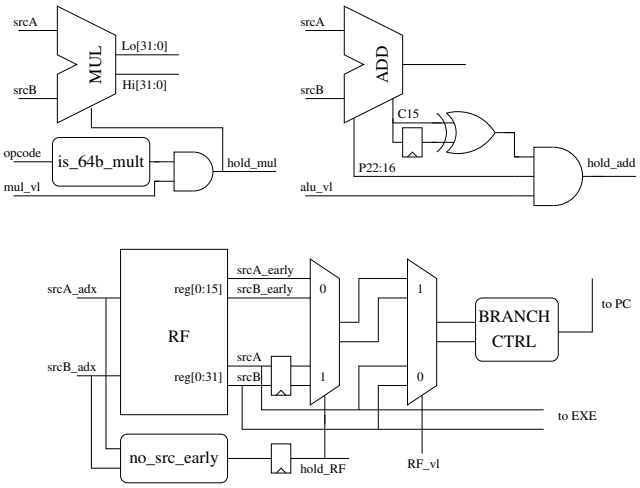

Figure 4: Hold functions for extra latency operation.

Figure 4 reports a schematic representation of the "hold functions" that allow functional units to operate with an extra clock cycle if required. For the 64-bits multiplications, the condition is simply the logic AND of a decoded opcode and signal `mul_vl`. For the instructions that use the ALU the activation of the critical path is data-dependent. Again, the `alu_vl` signal at logic one is required. Although not shown in figure, the PC+8 adder and the corresponding VL enable signal `pcp8_vl` and hold function are identical to the ALU ones. As for the RF, if signal `RF_vl` is active, the branch controller is fed with "early" signals if registers R0:R15 have been addressed or with one clock cycle late signals (due to the extra register) if registers R16:R31 have been chosen.

Signals `mul_vl`, `alu_vl`, `pcp8_vl` and `RF_vl` – in the following indicated as `*_vl` – are set to logic one in the worst case only, that is if the corresponding critical path requires an extra cycle. When at logic zero, no extra cycle is required. Next subsection discusses how these signals are set.

4.1 Software BIST

To decide whether VL is necessary or not, a test of the delay of the processor critical paths is needed. If such paths end in architected registers, it is not necessary to add logic for testing because the result of delay critical operations are fully exposed to the software. A software built-in self-test (BIST) routine can be then written for this purpose. The only caveat is that checking operations must be reliable and always produce a correct result, no matter the process corner. This can be obtained by properly setting the `*_vl` signals prior to executing a checking instruction.

We thus developed a short BIST routine that is run after reset. It makes use of standard MIPS instructions and can be compiled with a standard tool chain. It interacts with the hardware by setting a configuration register made of 4 bits, the *VL Mask Register* (VLMR). Each of the VLMR bits is connected to one of the `*_vl` signals. We used the fact that in the standard MIPS architecture the coprocessor 0 (COP0), which handles system functions, contain registers numbered from 0 to 15, but the register opcode field in instructions `MFC0` and `MTC0`, which move data between COP0 and RF, is 5-bits wide. As a result, we can address up to 32 registers in COP0 and so we mapped the VLMR as the 16th register, in a way totally transparent to the compiler.

