An Asynchronous Register Bypass Transformation

Karl Papadantonakis

Computer Science Department
California Institute of Technology

caltechCSTR/2003.005
An Asynchronous Register Bypass Transformation

Karl Papadantonakis
Caltech Asynchronous VLSI group
6/25/2003

Acknowledgements

The idea and method of the bypass transformation comes straight from the asynchronous MIPS processor [MiniMIPS] designed by the Caltech Asynchronous VLSI group in 1998, led by Alain Martin. Mika Nyström helped recall and formulate the problem.

Introduction

After a CPU has been decomposed into top-level components such as registers and execution units, each register (or register file) is typically specified as doing a read followed by a write in each cycle: the instruction operands are read, then an operation is performed, and then the result is written. In CHP, this is specified as follows:

\[
\text{RegFile} \equiv \\
\begin{array}{c}
\ast[
\begin{array}{c}
RC?rc; \\
[rc \rightarrow RA?ra; \ RD!regs[ra] \\
\emptyset \rightarrow rc \rightarrow \text{skip}
\end{array}
\end{array}
\]

\[
WC?wc; \\
[wc \rightarrow WA?wa; \ WD?regs[wa] \\
\emptyset \rightarrow wc \rightarrow \text{skip}
\]
\]

Unfortunately, this is hard to directly implement efficiently, because it appears that the registers must be accessed in alternating read and write phases. Instead, for maximum throughput, we would like for a simultaneous read and write to be possible within a single cycle:

\[
\text{Core} \equiv \\
\ast[ (RC?rc; \ [rc \rightarrow RA?ra; \ RD!regs[ra] \ \emptyset \rightarrow rc \rightarrow \text{skip}]), \\
(WC?wc; \ [wc \rightarrow WA?wa; \ RD?regs[wa] \ \emptyset \rightarrow wc \rightarrow \text{skip})
\]

Fortunately, by introducing a simple control structure around the Core process, we can make it implement the RegFile specification. Formally, we will transform the RegFile process into the aggregation of Core and a control structure.

As an added benefit, the read-after-write latency is reduced, because a write followed by a read to the same address is handled by the control structure (and the core is not read in this case).
The Bypass Transformation

We begin with a general register file specification:

\[ RegFile \equiv \]

\[ *([RC]?rc; rc \to RA?ra; RD!regs[ra] \downarrow \to rc \to skip]; \]

\[ WC?wc; [wc \to WA?wa; WD?regs[wa] \downarrow \to wc \to skip] \]

The goal is to transform this process into a Core process which concurrently reads and writes, together with a collection of very simple processes.

The Sequential Transformation

Insert an initial-send control buffer:

\[ WC!false; \*([We!(WC?)] || *)[We?wc; [wc \to WA?wa; WD?regs[wa] \downarrow \to wc \to skip]; \]

\[ *([RC]?rc; [rc \to RA?ra; RD!regs[ra] \downarrow \to rc \to skip]; \]

\[ WC?wc; [wc \to WA?wa; WD?regs[wa] \downarrow \to wc \to skip]; \]

\[ RC?rc; [rc \to RA?ra; RD!regs[ra] \downarrow \to rc \to skip] \]

\[ WC!false \]

\[ *([We!(WC?)] || *)[We?wc; [wc \to WA?wa \downarrow \to wc \to skip]; \]

\[ RC?rc; [rc \to RA?ra \downarrow \to rc \to skip]; \]

\[ WC?wc; [wc \to WA?wa \downarrow \to wc \to skip]; \]

\[ RC?rc; [rc \to RA?ra \downarrow \to rc \to skip]; \]

\[ WC!false \]

\[ *([We!(WC?)] || *)[We?wc; [wc \to WA?wa \downarrow \to wc \to skip]; \]

\[ RC?rc; [rc \to RA?ra \downarrow \to rc \to skip]; \]

\[ WC?wc; [wc \to WA?wa \downarrow \to wc \to skip]; \]

\[ RC?rc; [rc \to RA?ra \downarrow \to rc \to skip]; \]

\[ WC!false; *([We!(WC?)] || *)[We?wc; [wc \to WA?wa \downarrow \to wc \to skip], \]

\[ RC?rc; [rc \to RA?ra \downarrow \to rc \to skip]; \]

\[ [we \to WD?regs[wa] \downarrow \to wc \to skip]; \]

\[ [rc \to RD!regs[ra] \downarrow \to rc \to skip] \]

