

A Method for Automatically Implementing FPGA-based Pipelined Microprocessors

Yu-xiang ZENG^{1,a}, Han WAN^{1,b,*}, Bo JIANG^{1,c} and Xiao-peng GAO^{1,d}

¹Beihang University, Beijing, China

^aturf1013@buaa.edu.cn, ^bwanhan@buaa.edu.cn, ^cjiangbo@buaa.edu.cn,
^dgxp@buaa.edu.cn

*Corresponding author

Keywords: Pipeline, Automatic, Stall, Bypass, Multi-cycle

Abstract. This paper presents a method of automatically generating the Verilog implementation of pipelined micro-processors. Based on the RTL descriptions of instructions, all types of hazards in pipelining are addressed optimally, especially in avoiding redundancy, reducing resource utilization and improving instruction throughput. Moreover, out-of-order execution mechanism is adopted in order to support multi-cycle instructions more efficiently. Besides, all the multiplexers and logics of control signals are analyzed and produced all by the method. The synthesized implementations of both pipelined controllers and datapaths are generated automatically, based on non-fixed architectures. A case study based on MIPS architecture not only explains the framework from input to simulation, but also illustrates the method gains almost equal performance with manual work.

Introduction

Hazards are the major hurdles of the pipelining, a widely-used technique in modern microprocessors. As many combinations of instructions and corner cases may lead to hazards in unanticipated ways eventually, it is difficult to guarantee complete considerations and perfect strategies for solving them all by manual development. Motivated by these situations, many studies have proposed some method in pipelining and they can be divided into two groups due to the degree of automation.

Using parameterizing and organizing the whole units, a pipelined microprocessor can be automatically generated[2,3,4]. These studies are usually based on one particular architecture and rely on more manual design work, but with good performance. The other group focuses on generating the major units, controllers and datapaths, completely automatically in non-fixed pipeline stage[5,6]. Even though the hazards can be resolved correctly, the performance of the final implementation can be easily damaged due to their strategies. Some potential redundancies exist in their method, which leads to more hardware resources and bad performance. In addition, he also does not mention the solutions to structural hazards related to multi-cycle instructions.

Above all, we aim to find solutions to handle automatically in all types of hazards and better support in multi-cycle instructions, since multiply and division operations are very common in real applications. Firstly, we briefly discuss our method of generating controller and datapath according to the sequences of instructions' Register Transfer Language (RTL). Next, we demonstrate the mathematical model to resolve data hazards and out-of-order mechanism to solve multi-cycle instructions with better performance. Finally, we compare the performance of our method with manual work.

Automatic Generation

Describe the Instructions Using RTL

Let uppercase letters X, Y, ... indicate the modules and lowercase letters a, b, ... indicate the ports of the modules. Therefore, sequences like X.a \rightarrow Y.b are used to represent that the port a of module X connects with the port b of module Y. Furthermore, sequences like X.a \gg are used to indicate the data from port a are needed to be pipelined. Fig. 1 is the example of an ADD instruction based on 5-stage MIPS architecture.