The BIST routine executed at startup checks in sequence the various units and sets the corresponding mask bits with a move to COP0, that is a `MTC0` instruction, using the mask value as the first argument and register 16 as the second one. The code, 70 lines of MIPS assembly, consists of four subroutines that activate the critical paths of every VL unit:
PC+8: To test this adder we preload at reset a fake *jump and link* (`JAL`) instruction in the IF/ID register with a PC value that stimulates the critical path. VL is also set to zero at reset (`pcp8_vl` = 0). The `jal` instruction updates register R31 in RF and the rest of the BIST routine checks its value to verify its correctness. Finally signal `pcp8_vl` gets updated through a proper `MTC0` instruction, if necessary.
RF and conditional branches: The variable-latency is removed (i.e. `RF_vl` = 0) and two *slow* registers – that is in range R16:R31 – are first reset, then preloaded with all ones and finally compared through a *branch if equal* (`BEQ`) instruction. If the result of the comparison arrives in time, the program counter jumps to an instruction that resets one of the two registers to zero and sets the variable-latency again in the RF. Otherwise, if the comparison path is slow, the program continues its flow increasing the program counter. The content of the register will not be reset in that slow case. In both cases the variable-latency is set again (`RF_vl` = 1) because the program checks the content of the register through a *branch if equal to zero* (`BEQZ`) instruction which we want to be correctly executed in both fast or slow cases. The program assumes that the register was reset. In the positive case the program branches to a location in which `RF_vl` is set to zero, otherwise `RF_vl` remains at one.
ALU Adder: Testing the ALU boils down to write an assembly routine that resets to zero signal `alu_vl` and then adds two numbers chosen in such a way that the carry propagates all the way down the adder tree. The result of the addition gets stored in a general purpose register and further checked for correctness. Finally `alu_vl` gets updated.
Multiplier: This test is similar to the ALU one. It consists in disabling the two-cycle execution by resetting `mul_vl` and multiplying two numbers which activate the critical path of the higher 32-bits part of the result. The result is checked for correctness. If incorrect, signal `mul_vl` is asserted again.

5. SYNTHESIS METHOD AND RESULTS

We developed the RTL code of our RISC in VHDL and made logic synthesis experiments with a standard-cell library of a commercial CMOS 45 nm technology characterized in various process corner cases. To find upper and lower bounds of the clock frequency, we first synthesized our design in the worst and typical corners with tight constraints for the VL paths, that is by forcing them to always complete in one clock cycle. We found out a gap of 23% between typical and worst case clock frequencies (333 MHz vs. 270 MHz).

To avoid the yield loss of the typical case design, we synthesize the design in the worst case but allow some of the paths, the critical ones of the VL units, to exceed the clock cycle constraint. By doing so, we speed the worst case design up and possibly let it run at the faster speed that a typical case would permit. Of course, a higher clock frequency does not necessarily mean a higher performance and we must take into account the possible penalty for sporadic two-cycle execution. We discuss the balance between higher clock frequency and IPC penalty in section 6.

The next step is the development of ad hoc synthesis scripts to obtain our goal. A global clock constraint applies to the whole design except the critical paths of the VL units which can exceed the clock period. We used the `set_max_delay` Synopsys command for this purpose. The amount of extra delay these paths are given should be large enough so that the synthesis tool can relax them and optimize the other clock-constrained paths, but not too large: We don't want to incur any penalty, should the silicon in which the processor is deployed be faster than the worst case. Therefore the delay must be short enough so that all VL units don't need an extra clock cycle in the typical case. After a number of trial-and-error experiments it turned out that an extra delay of 1.1 ns was sufficient to this aim.

We were finally able to achieve timing closure at the same clock frequency of the typical case, 333 MHz. Interestingly, area and power figures exhibited little variations across the experiments. We can then claim that we obtained the performance of typical case design almost at no area/power penalty and without incurring the low yield of the typical case. Table 1 summarizes the results that we obtained.

Table 1: Synthesis Results. WC = worst case design; VL_WC = worst case design with variable-latency; TYP = typical case design.

	area (μm^2)	power (mW@ck freq.)	power (μW/MHz)	ck freq. (MHz)
WC	44218	4.783	17.7	270
VL_WC	44354	5.838	17.5	333
TYP	42300	6.030	18.09	333

6. VERIFICATION AND BENCHMARKING

We checked through logic simulations the correctness of our RTL code against a Verilog reference design publicly available [13][14]. The comparison concerned only the sequence of values in the time traces of the architected states (PC, register file and special register), regardless the precise timing which differs due to our LI distributed stalling logic. We used a set of test programs available on [14].

