Information Flow Analysis of a Verified In-Order Pipelined Processor

Ning Dong[1], Roberto Guanciale[1], Mads Dam[1], and Andreas Lööw[2]

[1]KTH Royal Institute of Technology, Sweden
[2]Imperial College London, UK

Abstract

We implement a verified in-order pipelined processor Silver-Pi for the RISC ISA Silver in the HOL4 interactive theorem prover. The correctness of the processor is established by exhibiting a trace relation between the circuit and the Silver ISA. Based on the correctness proof, we prove the conditional noninterference (CNI) of the processor. The CNI formulates that executing programs on the processor does not leak additional information on the timing channel than permitted by a leakage function expressed at the ISA level.

1 Introduction

Silver-Pi is a 5-stage in-order pipelined processor for the RISC ISA Silver [2]. The processor is implemented using the HOL4 Verilog library [2, 3] for formally verified circuits. The correctness of the processor is proved by exhibiting a trace relation between the pipelined circuit and the Silver ISA. The trace relation is constructed using a unique scheduling function that indicates the processed ISA-level instruction in a pipeline stage. The circuit implementation and correctness proof are formalized in HOL4 [1], and accessible at `https://github.com/kth-step/Silver-Pi`.

To prevent timing side channels, we define the notion of conditional noninterference (CNI) which formulates that the processor does not leak more information via the circuit's timing channel than what is expected by a leakage model expressed at the ISA level. In this technique report, we explicitly show the non-mechanized proof of the CNI for Silver-Pi.

The report is constructed as follows:

- Section 2 introduces the background, mainly the processor implementation and correctness proof.
- Section 3 describes the CNI proof for the processor.

2 Background

This section presents the background of our formalization including circuit implementation and correctness proof. The definitions and proofs in Section 2 are available in our HOL4 formalisation.

2.1 Silver ISA

The semantics of the Silver ISA is modelled by a state transition relation: $s \to s'$, which represents the atomic execution of one instruction. The ISA trace σ is produced by $\to$ starting from an initial state s. Formally, $\sigma = s \to s' \to s'' \cdots$. The $\sigma(n)$ represents the ISA state after completing n instructions in σ, i.e., $\sigma(n) = \ s \to^n$.

The Silver ISA state is a record, $s = (PC, M, R, CF, OF, DI, DO, ME)$. Here, PC, M, and R are the program counter, memory, and register file respectively. Two flags (CF and OF) are used to record carry and overflow for the `ALU` (arithmetic logic unit) add and subtraction computations, DI and DO are two data ports for I/O operations, and ME is a trace to record memory states but is never used by the ISA. The following items are intermediate fields used by the Silver ISA to process instructions:

- *opc*: operation code indicates the current operation of the instruction.
- *func*: function code indicates the current functionality of `ALU` or the shift operation `SHF`.
- *Ra*, *Rb*, *Rw*: data resource fields are followed by their corresponding flags (*Fa*, *Fb*, and *Fw*) to indicate it as a register address (flag is 0) or an immediate constant (flag is 1).
- *Da*, *Db*, *Dw*: data read from R with their addresses if the flag is 0, otherwise, the constant.
- *ad*, *v*: data address and value for memory load and store.

We highlight the following operations in the Silver ISA since they affect the pipeline implementation and verification:

- `JMP` and `CJMP`: unconditional and conditional jumps.
- `MLD` and `MSTR`: memory load and store.
- `INTR`: interrupt.
- `ACC`: acceleration.

The Silver ISA model completes all instructions internally i.e., without any interaction with the external environment. However, a Silver processor requires communications with external hardware components like memory to process some kinds of instructions (including `MLD`, `MSTR`, `INTR`, and `ACC`).

2.2 Pipelined Circuit

Silver-Pi implements a typical 5-stage pipeline as shown in Figure 1.

2.2.1 Pipeline challenges

The pipelined processor handles common pipeline challenges including data hazards, external delays, and mispredicted program counters.

Data hazards The pipeline may process interdependent instructions, for example, a program of two instructions `i0: R1 := R0+1; i1: R2 := R1-2;`. The pipeline must prevent `i1` from reading a wrong (old) value for the register `R1` in the **ID** stage, when `i0` is still being processed in the pipeline and its result has not been committed to the register file. Silver-Pi uses pipeline

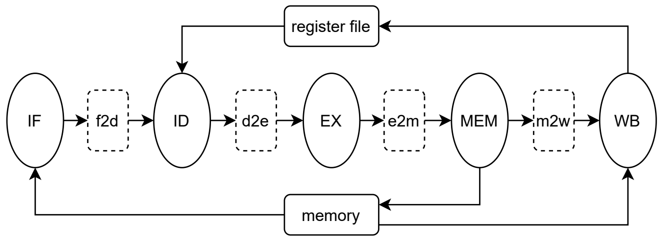

Figure 1: Simplified view of a 5-stage pipeline

stalling by checking whether each register's address (Ra, Rb, Rw) is affected by instructions in the **EX**, **MEM**, and **WB** stages. If data hazards are identified, a control unit stalls the instruction in **ID** stage until data hazards disappear.

External delays Requests issued by the **MEM** stage can take several processor cycles to be answered. Normally any new request to the same external hardware component will be ignored during the waiting cycles. For instance, when the **MEM** stage issues a `MLD` request to the memory, the pipeline is stalled until the memory replies the result of `MLD` to the **WB** stage, which is then committed to the register file R. The same approach is applied to the other three kinds of Silver instructions that communicate with external components and that can take several hardware cycles to be answered: `MSTR`, `INTR`, and `ACC`.

Mispredicted program counters The pipeline fetches instructions speculatively for the next cycle until the next PC is determined. These speculatively fetched instructions can be wrong when an instruction in the pipeline modifies the program counter (i.e., `JMP` and `CJMP`). Consider the example in Table 1, instructions `i0 - i3` are regular operations, and `i4` is a `JMP`. The instructions `i4'` and `i4''` are stored in the next two addresses to `i4` in the memory, and `i5` is stored at the target address of `i4`. The `i4'` and `i4''` are speculatively fetched at the cycle $t + 1$ and $t + 2$. In Silver ISA, target addresses of jumps are determined only after the `ALU` results are available in the **EX** stage. In addition, the PC cannot be affected by external hardware components like memory. For these reasons, we implement a jump handler in the **EX** circuit. For the example in Table 1, after the **EX** circuit computes the target of `i4`, the instructions `i4'` and `i4''` are flushed as `NOP` (no operation) and the proper next instruction `i5` is fetched at $t + 3$.

2.2.2 Circuit states

The processor state c contains all fields used by the pipelined circuit. To process instructions, the processor communicates with an external environment mainly a memory subsystem and interrupt

	IF	**ID**	**EX**	**MEM**	**WB**
t	`i4(JMP)`	`i3`	`i2`	`i1`	`i0`
$t+1$	`i4'`	`i4(JMP)`	`i3`	`i2`	`i1`
$t+2$	`i4''`	`i4'`	`i4(JMP)`	`i3`	`i2`
$t+3$	`i5`	`NOP`	`NOP`	`i4(JMP)`	`i3`

Table 1: Processing a jump in the pipeline

handler. Since c contains more than 100 fields, we show the following fields that the CNI proof depends on and omit others.