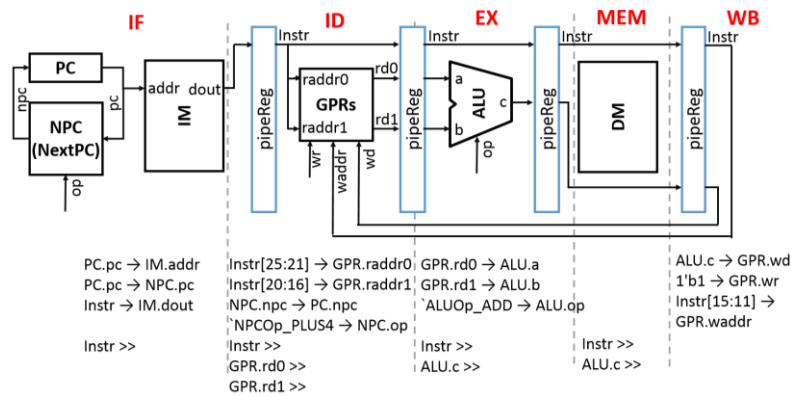


Fig. 1 An RTL example of ADD instruction

Pipelined Controller

Control signals can be naturally divided into two groups. One of them is called external control signals since they are directly appeared in RTL so that their logic could be integrated directly from RTL sequences, such as GPR.wr which denotes whether GPR is written or not. Since RTL sequences should contain their specific value for every instruction, we could easily get the combinational logic. On the contrary, other signals generated by synthesizing all RTL sequences are called internal control signals. Selecting signals of multiplexers are the most typical ones and can be divided into two kinds according to their functions. One of them is called Port Mux which one data should be selected from multiple data for different instructions. As different source ports may connect with the same target port after hashing all the RTL sequences, Port Mux is apparently required in order to choose the right source as input according to opcode of instructions. The other type is called Send Mux because bypassing implies the connections between the sending data and the functional units. The main difference between these two types is how to generate the logic of selecting signals. We will discuss the conditions of bypassing in the next section. After all the control signals are established, a controller can be easily implemented.

Pipelined Datapath

Datapath of pipelined microprocessor usually consists of core units, multiplexers, pipeline registers and the connections between them. The basic information about core units is parsed from user-defined verilog files, multiplexers generated automatically, pipeline registers produced by merging sequences like `X.a >>` at each stage. The set of `X.a` are just the inputs of pipeline registers. In the end, after appending the outputs of controller to the inputs of datapath, the datapath would be formed up eventually.

Hazard Resolution

Data Hazards

Since there are many possible combinations of instructions that can lead to data hazards, it is imperative to find a formula to choose bypassing or stalling properly. For a better explanation, let P_D be the decode stage and P_W be the write-back stage. A register model is denoted by R . An $rInsn$ is the instruction which reads the register when a data hazard occurs. The stage of $rInsn$ is denoted by P_{rInsn} . Similarly, a $wInsn$ is an instruction that writes the register when a data hazard occurs. The stage of $wInsn$ is denoted by P_{wInsn} . P_U indicates the earliest stage for the $rInsn$ using contents of register R , and P_E indicates the stage that $wInsn$ just finishes calculating the result to write back.

Firstly, we can easily define the simplest case, which consists of both $rInsn$ and $wInsn$:

$$\{(P_{rInsn}, P_{wInsn}) | \forall P_{rInsn} \in [P_D, P_U], \forall P_{wInsn} \in [P_{rInsn} + 1, P_W]\}. \quad (1)$$

Since P_W is usually the last stage and $wInsn$ should be issued earlier than $rInsn$, consequently P_{wInsn} falls into the range of $[P_{rInsn} + 1, P_W]$. While for P_{rInsn} , only ranges from P_D to P_U is needed to be considered, this is because as long as the expected value is obtained at the P_U stage, that value is held at $\forall P_i, P_i > P_U$ stage with the help of pipeline registers.

To address this case, we can stall the microprocessor until the expected values are finally calculated. This strategy is defined by

$$H(P_{rInsn}, P_{wInsn}) = \begin{cases} \text{bypassing}, & \forall P_{wInsn} \in (P_E, P_W] \\ \text{stalling}, & \text{otherwise} \end{cases} \quad (2)$$

Table 1 shows how data hazards are solved by applying (2) when $P_D = 2, P_W = 10, P_U = 6, P_E = 7$.

Table 1. Resolution to the case of hazard pairs

$P_{rInsn} \backslash P_{wInsn}$	3	4	5	6	7	8	9	10
2	stall	stall	stall	stall	stall	bypass	bypass	bypass
3		stall	stall	stall	stall	bypass	bypass	bypass
4			stall	stall	stall	bypass	bypass	bypass
5				stall	stall	bypass	bypass	bypass
6					stall	bypass	bypass	bypass

Unfortunately there are two major defects in (2):

1) Redundancies are hidden.

As stalling usually indicates $rInsn$ stays still while $wInsn$ moves forward, some hazard pairs may never appear during actually running. In Table 1, stalling is adopted when $P_{rInsn} = 2, P_{wInsn} = 3$. One cycle later, $P'_{rInsn} = 2, P'_{wInsn} = 4$ which means pairs like $P_{rInsn} = 3, P_{wInsn} = 4$ or $P_{rInsn} = 4, P_{wInsn} = 5$ never exists. Consequently, the solutions to these pairs become redundant eventually.

2) Not all stalling is necessary.

Without doubt, as long as the expected value is achieved at P_U stage in time, the data hazard is settled. However, (2) indicates stalling cannot be stopped until $wInsn$ leaves the P_E stage. However, considering this situation, when $P_{rInsn} = 2, P_{wInsn} = 7$ with no suspending, it becomes $P'_{rInsn} = 3, P'_{wInsn} = 8$ after a cycle and exactly fulfils the

condition of bypassing. Consequently $rInsn$ can still obtain the expected value, stalling is unnecessary at this particular time.