Reshuffle the control, using slack-elasticity; this means WD must be buffered (as WC was).

\[ WC!false; *([We!(WC?)] || *)[We?wc; [wc \to WA?wa \downarrow \to wc \to skip], \]

\[ RC?rc; [rc \to RA?ra \downarrow \to rc \to skip]; \]

\[ bypass := rc \land wc \land (wa = ra); \]

\[ readNew := rc \land \neg bypass; \]

\[ [we \to WD?d \downarrow \to wc \to skip]; \]

\[ [wc \to regs[wa] := d \downarrow \to wc \to skip] ";" \]

\[ [bypass \to RD!d \]

\[ \] \]

\[ [\neg readNew \to RD!regs[ra] \]

\[ \downarrow \neg rc \to skip \]

\[ ] \]

Finally, change the ";" to a ",".

To gain performance, it may be necessary to buffer WA (as WD and WC were buffered).
Bypass Decomposition

Before we can decompose the above, we must make two copies of \( d \) (namely \( d\text{Write} \) and \( d\text{Bypass} \)), so that each copy is used only once. Also we make a variable \( d\text{Read} \), so that \( \text{regs} \) is referred to entirely within "one semicolon":

\[
\begin{align*}
\text{WC}&\text{false;} \quad *\{\text{WC!(WC?)!} \ || \\
\text{WC?}&\text{wc;} \quad [\text{wc} \rightarrow \text{WA?wa} \ \ \ \text{if wc} \rightarrow \text{skip}] , \\
\text{RC?}&\text{rc;} \quad [\text{rc} \rightarrow \text{RA?ra} \ \ \ \text{if rc} \rightarrow \text{skip}] ; \\
\text{bypass} &:= \text{rc} \land \text{wc} \land (\text{wa} = \text{ra}); \ \ \text{readNew} := \text{rc} \land \neg\text{bypass}, \ \text{justWrite} := \text{wc} \land \neg\text{bypass}; \\
\text{bypass} &\rightarrow \text{WD?d;} \ \ \text{dWrite} := \text{d} \ \ \text{dBypass} := \text{d} \\
\text{justWrite} &\rightarrow \text{WD?dWrite} \\
\text{if we} &\rightarrow \text{skip} \\
\text{readNew} &\rightarrow \text{dRead} := \text{regs[ra]} \ \ \ \text{if readNew} \rightarrow \text{skip}] ; \\
\text{bypass} &\rightarrow \text{RD!dBypass} \\
\text{readNew} &\rightarrow \text{RD!dRead} \\
\text{if we} &\rightarrow \text{skip} \\
\text{readNew} &\rightarrow \text{Core.RA[ra]} \ \ \ \text{if readNew} \rightarrow \text{skip}] \\
\end{align*}
\]

Finally, we replace each block of operations by a corresponding reactive process:

\[
\begin{align*}
\text{Compare} &\equiv \\
\text{Compare.WC}&\text{false;} \quad *\{\text{Compare.WC!(WC?)!} \ || \\
\text{Compare.WC?}&\text{wc;} \quad [\text{wc} \rightarrow \text{Compare.WA?wa;} \ \ \text{Core.WA!wa} \ \ \ \text{if wc} \rightarrow \text{skip}] , \\
\text{Compare.}&\text{RC?}&\text{rc;} \quad [\text{rc} \rightarrow \text{Compare.RA?ra;} \ \ \ \text{if rc} \rightarrow \text{skip}] ; \\
\text{bypass} &:= \text{rc} \land \text{wc} \land (\text{wa} = \text{ra}); \ \ \text{readNew} := \text{rc} \land \neg\text{bypass}, \ \text{justWrite} := \text{wc} \land \neg\text{bypass}; \\
\text{c} &:= \text{makeCommand}(\text{rc}, \text{wc}, \text{bypass}, \text{readNew}, \text{justWrite}); \\
\text{WSplit.C!c, Core.C!c, RMerge.C!c,} \\
\text{if we} &\rightarrow \text{Core.WA!wa} \ \ \ \text{if we} \rightarrow \text{skip}] , \\
\text{readNew} &\rightarrow \text{Core.RA[ra]} \ \ \ \text{if readNew} \rightarrow \text{skip}] \\
\text{WSplit} &\equiv \\
\text{WSplit.}&\text{C?}&\text{c;} \\
\text{bypass}(c) &\rightarrow \text{WSplit.WD?d;} \ \ \text{Core.D!d, RMerge.DBypass!d} \\
\text{justWrite}(c) &\rightarrow \text{Core.D!(WSplit.WD?)} \\
\text{if we}(c) &\rightarrow \text{skip} \\
\text{Core} &\equiv \\
\text{Core.}&\text{C?}&\text{c;} \\
\text{if we}(c) &\rightarrow \text{Core.D?regs[Core.WA?] \ \ \ \text{if we} \rightarrow \text{skip}] , \\
\text{readNew}(c) &\rightarrow \text{RMerge.DRead!regs[Core.RA?] \ \ \ \text{if readNew} \rightarrow \text{skip}] \\
\text{RMerge} &\equiv \\
\text{RMerge.}&\text{C?}&\text{c;} \\
\text{bypass}(c) &\rightarrow \text{RMerge.RD!(RMerge.DBypass?)} \\
\text{readNew}(c) &\rightarrow \text{RMerge.RD!(RMerge.DRead?)} \\
\text{if we}(c) &\rightarrow \text{skip} \\
\end{align*}
\]