- PC_g, cmd_g, ir_g, ad_g: the program counter, command issuing fetch/load/store requests to the memory, interrupt request, and data address respectively. They are the processor's output fields for interacting with the environment.
- st_g: the processor's internal state identifies if the processor is waiting for the response from the external environment, or working normally.
- ast_g: the state of the internal accelerator identifies the process of `ACC` computation.
- $exRa_{id}$,$memRa_{id}$,$wbRa_{id}$,$exRb_{id}$,$memRb_{id}$,$wbRb_{id}$,$exRw_{id}$,$memRw_{id}$,$wbRw_{id}$,: these flags check whether the reading addresses (Ra_{id}, Rb_{id}, and Rw_{id}) in the **ID** stage are affected by instructions in the **EX**, **MEM**, **WB** stages. For simplicity, we use hzd_{id} to represent them.
- jmp_{ex}: a jump flag in the **EX** stage indicates that a `JMP` or `CJMP` modifies the program counter PC_g.
- mld_{mem}, $mstr_{mem}$, $intr_{mem}$, acc_{mem}: flags in the **MEM** stage represent the `MLD`, `MSTR`, `INTR`, and `ACC` request respectively to the environment or internal accelerator, which may take several hardware cycles to be answered.
- $enable_{if}$, $enable_{id}$, $enable_{ex}$, $enable_{mem}$, $enable_{wb}$, $flush_{id}$, $flush_{ex}$, $flush_{mem}$: these flags control the pipeline stages and maintain the instruction processing, called control flags *ctrl*. The enable flags enable the corresponding pipeline stage to process a new instruction delivered from the previous stage. The flush flags flush the stage when certain pipeline challenges happen (i.e., data hazards, external delays, and mispredicted program counters). The **IF** stage does not need a flush flag as the flush is done by modifying the program counter PC_g. The pipelined circuit does not flush the **WB** stage as flushes are already done in previous stages.

The environment state is a record, $e = (M, DI, inst, data, rdy, mirdy, iack)$. The M and DI are the same memory and data port as the ISA state. Other fields excluding M and DI are the environment's outputs to the processor. The *inst* and *data* are instruction and data values from the memory, and *rdy* indicates the memory request is finished and the memory is able to process the next request. The *mirdy* means the memory initialization is finished. The *iack* is an acknowledgement of the interrupt handler to the processor to inform that the `INTR` request is finished.

2.3 Correctness

Giving an environment trace $\beta = e \to e' \to e'' \cdots$, the circuit definition $ag\pi$ generates the processor's execution trace $\alpha = c \to c' \to c'' \cdots$, $\alpha(t)$ and $\beta(t)$ are the processor and environment state at the cycle t respectively. By composing the processor and environment, ϕ represents the circuit execution traces, i.e., traces such that exist $\alpha = ag\pi\ \beta$ and $\phi(t) = (\alpha(t), \beta(t))$.

Environment assumption The environment trace β in ϕ satisfies an assumption AX that describes the expected behaviour of the environment's components including a memory subsystem *mem_env*, a memory initialization controller *mem_start_env*, an interrupt handler *intr_env*, and a data port controller *di_env*. Formally:

Definition 1.

$$
\begin{aligned}
\mathit{mem\_env}\ \phi \triangleq{} & \forall\ t.(\alpha(t).cmd_g = fetch \wedge \beta(t-1).rdy \Rightarrow \\
& \exists m.(\forall p \leq m.\ \beta(t+p).M = \beta(t-1).M)\ \wedge \\
& \quad (\forall p < m.\ \neg\beta(t+p).rdy)\ \wedge\ \beta(t+m).rdy\ \wedge \\
& \quad \beta(t+m).inst = \beta(t).M[\alpha(t).PC_g])\ \wedge \\
& \quad (\alpha(t).cmd_g = fetch + load \wedge \beta(t-1).rdy \Rightarrow \\
& \exists m.(\forall p \leq m.\ \beta(t+p).M = \beta(t-1).M)\ \wedge \\
& \quad (\forall p < m.\ \neg\beta(t+p).rdy)\ \wedge\ \beta(t+m).rdy\ \wedge \\
& \quad \beta(t+m).inst = \beta(t).M[\alpha(t).PC_g]\ \wedge \\
& \quad \beta(t+m).data = \beta(t).M[\alpha(t).ad_g])\ \wedge \\
& \quad (\alpha(t).cmd_g = fetch + store \wedge \beta(t-1).rdy \Rightarrow \\
& \exists m.(\forall p \leq m.\ \beta(t+p).M = \beta(t-1).M)\ \wedge \\
& \quad (\forall p < m.\ \neg\beta(t+p).rdy)\ \wedge\ \beta(t+m).rdy\ \wedge \\
& \quad \beta(t+m).M = \beta(t).M\langle|\alpha(t).ad_g := \alpha(t).v_g|\rangle\ \wedge \\
& \quad \beta(t+m).inst = \beta(t).M[\alpha(t).PC_g])\ \wedge \\
& \quad (\alpha(t).cmd_g = nothing \wedge \beta(t-1).rdy \Rightarrow \\
& \quad \beta(t).rdy\ \wedge\ \beta(t).M = \beta(t-1).M\ \wedge \\
& \quad \beta(t).inst = \beta(t-1).inst\ \wedge\ \beta(t).data = \beta(t-1).data) \\
\mathit{mem\_start\_env}\ \phi \triangleq{} & \exists m.\ \beta(m).mirdy \\
\mathit{intr\_env}\ \phi \triangleq{} & \forall\ t.\alpha(t).ir_g \wedge \beta(t-1).iack \Rightarrow \\
& \exists m.(\forall p < m.\ \neg\beta(t+p).iack)\ \wedge\ \beta(t+m).iack \\
\mathit{di\_env}\ \phi \triangleq{} & \forall\ t.\beta(t).DI = \beta(0).DI \\
AC\ \phi \triangleq{} & \mathit{mem\_env}\ \phi \wedge \mathit{mem\_start\_env}\ \phi \wedge \mathit{intr\_env}\ \phi \wedge \mathit{di\_env}\ \phi
\end{aligned}
$$

Scheduling function The scheduling function I maps the processing instruction in a pipeline stage k at cycle t, which is defined as follows:

Definition 2.

$$
\begin{aligned}
&(I(k,0) = \mathit{if}\ k = \mathbf{IF}\ \mathit{then}\ 1\ \mathit{else}\ \bot)\ \wedge\\
&(I(k,t+1) = \mathit{case}\ k\ \mathit{of}\\
&\quad \mathbf{IF} \Rightarrow (\mathit{if}\ \neg\alpha(t).\mathit{enable}_{if}\ \mathit{then}\ I(\mathbf{IF},t)\\
&\qquad \mathit{else\ if}\ \alpha(t).\mathit{enable}_{if} \wedge \alpha(t).\mathit{jmp}_{ex}\ \mathit{then}\ I(\mathbf{EX},t)+1\\
&\qquad \mathit{else\ if}\ \alpha(t).\mathit{enable}_{if} \wedge (\mathit{is\_jmp}_{isa}\ \sigma(I(\mathbf{IF},t)-1) \vee \mathit{is\_jmp}_{isa}\ \sigma(I(\mathbf{ID},t)-1))\ \mathit{then}\ \bot\\
&\qquad \mathit{else}\ I(\mathbf{IF},t)+1)\\
&\quad \mathbf{ID} \Rightarrow (\mathit{if}\ \neg\alpha(t).\mathit{enable}_{id}\ \mathit{then}\ I(\mathbf{ID},t)\\
&\qquad \mathit{else\ if}\ \alpha(t).\mathit{enable}_{id} \wedge (\alpha(t).\mathit{jmp}_{ex} \vee \mathit{is\_jmp}_{isa}\ \sigma(I(\mathbf{ID},t)-1))\ \mathit{then}\ \bot\\
&\qquad \mathit{else}\ I(\mathbf{IF},t))\\
&\quad \mathbf{EX} \Rightarrow (\mathit{if}\ \neg\alpha(t).\mathit{enable}_{ex}\ \mathit{then}\ I(\mathbf{EX},t)\\
&\qquad \mathit{else\ if}\ \alpha(t).\mathit{enable}_{ex} \wedge (\alpha(t).\mathit{jmp}_{ex} \vee \alpha(t).\mathit{hzd}_{id})\ \mathit{then}\ \bot\\
&\qquad \mathit{else}\ I(\mathbf{ID},t))\\
&\quad \mathbf{MEM} \Rightarrow (\mathit{if}\ \neg\alpha(t).\mathit{enable}_{mem}\ \mathit{then}\ I(\mathbf{MEM},t)\\
&\qquad \mathit{else\ if}\ \alpha(t).\mathit{enable}_{mem} \wedge (\alpha(t).\mathit{mld}_{mem} \vee \alpha(t).\mathit{mstr}_{mem} \vee\\
&\qquad \alpha(t).\mathit{intr}_{mem} \vee \alpha(t).\mathit{acc}_{mem})\ \mathit{then}\ \bot\\
&\qquad \mathit{else}\ I(\mathbf{EX},t))\\
&\quad \mathbf{WB} \Rightarrow (\mathit{if}\ \neg\alpha(t).\mathit{enable}_{wb}\ \mathit{then}\ I(\mathbf{WB},t)\\
&\qquad \mathit{else}\ I(\mathbf{MEM},t))
\end{aligned}
$$

The ISA-level function $\mathit{is\_jmp}_{isa}$ checks `JMP` and `CJMP` with its condition, formally $\mathit{is\_jmp}_{isa}\ s = (\mathit{decode\_opc}_{isa}\ s = \texttt{JMP}) \vee (\mathit{decode\_opc}_{isa}\ s = \texttt{CJMP} \wedge \mathit{cjmp\_cond}_{isa}\ s)$, where the function $\mathit{decode\_opc}_{isa}$ decodes the ISA state's *opc* and identifies the current operation, and the function $\mathit{cjmp\_cond}_{isa}$ generates the condition for `CJMP`.