Equation (3) is further solution to overcome previous two defects:

$$H(P_{rInsn}, P_{wInsn}) = \begin{cases} \text{bypassing}, P_{rInsn} = P_D, P_{wInsn} \in (P_E, P_W] \\ \text{bypassing}, P_{rInsn} \in (P_D, P_U], P_{wInsn} = P_E + 1, \\ \text{stalling}, P_{rInsn} = P_D, P_E > P_U, \\ \quad P_{wInsn} \in [P_D + 1, P_D + P_E - P_U] \\ \text{unnecessary, otherwise} \end{cases} \quad (3)$$

PROOF. We can prove it from two aspects:

1) $P_{wInsn} > P_E$

When $P_{rInsn} = P_D, P_{wInsn} > P_E$, it indicates $wInsn$ has already finished computing the write-data according to the definition of P_E . Bypassing can be established correctly.

When $P_{rInsn} \neq P_D, P_{wInsn} = P_E + 1$, it means $wInsn$ has recently finished computing. Thus it is the proper timing to forward the data from P_{wInsn} stage to P_{rInsn} stage.

When $P_{rInsn} \neq P_D, P_{wInsn} \neq P_E + 1$, apparently at this moment $P_{rInsn} > P_D$ and $P_{wInsn} > P_E + 1$. As a result, let us compute $\Delta_{rInsn} = P_{rInsn} - P_D, \Delta_{wInsn} = P_{wInsn} - (P_E + 1)$. If $\Delta_{rInsn} \leq \Delta_{wInsn}$, thereby Δ_{rInsn} cycles before, they were

$$\begin{aligned} P'_{rInsn} &= P_{rInsn} - \Delta_{rInsn} = P_D, \\ P'_{wInsn} &= P_{wInsn} - \Delta_{rInsn} \geq P_{wInsn} - \Delta_{wInsn} = P_E + 1. \end{aligned}$$

which means at this particular moment the data from $wInsn$ should have been forwarded to $rInsn$. On the other side, if $\Delta_{rInsn} > \Delta_{wInsn}$, then Δ_{wInsn} cycles before, they were

$$\begin{aligned} P'_{rInsn} &= P_{rInsn} - \Delta_{wInsn} > P_{rInsn} - \Delta_{rInsn} > P_D, \\ P'_{wInsn} &= P_{wInsn} - \Delta_{wInsn} = P_E + 1. \end{aligned}$$

which indicates $rInsn$ has also obtained the expect value. It is also unnecessary.

2) $P_{wInsn} \leq P_E$

Let us introduce some auxiliary variables at first:

$$\Delta_{rInsn} = P_U - P_{rInsn} + 1, \Delta_{wInsn} = (P_E + 1) - P_{wInsn} + 1.$$

Δ_{rInsn} is denoted how long it will take $rInsn$ to reach P_U stage and Δ_{wInsn} is denoted how long it will cost $wInsn$ to arrive at P_E stage. The difference between them is

$$\Delta = \Delta_{wInsn} - \Delta_{rInsn} = P_E - P_U + P_{rInsn} - P_{wInsn} + 1. \quad (4)$$

If $\Delta > 0$, it actually indicates $wInsn$ needs more cycles to finish the computation, which leads to invalid bypassing. If $P_{rInsn} = P_D$ at this moment, stalling becomes the only choice. Otherwise, when $P_{rInsn} > P_D$, then $(P_{rInsn} - P_D)$ cycles before, they were

$$\begin{aligned} P'_{rInsn} &= P_{rInsn} - (P_{rInsn} - P_D) = P_D, \\ P'_{wInsn} &= P_{wInsn} - (P_{rInsn} - P_D), \\ P'_{rInsn} - P'_{wInsn} &= P_{rInsn} - P_{wInsn}, \\ \Delta' &= P_E - P_U + P'_{rInsn} - P'_{wInsn} + 1 = \Delta. \end{aligned}$$

Since $P'_{rInsn} = P_D$ and $\Delta' = \Delta > 0$, stalling should be adopted just as the former condition. Consequently, $(P_{rInsn} - P_D)$ cycles later, although $wInsn$ would arrive at P_{wInsn} stage, it is impossible for $rInsn$ to reach P_{rInsn} stage. It is redundant.