- \( WC \equiv \text{Compare.WC}, \ WA \equiv \text{Compare.WA}, \text{ and } WD \equiv \text{WSplit.WD}. \)
- \( RC \equiv \text{Compare.RC}, \ RA \equiv \text{Compare.RA}, \text{ and } RD \text{ comes from } RMerge. \)
Block Diagram

![Diagram](image)

Figure 1. Block Diagram.

**Dead Control Elimination**

We refrain from sending any control signal which would have no effect on the receiver. This saves communication, and eliminates cases which would otherwise need to be considered by the receiver.

\[
\text{Compare} \equiv \\
\text{Compare.WC}!\text{false}; *\text{[Compare.WC(WC?)]} \mid \\
*\text{[Compare.WC?wc; [wc } \rightarrow \text{Compare.WA?wa } 0 \rightarrow \text{wc } \rightarrow \text{skip}],} \\
\text{Compare.RC?rc; [rc } \rightarrow \text{Compare.RA?ra } 0 \rightarrow \text{rc } \rightarrow \text{skip];} \\
bypass := rc \land wc \land (wa = \text{ra}); \text{readNew := rc } \land \land \text{bypass}, \text{justWrite := wc } \land \land \text{bypass;} \\
c := \text{makeCommand(rc, wc, bypass, readNew, justWrite);} \\
[wc } \rightarrow \text{WSplit.C!c } 0 \rightarrow \text{wc } \rightarrow \text{skip}], \\
[\text{wc } \lor \text{readNew } \rightarrow \text{Core.C!c } 0 \rightarrow \text{wc } \land \land \text{readNew } \rightarrow \text{skip}], \\
[rc } \rightarrow \text{RMerge.C!c } 0 \rightarrow \text{rc } \rightarrow \text{skip}], \\
[wc } \rightarrow \text{Core.WA!wa } 0 \rightarrow \text{wc } \rightarrow \text{skip}, \\
[\text{readNew } \rightarrow \text{Core.RA!ra } 0 \rightarrow \text{readNew } \rightarrow \text{skip}] \\
\]

\[
\text{WSplit} \equiv \\
*\text{[WSplit.C?c, WSplit.WD?d;} \\
\text{bypass(c ) } \rightarrow \text{Core.D!d, RMerge.DBypass!d} \\
\text{justWrite(c ) } \rightarrow \text{Core.D!d} \\
\]

\[
\text{Core} \equiv \\
*\text{[Core.C?c;} \\
\text{wc(c ) } \rightarrow \text{Core.D?regs[Core.WA?] 0 } \rightarrow \text{wc } \rightarrow \text{skip}], \\
[\text{readNew(c ) } \rightarrow \text{RMerge.DRead!regs[Core.RA?] 0 } \rightarrow \text{readNew } \rightarrow \text{skip}] \\
\]

\[
\text{RMerge} \equiv \\
*\text{[RMerge.C?c;} \\
\text{bypass(c ) } \rightarrow \text{RD!(RMerge.DBypass?)} \\
\text{readNew(c ) } \rightarrow \text{RD!(RMerge.DRead?)} \\
\]

\[
\text{RegFile} \equiv \text{Compare } \mid \mid \text{ WSplit } \mid \mid \text{ Core } \mid \mid \text{ RMerge }
\]
Bibliography