Software condition To prevent self-modifying programs, any software executing on our pipelined processor must follow the software condition SC that no instruction is modified in the memory by the previous four instructions being processed in the pipeline (see Definition 3). The circuit behaviour is undefined when SC is violated.

Definition 3.

$$
\begin{aligned}
SC\ \sigma \triangleq \forall n.\ \mathit{decode\_opc}_{isa}\ \sigma(n) = \texttt{MSTR}\ \wedge\ n < i < n+5 \Rightarrow\\
\sigma(i).PC \neq \mathit{mem\_ad}_{isa}\ \sigma(n)
\end{aligned}
$$

The function $\mathit{mem\_ad}_{isa}$ extracts the data address *ad* for memory operations.

Relations To demonstrate the equivalence between the pipelined circuit and ISA, the trace relation $\sim_I$ is defined with the help of the scheduling function I. The initial relation $\sim_0$ guarantees that the circuit and ISA start from corresponding initial states: $\phi(0) \sim_0 \sigma(0)$. Because the $\sim_I$ and $\sim_0$ argue details of the circuit implementation, we omit the concrete definitions here and refer the reader to our HOL4 formalization.

Theorem 1. *If the initial circuit and ISA states are consistent $\phi(0) \sim_0 \sigma(0)$, the external environment satisfies $AX\ \phi$, and the program satisfies the software condition $SC\ \sigma$, then the trace relation is met with a unique scheduling function for ϕ and σ: $\phi \sim_I \sigma$.*

In the following, we use $\simeq_I$ to represent that ϕ corresponds to σ: $\phi \simeq_I \sigma \triangleq AX\ \phi \wedge \phi \sim_0 \sigma \wedge \phi \sim_I \sigma$.

3 Information Flow Security

This section presents additional definitions for information flow analysis and the proof for conditional noninterference of Silver-Pi steply starting from key lemmas to the final theorem.

3.1 Definitions

ISAs serve as the main interface between software and hardware, ensuring the correctness and security of software. However, ISAs do not capture non-functional aspects of systems, like the execution time, that can be utilized by an attacker to infer confidential data. To scope our work, we consider the attacker as an external agent that can monitor the timing channel when outputs are produced by our system. For Silver, this corresponds to measuring the clock cycles elapsed between `INTR`. It is usually infeasible to verify resilience against side channels by taking into account both software and processor design at the same time. In practice, these analyses are usually done by using observational models, which extend the ISA with leakage functions that overapproximate what influences the side channels.

Observation function The observation function obs_{ag} extracts the part of the Silver ISA state that can affect the execution time of a program, $s_1 \approx_{obs_{ag}} s_2$ means that these states are indistinguishable by the attacker. Formally,

Definition 4.

$$\begin{aligned} s_1 \approx_{obs_{ag}} s_2 \triangleq\ & (s_1.PC = s_2.PC) \wedge (s_1.M[s_1.PC] = s_2.M[s_2.PC]) \wedge \\ & (decode\_opc_{isa}\ s_1 = \texttt{MLD} \vee \texttt{MSTR} \Rightarrow mem\_ad_{isa}\ s_1 = mem\_ad_{isa}\ s_2) \wedge \\ & (decode\_opc_{isa}\ s_1 = \texttt{CJMP} \Rightarrow cjmp\_cond_{isa}\ s_1 = cjmp\_cond_{isa}\ s_2) \end{aligned}$$

These ISA-level functions are defined in our HOL4 formalization and used for correctness proof. Since obs_{ag} requires the two ISA states to process the same instruction, $decode\_opc_{isa}\ s_1 = decode\_opc_{isa}\ s_2$ directly.

We extend observation equivalence pointwise to ISA traces $\sigma_1 \approx_{obs} \sigma_2$. Notice that since the attacker observes the PC, observation equivalent traces have the same length, as is common for constant time programming.

Conditional noninterference The strategy is to take the ISA traces as a reference for permitted information flows: $\sigma_1 \approx_{obs} \sigma_2$ means that these traces are indistinguishable by the attacker. For their corresponding circuit traces $\phi_1 \simeq_{I_1} \sigma_1$ and $\phi_2 \simeq_{I_2} \sigma_2$, the circuit is secure if the traces do not leak more information than the ISA level. To guarantee that, we require $I_1(k,t) = I_2(k,t)$ for all pipeline stages k at every cycle, which means that the two circuit traces process the same instruction at the same stage and therefore have the same time observations. For our attacker model, we need to ensure $I_1(k,t) = I_2(k,t)$ at the cycle t when `INTR` happens. To guarantee this, it is necessary to keep track of instruction processing in the circuit over time. Let Σ be the set of valid ISA traces that processors can implement, conditional noninterference is defined as follows:

Definition 5. *A pipelined circuit is conditional noninterferent with respect to the observation function obs, written CNI(obs), if for any two ISA traces σ_1 and σ_2 in Σ such that $\sigma_1 \approx_{obs} \sigma_2$, for any* *circuit trace ϕ_1 with a scheduling function I_1 satisfying $\phi_1 \simeq_{I_1} \sigma_1$, there exists a circuit trace ϕ_2 and scheduling function I_2 such that $\phi_2 \simeq_{I_2} \sigma_2$, and $\forall k\ t.\ I_1(k,t) = I_2(k,t)$.*

$$\begin{aligned} CNI(obs) \triangleq\ & \forall \sigma_1\ \sigma_2\ \phi_1\ I_1.\ \sigma_1 \approx_{obs} \sigma_2 \ \wedge\ \phi_1 \simeq_{I_1} \sigma_1 \Rightarrow \\ & \exists \phi_2\ I_2.\ \phi_2 \simeq_{I_2} \sigma_2 \ \wedge\ (\forall k\ t.\ I_1(k,t) = I_2(k,t)) \end{aligned}$$

Environment constraint To reason about the circuit's timing channel, we use an environment constraint EC_{ag} which requires that two environment traces β_1 and β_2 respond to their processor traces α_1 and α_2 respectively at the same cycle t if all processor's requests before t are identical. The EC_{ag} is defined as follows to constrain the memory subsystem, the interrupt handler, and the memory initialization controller. The data port controller is excluded as it does not affect the execution time of programs.

Definition 6.

$$\begin{aligned} EC_{ag}(\phi_1, \phi_2) \triangleq\ & (\forall t\ t'.\ t' \leqslant t \wedge \alpha_1(t').cmd_g = \alpha_2(t').cmd_g \wedge \alpha_1(t').PC_g = \alpha_2(t').PC_g \wedge \\ & \quad (\alpha_1(t').cmd_g = load/store \Rightarrow \alpha_1(t').ad_g = \alpha_2(t').ad_g) \Rightarrow \\ & \quad \beta_1(t).rdy = \beta_2(t).rdy) \wedge \\ & (\forall t\ t'.\ t' \leqslant t \wedge \alpha_1(t').ir_g = \alpha_2(t').ir_g \Rightarrow \beta_1(t).iack = \beta_2(t).iack) \wedge \\ & (\forall t.\ \beta_1(t).mirdy = \beta_2(t).mirdy) \end{aligned}$$

Circuit low-equivalence Some circuit fields can either directly affect the scheduling results like $enable_{ex}$ or be observed by the environment like cmd_g, and thus impact the execution time of programs. Therefore, we define the circuit low-equivalence $\approx_f$ of these fields between two circuit traces.