Conversely, $\Delta \leq 0$ indicates that $rInsn$ has not used the operand yet by the time $wInsn$ has just finished calculating. Then $(P_E + 1 - P_{wInsn})$ cycles later, they will be

$$\begin{aligned} P'_{rInsn} &= P_{rInsn} + (P_E + 1 - P_{wInsn}) = P_U + \Delta, \\ P'_{wInsn} &= P_{wInsn} + (P_E + 1 - P_{wInsn}) = P_E + 1 \end{aligned}$$

As $\Delta \leq 0$, then $P'_{rInsn} \leq P_U$ indicates bypassing from $wInsn$ to $rInsn$ can be established at this moment. Thus we know data hazards would be resolved after all, it is unnecessary to be considered right now.

Above all, pairs of data hazards can be resolved by (3). Besides, stalling only happens when necessary and the expected value is also forwarded once so that no further resources are required for this data hazards. Via (3), as Table 2 shows, only 8 output 30 cases are necessary in the same case of Table 1.

Table 2. Optimized resolution to previous case

$P_{rInsn} \backslash P_{wInsn}$	3	4	5	6	7	8	9	10
2	stall					bypass	bypass	bypass
3						bypass		
4						bypass		
5						bypass		
6						bypass		

Now let us consider the complex situation, which is

$$\{(P_{rInsn}, P_{wInsn}^{(0)}, P_{wInsn}^{(1)}, \dots) | P_{rInsn} \in [P_D, P_U], \\ P_{wInsn}^{(i)} \in [P_{rInsn} + 1, P_W], P_{wInsn}^{(i)} < P_{wInsn}^{(i+1)}\} \quad (5)$$

This defines the situation that at least one $wInsn$ exists along with only one $rInsn$ when a data hazard occurs. However, only the latest updated value is eager. As the subsequence $(P_{wInsn}^{(0)}, P_{wInsn}^{(1)}, \dots)$ is monotonously increasing according to (5). Actually only the conflict between P_{rInsn} and $P_{wInsn}^{(0)}$ is essential to be considered and $H(P_{rInsn}, P_{wInsn}^{(0)})$ is the resolution. We can easily construct a priority circuit based on the values of $P_{wInsn}^{(i)}$.

Finally let us solve the most difficult case when multiple data hazards occur simultaneously. In other words, some instructions may have hazards with previous instructions as a $rInsn$, meanwhile as a $wInsn$ has data hazards with subsequent instructions. According to (3), we discover that when $\forall P_{rInsn} > P_D$, $rInsn$ either has already obtained the desirable data or would definitely receive it after several cycles. So let us consider the data hazards when $\min(P_{rInsn}^{(0)}, P_{rInsn}^{(1)}, \dots) = P_{rInsn}^{(j)} = P_D$ at first. Then it is easy to determine the set of previous $wInsn$ interrelated to this $rInsn^{(j)}$ denoted by S_{wInsn} . The instructions in S_{wInsn} either do not have data hazards with previous writing instructions or the data hazard could be solved or even already solved by bypassing. Eventually it turns into the case defined by (5) and $H(P_{rInsn}, P_{wInsn}^{(j)})$ is the resolution. Similar conclusion can be drawn when $P_{rInsn}^{(j)} > P_D$.

Structural Hazards

Structural hazards are usually emerged when multiple instructions are requiring the same unit simultaneously. Multi-cycle(MC) instructions, which take several cycles to finish operations, may easily cause structural hazards. In fact, multiplication and division instructions are the most typical MC instructions. Since stalling can easily damage to the performance, out-of-order execution is adopted here to avoid that as far as possible. As irrelevant instructions are allowed to enter the stage while a multiplication or division instruction is keeping at the same stage in Multiply and Divide Unit(MDU), compilers will gain more flexibility to schedule instructions efficiently. Unfortunately structural hazards are hidden in the following situations:




- 1) Several instructions that require the same unit are issued in a certain period. E.g. many multiplication instructions are issued continuously and this would lead to structural hazards of MDU.
- 2) An MC instruction may enter the same stage along with another independent instruction. A structural hazard of the pipeline register between two stages is produced as both instructions need to move to the next stage at the same time.