Then we cross-compiled with gcc – on a Linux x86 machine with MIPS target – the classic Dhrystone and a set of integer benchmarks taken from the MiBench suite [15]. We then ran several VHDL logic simulations with binary program and corresponding data loaded in instruction and data memories. The aim was to evaluate the VL penalty in terms of IPC with respect to a worst-case design which does not use VL. A penalty that is compensated by the increased clock frequency with respect to the worst-case approach.

Table 2: IPC Penalty (Ovh.) and Total Speed-Up (SU) with respect to WC design. Bch. = Benchmark; VL = VL configuration; M, P8, ALU, RF = Multiplier, PC+8, ALU and RF additional stalls.

Bch.	VL	M/P8/ALU/RF	Stalls	Cycles	Ovh.	SU
Dhr.	0000	0/0/0/0	0	21506	0.00%	+23%
	1111	10/0/2363/10	2383	23885	11.00%	+11%
	0001	0/0/0/10	10	21516	0.05%	+23%
	0010	0/0/2363/0	2363	23865	11.00%	+11%
	0100	0/0/0/0	0	21506	0.00%	+23%
	1000	10/0/0/0	10	21516	0.05%	+23%
Quick Sort	0000	0/0/0/0	0	139022	0.00%	+23%
	1111	125/0/13405/9	13537	152470	9.67%	+12%
	0001	0/0/0/7	7	139029	0.00%	+23%
	0010	0/0/13407/0	13407	152401	9.62%	+12%
	0100	0/0/0/0	0	139022	0.00%	+23%
	1000	125/0/0/0	125	139084	0.04%	+23%
AES	0000	0/0/0/0	0	158093	0.00%	+23%
	1111	3/0/4282/9	4292	162331	2.68%	+20%
	0001	0/0/0/7	7	158100	0.00%	+23%
	0010	0/0/4284/0	4284	162323	2.68%	+20%
	0100	0/0/0/0	0	158093	0.00%	+23%
	1000	3/0/0/0	3	158094	0.00%	+23%
CRC 32	0000	0/0/0/0	0	28063	0.00%	+23%
	1111	3/0/1238/9	1248	29308	4.44%	+18%
	0001	0/0/0/7	7	28070	0.02%	+23%
	0010	0/0/1240/0	1240	29300	4.41%	+18%
	0100	0/0/0/0	0	28063	0.00%	+23%
	1000	3/0/0/0	3	28064	0.00%	+23%
String Search	0000	0/0/0/0	0	245431	0.00%	+23%
	1111	3/0/31218/9	31228	276599	12.70%	+9%
	0001	0/0/0/7	7	245438	0.00%	+23%
	0010	0/0/31220/0	31220	276590	12.70%	+9%
	0100	0/0/0/0	0	245431	0.00%	+23%
	1000	3/0/0/0	3	245432	0.00%	+23%

Table 2 reports the results of penalty in terms of clock cycles for the selected benchmarks (overhead, Ovh. column) and the actual performance increase when the higher clock frequency is accounted for (speed-up, SU column). The VL configuration is a 4-bits code, with every digit representing VL enabled (1) or not (0) for every functional block. From left to right, the correspondence between code bits and units is the following: Multiplier, PC+8, ALU, Register File. We did not consider all possible combinations of functional units and instead evaluated the case of a single unit at a time (only one bit at logic 1) and the pessimistic case where all units are forced to operate in the slower mode (all bits at logic 1).

The results show that the largest contribution to extra latency is the ALU one whereas the remaining contributions to penalty are negligible. In fact, the worst case condition for the ALU 2's complement adder and subtracter is not as infrequent as it might seem at first glance. Every time two concordant numbers of similar absolute value are subtracted and produce a negative result, there is a sign that ripples through the bits ending in the most significant one, thus stimulating the critical path. (The same situation occurs when two discordant numbers are added.) As for the multiplier and the RF, their impact is little because multiplications are infrequent as are conditioned branches using registers R16:R31. The PC+8 adder has no impact because the address space used by the benchmarks is limited and it is never the case that the most significant bits of the PC+8 adder result switch from zero to one (or the opposite).