Definition 7.

$$\begin{aligned} \phi_1 \approx_f \phi_2 \triangleq \forall t.\ & \alpha_1(t).PC_g = \alpha_2(t).PC_g \wedge \alpha_1(t).cmd_g = \alpha_2(t).cmd_g \wedge \alpha_1(t).ad_g = \alpha_2(t).ad_g \wedge \\ & \alpha_1(t).st_g = \alpha_2(t).st_g \wedge \alpha_1(t).ir_g = \alpha_2(t).ir_g \wedge \\ & \alpha_1(t).ast_g = \alpha_2(t).ast_g \wedge \alpha_1(t).ctrl = \alpha_2(t).ctrl \end{aligned}$$

The PC_g is updated by the processor function *IF_PC_update*. The next 4 fields are operated by the processor function *agp32_next_state* that uses the circuit state at the cycle t to update these fields for the next cycle. The ast_g is updated by the accelerator *acc_compute*, and the control flags *ctrl* are generated by the function *Hazard_ctrl* at a cycle t and then take effect at the next cycle. As a part of the processor implementation, the above 4 functions are available in our HOL4 code.

3.2 Proofs

Lemma 1 shows that if the maximal result of the scheduling function at the cycle $t+1$ is less or equal to n, then the condition is also true for the previous cycle t.

Lemma 1. *If $\forall k.\ I(k,t+1) \neq \bot \Rightarrow I(k,t+1) \leq n$, then $\forall k.\ I(k,t) \neq \bot \Rightarrow I(k,t) \leq n$.*

Proof. Lemma 1 is proved by checking every pipeline stage. If a pipeline stage is disabled for the cycle $t+1$, i.e. $\neg\phi(t).enable_k$, then $I(k,t+1) = I(k,t) \neq \bot$. From the assumption, $I(k,t) \leq n$ is proved. When a stage is enabled $\phi(t).enable_k$, the following applies:

- **IF**: If a jump happens in the **EX** stage at the cycle t ($\phi(t).jmp_{ex}$), then $I(\mathbf{IF}, t) = \bot$ since the next two instructions after `JMP` or `CJMP` are wrongly fetched. For $\bot$, the proof is automatically done as it violates the assumption. If there is no jump in the **EX** stage but the **ID** stage processed a jump at the cycle t, then $I(\mathbf{IF}, t) = \bot$ as well. Otherwise for regular cases, because $\forall t.\ \phi(t).enable_{if} = \phi(t).enable_{id}$ from an internal lemma in Theorem 1, $I(\mathbf{ID}, t+1) = I(\mathbf{IF}, t) \neq \bot$, meaning $I(\mathbf{IF}, t) \leq n$.

- **ID**: If $\phi(t).jmp_{ex}$, then $I(\mathbf{ID}, t) = \bot$. If there is no jump but data hazards are identified at the cycle t ($\phi(t).hzd_{id}$), then the **ID** stage is disabled for the cycle $t+1$, i.e. $\neg\phi(t).enable_{id}$, which violates the condition and thus ignored. Otherwise, because of a lemma $\forall t.\ \phi(t).enable_{id} \Rightarrow \phi(t).enable_{ex}$, $I(\mathbf{EX}, t+1) = I(\mathbf{ID}, t) \neq \bot$, the proof is done.

- **EX**: If there is a request issued by the **MEM** stage at the cycle t i.e., $\phi(t).mld_{mem} \vee \phi(t).mstr_{mem} \vee \phi(t).intr_{mem} \vee \phi(t).acc_{mem}$, then the **EX** stage is disabled for the cycle $t+1$, i.e. $\neg\phi(t).enable_{ex}$. Otherwise, $\forall t.\ \phi(t).enable_{ex} \Rightarrow \phi(t).enable_{mem}$, $I(\mathbf{MEM}, t+1) = I(\mathbf{EX}, t) \neq \bot$, so $I(\mathbf{EX}, t) \leq n$.

- **MEM**: A lemma shows $\forall t.\ \phi(t).enable_{mem} = \phi(t).enable_{wb}$, so $I(\mathbf{WB}, t+1) = I(\mathbf{MEM}, t) \neq \bot$ and $I(\mathbf{MEM}, t) \leq n$.

- **WB**: As the above **MEM** stage mentioned, $I(\mathbf{WB}, t+1) = I(\mathbf{MEM}, t)$. The proof considers possible $\bot$ cases in the pipeline:

  1. $I(\mathbf{MEM}, t) \neq \bot$: For this case, $I(\mathbf{WB}, t) \neq \bot \wedge I(\mathbf{WB}, t+1) \neq \bot$. $I(\mathbf{WB}, t) < I(\mathbf{WB}, t+1)$ according to a lemma in the correctness proof. Since $I(\mathbf{WB}, t+1) \leq n$, $I(\mathbf{WB}, t) \leq n$.
  2. $I(\mathbf{MEM}, t) = \bot \wedge I(\mathbf{EX}, t) \neq \bot$: $I(\mathbf{WB}, t) < I(\mathbf{EX}, t)$ from a correctness lemma, $I(\mathbf{MEM}, t+1) = I(\mathbf{EX}, t) \neq \bot$ because of the pipeline scheduling. So, $I(\mathbf{WB}, t) < I(\mathbf{MEM}, t+1)$ and $I(\mathbf{MEM}, t+1) \leq n$ from the assumption, the proof is done.
  3. $I(\mathbf{MEM}, t) = \bot \wedge I(\mathbf{EX}, t) = \bot \wedge I(\mathbf{ID}, t) \neq \bot$: Similarly, $I(\mathbf{WB}, t) < I(\mathbf{ID}, t)$ and $I(\mathbf{EX}, t+1) = I(\mathbf{ID}, t) \neq \bot$. So, $I(\mathbf{WB}, t) < I(\mathbf{EX}, t+1) \leq n$.
  4. $I(\mathbf{MEM}, t) = \bot \wedge I(\mathbf{EX}, t) = \bot \wedge I(\mathbf{ID}, t) = \bot \wedge I(\mathbf{IF}, t) \neq \bot$: $I(\mathbf{WB}, t) < I(\mathbf{IF}, t)$ and $I(\mathbf{ID}, t+1) = I(\mathbf{IF}, t) \neq \bot$. So, $I(\mathbf{WB}, t) < I(\mathbf{ID}, t+1) \leq n$.
  5. all stages except for **WB** are $\bot$: this case is impossible since an internal lemma indicates that $\forall t. I(\mathbf{ID}, t) = \bot \wedge I(\mathbf{EX}, t) = \bot \Rightarrow I(\mathbf{IF}, t) \neq \bot$.

□

As Definition 2 demonstrated, pipeline challenges at a cycle t affect the scheduling results for the next cycle. To ensure the same scheduling results in two circuit traces, Lemma 2, 3 and 4 guarantee that the jmp_{ex}, $mld/mstr/intr/acc_{mem}$, and hzd_{id} in two circuit traces have the same value at the cycle t respectively.