The key point to avoid this hazard is to insulate the instructions which use the same hardware resource. In addition, shadow registers are used to latch the information of MC instructions like address and data of instructions in case of restoring execution of the MC instructions. Let N_{mcp} be the number of cycles to finish the multiple operations and obviously $N_{mcp} > 1$ is reasonable. Let P_{mcp} be the stage of the multiple operations. Only stalling could guarantee that an MC instruction finishes operations at P_{mcp} stage, meanwhile only a bubble steps into the same next stage. It is reasonable to make the MC instruction move to the next stage as soon as possible by ignoring the bubble. Actually the timing for the stalling is $\Delta = N_{mcp} - 1$ cycles after the MC instruction leaves the P_D stage. An MC instruction would finish operations at the P_{mcp} stage when $P_{mcp} + N_{mcp} - P_D$ cycles after leaving the P_D stage. And non MC instruction would take $P_{mcp} - P_D + 1$ cycles to leave P_{mcp} stage. Then the difference between them is the exact timing for the instruction may enter the same stage along with MC instruction:

$$\Delta = (P_{mcp} + N_{mcp} - P_D) - (P_{mcp} - P_D + 1) = N_{mcp} - 1 \quad (6)$$

A counter starts to count down from Δ as long as the MC instruction leaves P_D stage. A stalling would be impulse when the counter changes to zero. Table 3 depicts the case when $P_{mcp} = P_E$, $N_{mcp} = 4$. At the 6th cycle, normally both AND and MULT instructions would enter the P_M stage at the same time. In order to prevent the structural hazard of the pipeline register between P_E stage and P_M stage, pipeline should be stalled at the 5th cycle as well as $\Delta = 3$ cycles after MULT leaves the P_D stage.

Table 3. A case of structural hazard of pipeline register

Inst. \ Cycle	1	2	3	4	5	6	7	8
mult r1, r2	F	D	E_1	E_2	E_3	E_4		
nop		F	D	E	M	W		
nop			F	D	E	M	W	
and r3,r2,r4				F	D			
						D	E	M

Consequently the structural hazard is settled just as expected.

Control Hazards

Static predicting and delay slot are used to resolve control hazards. Many studies have already done this part efficiently. For more details, please refer to [6].

A Case Study

The Aforementioned methods was adopted to implement a textbook 5-stage MIPS microprocessor which supports 53 instructions. 35 students, who have been taught related knowledge for a whole semester, implemented the same architecture by hand. We compare the average value of manual work and automatic work in four aspects: clock frequency, the number of Flip-Flops (FF), the number of BELs (which includes all basic logic primitives like LUT, MUXCY, etc.) and power. From the result in Table 4, it is obvious to conclude that automatic method has comparable performance as average of manual works and gain the best clock frequency with a little more resources.

Table 4. Comparisons between manual and automatic work

		Clock (MHz)	FF	BEL	Power (mW)
Manual	Best	137.86	1442	2373	146.67
	Average	67.40	1903	4701	203.32
Automatic		155.65	1674	3143	188.15

Conclusions

We introduce an overall method for automatically generating pipelined controllers and pipelined datapaths at first. For data hazards, we propose the resolution that can guarantee all the strategies are necessary to gain better performance. We also explain how to support multi-cycle instructions with out-of-order execution and how to resolve structural hazards caused by them. At last, a case study shows that the generated microprocessors have nearly equal performance with the manual work.

Acknowledgement

Our thanks to the support from Beihang University's Teaching Research Foundation(4006007).

References

- [1] L. Hennessy and D. A. Patterson. Computer Architecture-A Quantitative Approach (5. ed.). Morgan Kaufmann, 2012.
- [2] M. Itoh, S. Higaki, Y. Takeuchi, A. Kitajima, M. Imai, J. Sato, and A. Shiomi. PEAS-III: an ASIP design environment. ICCD'00, Sep. 2000, pages 430–436.
- [3] P. Mishra, A. Kejariwal, and N. Dutt. Synthesis-driven exploration of pipelined embedded processors. VLSI'04, Jan. 2004, pages 921–926.
- [4] H. Y. Cheah, S. A. Fahmy, and N. Kapre. Analysis and optimization of a deeply pipelined FPGA soft processor. FPT'14, Dec. 2014, pages 235–238.
- [5] D. Kroening and W. J. Paul. Automated pipeline design. DAC'01, June, 2001, pages 810–815.

- [6] E. Nurvitadhi, J. C. Hoe, T. Kam, and S. Lu. Automatic pipelining from transactional datapath specifications. TCAD'11, 30(3):441–454.
- [7] P. Yiannacouras, J. G. Steffan, and J. Rose. Exploration and customization of fpga-based soft processors. TCAD'07, 26(2):266–277.