It turns out that in the worst case the penalty can be as high as 12.7%, but still the speed-up is between +9% and +23% thanks to the higher clock frequency. The harmonic average of the speed-up across all the benchmarks and the tested VL configurations is +19%.

As a final test we simulated the post-synthesis gate-level netlist annotated with the delays and verified its correctness against the pre-synthesis design. The annotation of delays allowed us also to check the correctness of the BIST routine which proved really effective in setting the correct VL.

7. CONCLUSIONS

In this paper we discussed the design of a latency-insensitive MIPS R2000 microarchitecture with support for variable-latency operations. A software BIST allows to discover the "status" of the silicon onto which the processor is mapped. In the typical manufacturing process case, VL is disabled and no instructions per cycle penalty occurs. In the worst case, some instructions may require an additional clock cycle to complete. However, due to the fact that these cases occur rarely and to the increase of clock frequency that our approach permits, worst-case performance gets closer to the typical one. As the results on a series of program benchmarks show, the overall speed up, which accounts for higher clock frequency and possibly lower instructions per cycle, ranges between 9% and 23%, with an harmonic average of +19%. Compared to a standard worst-case approach, the design is more efficient, because it makes clock frequency limited by frequent instructions rather than infrequent ones.

Looking ahead, we will improve the core with bypass loops, something that will reduce data-dependency stalls but will complicate the latency-insensitive control because of the *join controllers* needed to handle the bypass multiplexers.

8. REFERENCES

[1] D. Ernst *et al.* ,"Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation," Proc. MICRO-36, Dec. 2003, pp. 7–18.

[2] D. Blaauw *et al.*, "Razor II: In Situ Error Detection and Correction for PVT and SER Tolerance," Proc. ISSCC, Feb. 2008, pp. 400–622.

[3] S. Ghosh *et al.*, "CRISTA: A New Paradigm for Low-Power, Variation-Tolerant, and Adaptive Circuit Synthesis Using Critical Path Isolation," IEEE TCAD, vol. 26, no. 11, Nov. 2007, pp. 1947–1956.

[4] L.P. Carloni *et al.*, "A Methodology for Correct-by-Construction Latency Insensitive Design," Proc. ICCAD, Nov. 1999, pp. 309–315.

[5] A.J. Martin *et al.*, "The design of an asynchronous MIPS R3000 microprocessor," Proc. ARVLSI, Sep. 1997, pp. 164-181.

[6] H.M. Jacobson *et al.*, "Synchronous Interlocked Pipelines," Proc. ASYNC, Apr. 2002, pp. 3–12.

[7] J. Cortadella *et al.*, "Synthesis of Synchronous Elastic Architectures, Proc. DAC, July 2006, pp. 657–662.

[8] X. Liang and D. Brooks, "Mitigating the Impact of Process Variations on CPU Register File and Execution Units," Proc. MICRO-39, Dec. 2006, pp. 504–514.

[9] D. Bañeres *et al.*, "Variable-Latency Design by Function Speculation," Proc. DATE, Apr. 2009, pp. 1704–1709.

[10] M. Olivieri, "Design of Synchronous and Asynchronous Variable-Latency Pipelined Multipliers," IEEE TVLSI, vol. 9, no. 2, April 2001, pp. 365–376.

[11] P. Ndai *et al.*, "Trifecta: A Nonspeculative Scheme to Exploit Common, Data-Dependent Subcritical Paths," IEEE TVLSI, vol. 18, no. 1, Jan. 2010, pp. 53–65.

[12] R.P. Brent and H.T. Kung, "A Regular Layout for Parallel Adders," IEEE TCOMP, vol. C-31, no. 3, 1982, pp. 260–264.

[13] N. Pinckney *et al.*, "A MIPS R2000 Implementation," Proc. DAC, June 2008, pp. 102–107.

[14] "Google Code hmc-mips," http://code.google.com/p/hmc-mips/

[15] University of Michigan at Ann Arbor - Electrical Engineering and Computer Science Department, *MiBench: a free, commercially representative embedded benchmark suite*, http://www.eecs.umich.edu/mibench/