Lemma 2. *For any $\phi_1 \simeq_{I_1} \sigma_1$ and $\phi_2 \simeq_{I_2} \sigma_2$, $\sigma_1 \approx_{obs_{ag}} \sigma_2$, if the programs in two ISA traces are not self modifying SC σ_1 and SC σ_2, $\forall k. I_1(k,t) = I_2(k,t)$ at the cycle t, and $\forall k.\ I(k,t+1) \neq \bot \Rightarrow I(k,t+1) \leq n$ where n is the length of ISA traces σ_1 and σ_2, then $\phi_1(t).jmp_{ex} = \phi_2(t).jmp_{ex}$.*

Proof. The proof is related to the **EX** stage where the jump handler of Silver-Pi is located. From the assumption $\forall k. I_1(k,t) = I_2(k,t)$, $I_1(\mathbf{EX},t) = I_2(\mathbf{EX},t)$. If the scheduling result is $\bot$, then $\neg\phi_1(t).jmp_{ex} \wedge \neg\phi_2(t).jmp_{ex}$ as the correctness theorem 1 required. Otherwise $I_1(\mathbf{EX},t) = I_2(\mathbf{EX},t) = m$, and $m < n$ from Lemma 1. So, $\sigma_1(m-1) \approx_{obs_{ag}} \sigma_2(m-1)$. According to Theorem 1, $\phi_1(t).jmp_{ex} = is\_jmp_{isa}\ \sigma_1(m-1)$ and $\phi_2(t).jmp_{ex} = is\_jmp_{isa}\ \sigma_2(m-1)$. As the obs_{ag} regulated, the two ISA states have the same *opc* and conditions of `CJMP`, i.e., $decode\_opc_{isa}\ \sigma_1(m-1) = decode\_opc_{isa}\ \sigma_2(m-1) \wedge cjmp\_cond_{isa}\ \sigma_1(m-1) = cjmp\_cond_{isa}\ \sigma_2(m-1)$. Therefore, $is\_jmp_{isa}\ \sigma_1(m-1) = is\_jmp_{isa}\ \sigma_2(m-1)$, and $\phi_1(t).jmp_{ex} = \phi_2(t).jmp_{ex}$ is proved. □

Lemma 3. *For any $\phi_1 \simeq_{I_1} \sigma_1$ and $\phi_2 \simeq_{I_2} \sigma_2$, $\sigma_1 \approx_{obs_{ag}} \sigma_2$, if the programs in two ISA traces are not self modifying SC σ_1 and SC σ_2, $\forall k. I_1(k,t) = I_2(k,t)$ at the cycle t, and $\forall k.\ I(k,t+1) \neq \bot \Rightarrow I(k,t+1) \leq n$ where n is the length of ISA traces σ_1 and σ_2, then $\phi_1(t).mld_{mem} = \phi_2(t).mld_{mem} \wedge \phi_1(t).mstr_{mem} = \phi_2(t).mstr_{mem} \wedge \phi_1(t).intr_{mem} = \phi_2(t).intr_{mem} \wedge \phi_1(t).acc_{mem} = \phi_2(t).acc_{mem}$.*

Proof. These request fields are based on the operation code in the **MEM** stage opc_{mem}. For example, $\phi_1(t).mld_{mem} = (\phi_1(t).opc_{mem} = 4 \vee \phi_1(t).opc_{mem} = 5)$. So, the proof is similar to Lemma 2. From the assumption, $I_1(\mathbf{MEM},t) = I_2(\mathbf{MEM},t)$. If the scheduling result is $\bot$, then all these fields are false since the `NOP` instruction is inserted by the pipeline control unit as a result of flush and does not induce any operations on other parts of the circuit. If $I_1(\mathbf{MEM},t) = I_2(\mathbf{MEM},t) = m$, $\sigma_1(m-1) \approx_{obs_{ag}} \sigma_2(m-1)$ because of Lemma 1 and assumptions. From Theorem 1, these fields are related to the ISA states, e.g., $\phi_1(t).mld_{mem} = (decode\_opc_{isa}\ \sigma_1(m-1) = \texttt{MLD})$ and $\phi_2(t).mld_{mem} = (decode\_opc_{isa}\ \sigma_2(m-1) = \texttt{MLD})$. Because of obs_{ag}, $decode\_opc_{isa}\ \sigma_1(m-1) = decode\_opc_{isa}\ \sigma_2(m-1)$, and thus $\phi_1(t).mld_{mem} = \phi_2(t).mld_{mem}$. Accordingly, all request fields are proved. □

Lemma 4. *For any $\phi_1 \simeq_{I_1} \sigma_1$ and $\phi_2 \simeq_{I_2} \sigma_2$, $\sigma_1 \approx_{obs_{ag}} \sigma_2$, if the programs in two ISA traces are not self modifying SC σ_1 and SC σ_2, $\forall k. I_1(k,t) = I_2(k,t)$ at the cycle t, and $\forall k.\ I(k,t+1) \neq \bot \Rightarrow I(k,t+1) \leq n$ where n is the length of ISA traces σ_1 and σ_2, and there is no jump in the pipelined circuit $\neg\phi_1(t).jmp_{ex} \wedge \neg\phi_2(t).jmp_{ex}$, then $\phi_1(t).hzd_{id} = \phi_2(t).hzd_{id}$.*

Proof. Lemma 4 has an additional assumption for no jumps in the circuit compared to the previous two lemmas, because as Table 1 shown at the cycle $t+2$, the instruction in the **ID** stage will be flushed when a jump is identified by the **EX** stage. It means the data hazards will be ignored by the pipeline under a jump at the same cycle. The proof is related to the **ID** stage where $I_1(\mathbf{ID},t) = I_2(\mathbf{ID},t)$. If the scheduling result is $\bot$ under the condition $\neg\phi_1(t).jmp_{ex} \wedge \neg\phi_2(t).jmp_{ex}$, Theorem 1 shows that the hazards flags are flushed as false, i.e., $\neg\phi_1(t).hzd_{id} \wedge \neg\phi_2(t).hzd_{id}$. Otherwise for a result m, the proof is the same as before which maps the hazard flags hzd_{id} to the ISA states and then shows the equivalence of hzd_{id} between two circuit traces at the cycle t. □

Lemma 2, 3 and 4 ensure that pipeline challenges are handled in the same way at the cycle t. Based on them, Theorem 2 shows that the two circuit traces have the same scheduling results for the next cycle.

Theorem 2. *For any $\phi_1 \simeq_{I_1} \sigma_1$ and $\phi_2 \simeq_{I_2} \sigma_2$, $\sigma_1 \approx_{obs_{ag}} \sigma_2$, if the programs in two ISA traces are not self modifying SC σ_1 and SC σ_2, $\forall k. I_1(k,t) = I_2(k,t)$ at the cycle t, and $\forall k.\ I(k,t+1) \neq \bot \Rightarrow I(k,t+1) \leq n$ where n is the length of ISA traces σ_1 and σ_2, and $\phi_1(t) \approx_f \phi_2(t)$, then $\forall k. I_1(k,t+1) = I_2(k,t+1)$.*

Proof. Theorem 2 is proved by checking every pipeline stage at the cycle $t+1$ with the help of $\approx_f$ which contains circuit fields that directly affect the scheduling results. The $\approx_f$ means control flags *ctrl* have the same value in the two circuit traces. If a pipeline stage is disabled for the cycle $t+1$, i.e. $\neg\phi_1(t).enable_k \wedge \neg\phi_2(t).enable_k$, then $I_1(k, t+1) = I_1(k, t) \wedge I_2(k, t+1) = I_2(k, t)$. Because of the assumption $\forall k.\ I_1(k, t) = I_2(k, t)$, $I_1(k, t+1) = I_2(k, t+1)$ is proved. When a stage is enabled, the following shows the explicit proof:

- **IF**:

  1. If the **EX** stage has a jump (i.e., jmp_{ex} is true at the cycle t), $\phi_1(t).jmp_{ex} \wedge \phi_2(t).jmp_{ex}$ as Lemma 2 proves. According to Definition 2, $I_1(\mathbf{IF}, t+1) = I_1(\mathbf{EX}, t)+1 \wedge I_2(\mathbf{IF}, t+1) = I_2(\mathbf{EX}, t) + 1$. Since $I_1(\mathbf{EX}, t) = I_2(\mathbf{EX}, t)$, $I_1(\mathbf{IF}, t+1) = I_2(\mathbf{IF}, t+1)$.
  2. If a jump is in the **IF** or **ID** stage, as the obs_{ag} required and the fact $I_1(\mathbf{IF}, t) = I_2(\mathbf{IF}, t) \wedge I_1(\mathbf{ID}, t) = I_2(\mathbf{ID}, t)$, the results from $decode\_opc_{isa}$ are euqal in σ_1 and σ_2. So for this case, $I_1(\mathbf{IF}, t+1) = I_2(\mathbf{IF}, t+1) = \perp$.
  3. For the regular case, $I_1(\mathbf{IF}, t+1) = I_1(\mathbf{IF}, t) + 1 \wedge I_2(\mathbf{IF}, t+1) = I_2(\mathbf{IF}, t) + 1$, and therefore $I_1(\mathbf{IF}, t+1) = I_2(\mathbf{IF}, t+1)$.

- **ID**:

  1. If jmp_{ex} is true $\phi_1(t).jmp_{ex} \wedge \phi_2(t).jmp_{ex}$, the **ID** stage is flushed at the cycle $t+1$. So $I_1(\mathbf{ID}, t+1) = I_2(\mathbf{ID}, t+1) = \perp$.
  2. If a jump is in the **ID** stage at the cycle t, then the case is the same as the second case in the **IF** stage, $I_1(\mathbf{ID}, t+1) = I_2(\mathbf{ID}, t+1) = \perp$.
  3. Normally, $I_1(\mathbf{ID}, t+1) = I_1(\mathbf{IF}, t) \wedge I_2(\mathbf{ID}, t+1) = I_2(\mathbf{IF}, t)$, and $I_1(\mathbf{IF}, t) = I_2(\mathbf{IF}, t)$. So, $I_1(\mathbf{ID}, t+1) = I_2(\mathbf{ID}, t+1)$ is proved.

- **EX**:

  1. If jmp_{ex} is true $\phi_1(t).jmp_{ex} \wedge \phi_2(t).jmp_{ex}$, the **EX** stage is flushed at the cycle $t+1$. So $I_1(\mathbf{EX}, t+1) = I_2(\mathbf{EX}, t+1) = \perp$.
  2. If there is no jump but data hazards are found at the cycle t, $\phi_1(t).hzd_{id} \wedge \phi_2(t).hzd_{id}$ holds because of Lemma 4. Therefore, the **EX** stage is flushed at the cycle $t+1$ as no valid instruction to execute, $I_1(\mathbf{EX}, t+1) = I_2(\mathbf{EX}, t+1) = \perp$.
  3. Otherwise, $I_1(\mathbf{EX}, t+1) = I_1(\mathbf{ID}, t) \wedge I_2(\mathbf{EX}, t+1) = I_2(\mathbf{ID}, t)$, and $I_1(\mathbf{ID}, t) = I_2(\mathbf{ID}, t)$. So, $I_1(\mathbf{EX}, t+1) = I_2(\mathbf{EX}, t+1)$.

- **MEM**:

  1. If there is an external request issued by the **MEM** stage (mld_{mem}, $mstr_{mem}$, $intr_{mem}$ or acc_{mem}), the request is identical in the two circuit traces as Lemma 3 shows. Then the **MEM** stage is flushed at the next cycle, $I_1(\mathbf{MEM}, t+1) = I_2(\mathbf{MEM}, t+1) = \perp$
  2. Otherwise, $I_1(\mathbf{MEM}, t+1) = I_1(\mathbf{EX}, t) \wedge I_2(\mathbf{MEM}, t+1) = I_2(\mathbf{EX}, t)$, and $I_1(\mathbf{EX}, t) = I_2(\mathbf{EX}, t)$, leading to $I_1(\mathbf{MEM}, t+1) = I_2(\mathbf{MEM}, t+1)$.

- **WB**: The **WB** stage is straightforward, $I_1(\mathbf{WB}, t+1) = I_1(\mathbf{MEM}, t) \wedge I_2(\mathbf{WB}, t+1) = I_2(\mathbf{MEM}, t)$, and $I_1(\mathbf{MEM}, t) = I_2(\mathbf{MEM}, t)$. Finally, $I_1(\mathbf{WB}, t+1) = I_2(\mathbf{WB}, t+1)$.

□

Similarly Lemma 2, 3 and 4, the following lemmas 5, 6, and 7 show the equivalence of fields handling pipeline challenges for the cycle $t+1$ respectively. The additional assumption is $\phi_1(t) \approx_f \phi_2(t)$ which allows us to apply Theorem 2 in the proof. Then, the proof is established in the same way as the previous lemmas, since the circuit behaviours are still constrained by the ISA traces at the cycle $t+1$ according to Theorem 1 and Theorem 2.

Lemma 5. *For any $\phi_1 \simeq_{I_1} \sigma_1$ and $\phi_2 \simeq_{I_2} \sigma_2$, $\sigma_1 \approx_{obs_{ag}} \sigma_2$, if the programs in two ISA traces are not self modifying SC σ_1 and SC σ_2, $\forall k.I_1(k,t) = I_2(k,t)$ at the cycle t, $\forall k.\ I(k,t+1) \neq \bot \Rightarrow I(k,t+1) \leq n$ where n is the length of ISA traces σ_1 and σ_2, and $\phi_1(t) \approx_f \phi_2(t)$, then $\phi_1(t+1).jmp_{ex} = \phi_2(t+1).jmp_{ex}$.*

Lemma 6. *For any $\phi_1 \simeq_{I_1} \sigma_1$ and $\phi_2 \simeq_{I_2} \sigma_2$, $\sigma_1 \approx_{obs_{ag}} \sigma_2$, if the programs in two ISA traces are not self modifying SC σ_1 and SC σ_2, $\forall k.I_1(k,t) = I_2(k,t)$ at the cycle t, $\forall k.\ I(k,t+1) \neq \bot \Rightarrow I(k,t+1) \leq n$ where n is the length of ISA traces σ_1 and σ_2, and $\phi_1(t) \approx_f \phi_2(t)$, then $\phi_1(t+1).mld_{mem} = \phi_2(t+1).mld_{mem} \wedge \phi_1(t+1).mstr_{mem} = \phi_2(t+1).mstr_{mem} \wedge \phi_1(t+1).intr_{mem} = \phi_2(t+1).intr_{mem} \wedge \phi_1(t+1).acc_{mem} = \phi_2(t+1).acc_{mem}$.*

Lemma 7. *For any $\phi_1 \simeq_{I_1} \sigma_1$ and $\phi_2 \simeq_{I_2} \sigma_2$, $\sigma_1 \approx_{obs_{ag}} \sigma_2$, if the programs in two ISA traces are not self modifying SC σ_1 and SC σ_2, $\forall k.I_1(k,t) = I_2(k,t)$ at the cycle t, $\forall k.\ I(k,t+1) \neq \bot \Rightarrow I(k,t+1) \leq n$ where n is the length of ISA traces σ_1 and σ_2, $\phi_1(t) \approx_f \phi_2(t)$, and there is no jump in the pipelined circuit $\neg\phi_1(t+1).jmp_{ex} \wedge \neg\phi_2(t+1).jmp_{ex}$, then $\phi_1(t+1).hzd_{id} = \phi_2(t+!).hzd_{id}$.*

Based on Lemma 5, 6, and 7, Theorem 3 shows that the two circuit traces satisfy $\approx_f$ for the next cycle $t+1$. An additional assumption EC_{ag} is needed as the environment responses affect the circuit fields in $\approx_f$.

Theorem 3. *For any $\phi_1 \simeq_{I_1} \sigma_1$ and $\phi_2 \simeq_{I_2} \sigma_2$, $\sigma_1 \approx_{obs_{ag}} \sigma_2$, $\phi_1(t) \approx_f \phi_2(t)$, if the programs in two ISA traces are not self modifying SC σ_1 and SC σ_2, $\forall k.I_1(k,t) = I_2(k,t)$ at the cycle t, and $\forall k.\ I(k,t+1) \neq \bot \Rightarrow I(k,t+1) \leq n$ where n is the length of ISA traces σ_1 and σ_2, and the two circuit traces satisfy $EC_{ag}(\phi_1, \phi_2)$, then $\phi_1(t+1) \approx_f \phi_2(t+1)$.*

Proof. Based on the assumptions, Theorem 1 and 2 hold. According to Definition 7, the proof considers the following fields for the cycle $t+1$.

- PC_g: the PC_g at the cycle $t+1$ depends on the control flag for the **IF** stage. As $\phi_1(t) \approx_f \phi_2(t)$, $\phi_1(t).enable_{if} = \phi_2(t).enable_{if}$. If the **IF** stage is disabled $\neg\phi_1(t).enable_{if} \wedge \neg\phi_2(t).enable_{if}$, then PC_g is unchanged for the cycle $t+1$ and the assumption shows that $\phi_1(t).PC_g = \phi_2(t).PC_g$. When the **IF** stage is enabled, the jmp_{ex} at the cycle t affects the PC_g. From Lemma 2, $\phi_1(t).jmp_{ex} = \phi_2(t).jmp_{ex}$. If there is a jump, then $I_1(\mathbf{EX},t) = I_2(\mathbf{EX},t) = m$ since $\bot$ cannot cause a jump as the correctness required, and $\sigma_1(m) \approx_{obs_{ag}} \sigma_2(m)$ from the obs_{ag} assumption. Because of $\sigma_1(m) \approx_{obs_{ag}} \sigma_2(m)$, $\sigma_1(m).PC = \sigma_2(m).PC$. The PC_g is the jump target address as Theorem 1 shows, i.e., $\phi_1(t+1).PC_g = \sigma_1(m).PC \wedge \phi_2(t+1).PC_g = \sigma_2(m).PC$, and thus has the same value. Otherwise no jumps, $\phi_1(t+1).PC_g = \phi_1(t).PC_g + 4 \wedge \phi_2(t+1).PC_g = \phi_2(t).PC_g + 4$. Since $\phi_1(t).PC_g = \phi_2(t).PC_g$, the proof is done.

- cmd_g: the cmd_g shows the type of memory requests when the pipeline works normally (i.e., $st_g = 0$). From the assumption $\phi_1(t) \approx_f \phi_2(t)$, we know that $\phi_1(t).st_g = \phi_2(t).st_g \wedge$

$\phi_1(t).cmd_g = \phi_2(t).cmd_g$. If the pipeline does not work at the cycle t ($st_g \neq 0$), e.g., because of waiting for the external response, then the command remains unchanged at the cycle $t+1$, $\phi_1(t+1).cmd_g = \phi_1(t).cmd_g \wedge \phi_2(t+1).cmd_g = \phi_2(t).cmd_g$. So, $\phi_1(t+1).cmd_g = \phi_2(t+1).cmd_g$. If the pipeline is working $st_g = 0$, the requests from the **MEM** stage at the cycle t are considered, and Lemma 3 guarantees that ϕ_1 and ϕ_2 have the same value for these requests. The proof is completed straightforwardly.

1. mld_{mem}: $\phi_1(t+1).cmd_g = \phi_2(t+1).cmd_g = load$.
2. $mstr_{mem}$: $\phi_1(t+1).cmd_g = \phi_2(t+1).cmd_g = store$.
3. $intr_{mem}$ and acc_{mem}: $\phi_1(t+1).cmd_g = \phi_2(t+1).cmd_g = fetch$.

- ad_g: the ad_g at the cycle $t+1$ is updated when the pipeline is in the working state $st_g = 0$ and memory load or store requests are issued at the cycle t.

  1. mld_{mem}: From the assumptions, $\phi_1(t).mld_{mem} = \phi_2(t).mld_{mem}$ and $I_1(\mathbf{MEM}, t) = I_2(\mathbf{MEM}, t)$. If the scheduling result is $\perp$, then $\neg mld_{mem}$ violates the condition. If the result is m, then Theorem 1 shows $\phi_1(t+1).ad_{mem} = mem\_ad_{isa}\ \sigma_1(m-1) \wedge \phi_2(t+1).ad_{mem} = mem\_ad_{isa}\ \sigma_2(m-1)$. As $\sigma_1(m-1) \approx_{obs_{ag}} \sigma_2(m-1)$ and obs_{ag}, $mem\_ad_{isa}\ \sigma_1(m-1) = mem\_ad_{isa}\ \sigma_2(m-1)$ is guaranteed. Therefore, $\phi_1(t+1).ad_{mem} = \phi_2(t+1).ad_{mem}$ is proved.
  2. $mstr_{mem}$: The proof is the same as the above case for mld_{mem}.

  Otherwise, the ad_g is not changed $\phi_1(t+1).ad_g = \phi_1(t).ad_g \wedge \phi_2(t+1).ad_g = \phi_2(t).ad_g$, leading to $\phi_1(t+1).ad_g = \phi_2(t+1).ad_g$.

- st_g: the st_g represents the processor's internal state to interact with the environment and the separate accelerator. The assumption EC_{ag} is used to prove its equivalence. Since the processor traces ϕ_1 and ϕ_2 guarantee the same cmd_g, PC_g, ad_g and ir_g before the cycle $t+1$, the environment traces β_1 and β_2 reply in the same way according to EC_{ag}. From the assumption $\phi_1(t).st_g = \phi_2(t).st_g$, the following cases of st_g are considered:

  1. 0: the processor is working normally at the cycle t. If the environment is not ready $\neg\beta_1(t).rdy \wedge \neg\beta_2(t).rdy$, or there is a memory or interrupt request to the environment from the processor, then $\phi_1(t+1).st_g = \phi_2(t+1).st_g = 1$ to wait for the memory's reply. If there is a acc_{mem} request, $\phi_1(t+1).st_g = \phi_2(t+1).st_g = 2$ to wait for the accelerator. Otherwise, the state is unchanged $\phi_1(t+1).st_g = \phi_2(t+1).st_g = 0$ and the proof is done.
  2. 1: the processor is waiting for the memory's rdy signal. If $\beta_1(t).rdy \wedge \beta_2(t).rdy$ and there is no interrupt request, the processor returns to the working state $\phi_1(t+1).st_g = \phi_2(t+1).st_g = 0$. If $\beta_1(t).rdy \wedge \beta_2(t).rdy$ and there is an interrupt request, then $\phi_1(t+1).st_g = \phi_2(t+1).st_g = 4$ and the processor starts to wait for the interrupt handler's reply. Otherwise not ready, the processor continues with the same state $\phi_1(t+1).st_g = \phi_2(t+1).st_g = 1$.
  3. 2: the accelerator takes a certain number of cycles to compute the result for an `ACC` instruction since it is defined internally in the $ag\pi$. From the assumption $\phi_1(t).ast_g = \phi_2(t).ast_g$, the accelerator starts and finishes an `ACC` computation at the same cycle in ϕ_1 and ϕ_2. If the computation is done, $\phi_1(t+1).st_g = \phi_2(t+1).st_g = 0$ to return the normal state. Otherwise, $\phi_1(t+1).st_g = \phi_2(t+1).st_g = 2$.

4. 3: the initial processor state waits for the memory initialization. Since $\forall t.\beta_1(t).mirdy = \beta_2(t).mirdy$ from EC_{ag}, if the initialization is done, $\phi_1(t+1).st_g = \phi_2(t+1).st_g = 0$, or $\phi_1(t+1).st_g = \phi_2(t+1).st_g = 3$ if not.

5. 4: this case is similar to $st_g = 1$. If the processor gets the acknowledgement $\beta_1(t).iack \wedge \beta_2(t).iack$, $\phi_1(t+1).st_g = \phi_2(t+1).st_g = 0$. Otherwise, $\phi_1(t+1).st_g = \phi_2(t+1).st_g = 4$.

6. others: other cases are undefined in the processor, so the states remain unmodified $\phi_1(t+1).st_g = \phi_1(t).st_g \wedge \phi_2(t+1).st_g = \phi_2(t).st_g$ and the proof is completed.

- ir_g: the ir_g is issued by the processor to the interrupt handler when the $st_g = 1$ and there is a $intr_{mem}$. From the assumption and Lemma 3, ϕ_1 and ϕ_2 have the same state and interrupt request. So when the condition happens, $\phi_1(t+1).ir_g \wedge \phi_2(t+1).ir_g$. When waiting for the handler's reply $st_g = 4$, $\neg\phi_1(t+1).ir_g \wedge \neg\phi_2(t+1).ir_g$. Otherwise, the ir_g is unchanged $\phi_1(t+1).ir_g = \phi_1(t).ir_g \wedge \phi_2(t+1).ir_g = \phi_2(t).ir_g$, so $\phi_1(t+1).ir_g = \phi_2(t+1).ir_g$.

- ast_g: the accelerator's state is updated when acc_{mem} happens. From Lemma 3, $\phi_1(t).acc_{mem} = \phi_2(t).acc_{mem}$. If acc_{mem} is true, then $\phi_1(t+1).ast_g = \phi_1(t+1).ast_g = 0$. If not, the ast_g at the cycle $t+1$ depends on ast_g at the cycle t. If $\phi_1(t).ast_g = \phi_1(t).ast_g = 0$, then $\phi_1(t+1).ast_g = \phi_1(t+1).ast_g = 1$. Otherwise, the ast_g is unchanged and thus the proof is done.

- $ctrl$: the control flags are affected by the fields handling pipeline challenges and st_g at the current cycle. The following cases are considered in the proof:

  1. $st_g \neq 0$: the above proof shows $\phi_1(t+1).st_g = \phi_1(t+1).st_g$. When the st_g is not working, the pipeline is stalled totally, and $\phi_1(t+1).ctrl = \phi_1(t+1).ctrl$.
  2. $\neg rdy$: the pipeline continually fetches new instructions but the memory may reply a $\neg rdy$. If $\neg\beta_1(t+1).rdy \wedge \neg\beta_2(t+1).rdy$, the pipeline also stalls and $\phi_1(t+1).ctrl = \phi_1(t+1).ctrl$.
  3. mld_{mem},$mstr_{mem}$, acc_{mem} and $intr_{mem}$: Lemma 6 shows that these fields have the same value in ϕ_1 and ϕ_2. If any request is issued by the **MEM** stage, the **IF**, **ID** and **EX** stages are stalled, the **MEM** stage is flushed, and the **WB** stage works as usual, $\phi_1(t+1).ctrl = \phi_1(t+1).ctrl$.
  4. jmp_{ex}: Lemma 5 shows $\phi_1(t+1).jmp_{ex} = \phi_2(t+1).jmp_{ex}$. When a jump happens, the **ID** and **EX** stages are flushed and other stages continue to work, $\phi_1(t+1).ctrl = \phi_1(t+1).ctrl$.
  5. hzd_{id}: Lemma 7 proves $\phi_1(t+1).hzd_{id} = \phi_2(t+1).hzd_{id}$. If data hazards are identified in the **ID** stage, the **IF** and **ID** stages are stalled, the **EX** stage is flushed, and **MEM** and **WB** work normally, $\phi_1(t+1).ctrl = \phi_1(t+1).ctrl$.
  6. normal: if the above cases do not happen, all pipeline stages are enabled and not flushed, and therefore the proof is completed.

□

According to Theorem 2 and 3, instructions are processed in the same way by the processor in the two circuit traces if their ISA traces are indistinguishable, as the following theorem shows.

Theorem 4. *For any $\phi_1 \simeq_{I_1} \sigma_1$ and $\phi_2 \simeq_{I_2} \sigma_2$, $\sigma_1 \approx_{obs_{ag}} \sigma_2$, if the programs in two ISA traces are not self modifying SC σ_1 and SC σ_2, and the circuit traces ϕ_1 and ϕ_2 satisfy $EC_{ag}(\phi_1, \phi_2)$, then $\forall k\ t.I_1(k,t) = I_2(k,t)$ and $\phi_1 \approx_f \phi_2$.*

Proof. Theorem 4 proved by induction on the cycle t. For the initial cycle, $I_1(k,0) = I_2(k,0)$ directly, and $\phi_1(0) \approx_f \phi_2(0)$ because of $\sim_0$ and $\approx_{obs_{ag}}$. The proof for the induction step is divided into two parts: $I_1(k,t+1) = I_2(k,t+1)$ and $\phi_1(t+1) \approx_f \phi_2(t+1)$, which are proved by Theorem 2 and 3 respectively. □

Theorem 5 shows the existence of ϕ_2 and I_2 for σ_2 when ϕ_1 is determined.

Theorem 5. *If $\phi_1 \simeq_{I_1} \sigma_1$ and SC σ_1, $\sigma_1 \approx_{obs_{ag}} \sigma_2$ and SC σ_2, then there exists a circuit trace ϕ_2 and scheduling function I_2 satisfying $\phi_2 \simeq_{I_2} \sigma_2$, and $EC_{ag}(\phi_1, \phi_2)$.*

Proof. To construct ϕ_2, we compose a processor trace α_2 produced by $ag\pi$ with the following β_2.

$$\begin{aligned}
\beta_2(0) = \langle\!| & M := \sigma_2(0).M; DI := \sigma_2(0).DI; rdy := \beta_1(0).rdy; data := \bot; inst := \bot;\\
& mirdy := \beta_1(0).mirdy; iack := \beta_1(0).iack |\!\rangle \wedge\\
\beta_2(t+1) = \langle\!| & let\ t' = lvr(\beta_1, t+1)\ in\\
& M := if\ \alpha_2(t').cmd_g = store \wedge \beta_1(t+1).rdy\\
& then\ \beta_2(t).M[\alpha_2(t').ad_g := \alpha_2(t').v_g]\ else\ \beta_2(t).M;\\
& DI := \beta_2(t).DI; rdy := \beta_1(t+1).rdy;\\
& data := if\ \alpha_2(t').cmd_g = load \wedge \beta_1(t+1).rdy\\
& then\ \beta_2(t).M[\alpha_2(t').ad_g]\ else\ \beta_2(t).data;\\
& inst := if\ \alpha_2(t').cmd_g = fetch \wedge \beta_1(t+1).rdy\\
& then\ \beta_2(t).M[\alpha_2(t').PC_g]\ else\ \beta_2(t).inst;\\
& mirdy := \beta_1(t+1).mirdy; iack := \beta_1(t+1).iack |\!\rangle
\end{aligned}$$

The *lvr* returns the cycle when the latest valid memory request happened in β_1 before the given cycle, defined as follows where *MAX_SET* is a standard HOL4 function that returns the maximal number in a set.

$$lvr(\beta, t) = MAX\_SET\{t' + 1 | \beta(t').rdy \wedge (\forall t''.t' < t'' < t \Rightarrow \neg\beta(t'').rdy)\}$$

The $\sim_0$ is fulfilled by β_2's definition for the initial cycle. The EC_{ag} is satisfied by β_2's definition too, since β_2 has the same control flow to β_1, (i.e., the same value for fields *rdy*, *mirdy*, and *iack*).

For the proof of AX, the *mem_start_env* is proved by β_2's definition for the same *mirdy* in β_1 and β_2. The *di_env* is proved by β_2's definition as DI is always unchanged in β_2. The proof is mainly about the *mem_env* and the *intr_env* is similar to *mem_env*. For the *mem_env*, we take the example for fetch, and other cases like store and load are proved in the same way. By using induction on t and $\approx_f$ from Theorem 3, the two processor traces issued the last valid fetch request at the same previous cycle t'. Because of $AX\ \phi_1$ and the same *rdy* in β_1 and β_2, we apply the response time m in β_1 to β_2 and then β_2 fulfills the fetch constraint by its definition.

Given $AX\ \phi_2$, the correctness and existence of I_2 are proved by Theorem 1. □

Based on Theorem 4 and 5, the verified Silver-Pi is CNI with respect to obs_{ag} if ISA traces in Σ are valid (i.e. satisfying the *SC*). As Section 2.3 mentioned, the circuit behaviour is undefined when executing self modifying programs.

Theorem 6. *If all ISA traces in Σ satisfy SC i.e. $\forall \sigma . \sigma \in \Sigma \Rightarrow SC\ \sigma$, then the verified Silver-Pi is $CNI(obs_{ag})$.*

References

[1] HOL development team. HOL interactive theorem prover, 2023.

[2] A. Lööw, R. Kumar, Y. K. Tan, M. O. Myreen, M. Norrish, O. Abrahamsson, and A. C. J. Fox. Verified compilation on a verified processor. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 1041–1053. ACM, 2019.

[3] A. Lööw and M. O. Myreen. A proof-producing translator for Verilog development in HOL. In *Proceedings of the 7th International Workshop on Formal Methods in Software Engineering, FormaliSE@ICSE 2019, Montreal, QC, Canada, May 27, 2019*, pages 99–108. IEEE / ACM, 2019.