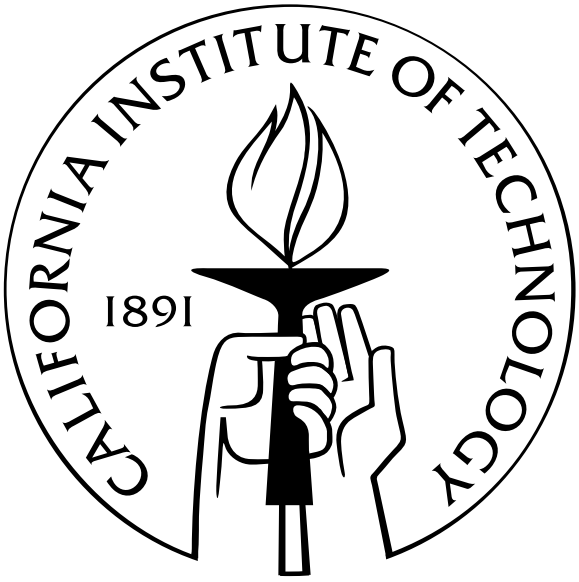

An Asynchronous Register Bypass Transformation

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:

$RegFile \equiv$

$$
\begin{array}{l}
*[\\
\quad RC?rc; \\
\quad [rc \longrightarrow RA?ra;\ \ RD!regs[ra] \\
\quad []\neg rc \longrightarrow skip \\
\quad]; \\
\\
\quad WC?wc; \\
\quad [wc \longrightarrow WA?wa;\ \ WD?regs[wa] \\
\quad []\neg wc \longrightarrow skip \\
\quad] \\
]
\end{array}
$$

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:

$Core \equiv$

$$
\begin{array}{l}
*[(RC?rc;\ \ [rc \longrightarrow RA?ra;\ \ RD!regs[ra]\ \ []\neg rc \longrightarrow skip]), \\
\quad (WC?wc;\ \ [wc \longrightarrow WA?wa;\ \ RD?regs[wa]\ \ []\neg wc \longrightarrow skip]) \\
]
\end{array}
$$

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:

$RegFile \equiv$

$$*[RC?rc;\ [rc \longrightarrow RA?ra;\ RD!regs[ra]\ []\neg rc \longrightarrow skip];$$
$$\ \ WC?wc;\ [wc \longrightarrow WA?wa;\ WD?regs[wa]\ []\neg wc \longrightarrow skip]$$
$$]$$

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:

$$WC!\mathbf{false};\ *[Wc!(WC?)]\ \ \|$$
$$\ \ Wc?wc;\ [wc \longrightarrow WA?wa;\ WD?regs[wa]\ []\neg wc \longrightarrow skip];$$
$$*[RC?rc;\ [rc \longrightarrow RA?ra;\ RD!regs[ra]\ []\neg rc \longrightarrow skip];$$
$$\ \ Wc?wc;\ [wc \longrightarrow WA?wa;\ WD?regs[wa]\ []\neg wc \longrightarrow skip]$$
$$]$$

- This is not standard CHP notation: $WC!\mathbf{false}$ in this case means channel WC is buffered and initialized with the message **false.**

Rewrite by redefining the loop body:

$$WC!\mathbf{false};\ *[Wc!(WC?)]\ \ \|$$
$$*[Wc?wc;\ [wc \longrightarrow WA?wa;\ WD?regs[wa]\ []\neg wc \longrightarrow skip];$$
$$\ \ Rc?rc;\ [rc \longrightarrow RA?ra;\ RD!regs[ra]\ []\neg rc \longrightarrow skip]$$
$$]$$

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

$$WC!\mathbf{false};\ *[Wc!(WC?)]\ \ \|$$
$$*[Wc?wc;\ [wc \longrightarrow WA?wa\ []\neg wc \longrightarrow skip],$$
$$\ \ RC?rc;\ [rc \longrightarrow RA?ra\ []\neg rc \longrightarrow skip];$$
$$\ \ [wc \longrightarrow WD?regs[wa]\ []\neg wc \longrightarrow skip];$$
$$\ \ [rc \longrightarrow RD!regs[ra]\ []\neg rc \longrightarrow skip]$$
$$]$$

Make a special case for $ra = wa$:

$$WC!\mathbf{false};\ *[Wc!(WC?)]\ \ \|$$
$$*[Wc?wc;\ [wc \longrightarrow WA?wa\ []\neg wc \longrightarrow skip],$$
$$\ \ RC?rc;\ [rc \longrightarrow RA?ra\ []\neg rc \longrightarrow skip];$$
$$\ \ bypass := rc \wedge wc \wedge (wa = ra);\ \ readNew := rc \wedge \neg bypass;$$
$$\ \ [wc \longrightarrow WD?d\ []\neg wc \longrightarrow skip];$$
$$\ \ [wc \longrightarrow regs[wa] := d\ []\neg wc \longrightarrow skip]\ \ ";"$$
$$\ \ [bypass \longrightarrow RD!d$$
$$\ \ []readNew \longrightarrow RD!regs[ra]$$
$$\ \ []\neg rc \longrightarrow skip$$
$$\ \ ]$$
$$]$$

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 $dWrite$ and $dBypass$), so that each copy is used only once. Also we make a variable $dRead$, so that $regs$ is referred to entirely within "one semicolon":

$$
\begin{array}{l}
\quad WC!\mathbf{false};\ *[\,Wc!(WC?)\,]\ \ \|\\
*[\,Wc?wc;\ \ [wc \longrightarrow WA?wa\ \ []\neg wc \longrightarrow skip],\\
\quad RC?rc;\ \ [rc \longrightarrow RA?ra\ \ []\neg rc \longrightarrow skip];\\
\quad bypass := rc \wedge wc \wedge (wa = ra);\ \ readNew := rc \wedge \neg bypass,\ \ justWrite := wc \wedge \neg bypass;\\
\\
\quad [bypass \longrightarrow WD?d;\ \ dWrite := d;\ \ dBypass := d\\
\quad []justWrite \longrightarrow WD?dWrite\\
\quad []\neg wc \longrightarrow skip\\
\quad];\\
\\
\quad [wc \longrightarrow regs[wa] := dWrite\ \ []\neg wc \longrightarrow skip],\\
\quad [readNew \longrightarrow dRead := regs[ra]\ \ []\neg readNew \longrightarrow skip];\\
\\
\quad [bypass \longrightarrow RD!dBypass\\
\quad []readNew \longrightarrow RD!dRead\\
\quad []\neg rc \longrightarrow skip\\
\ ]]
\end{array}
$$

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

$Compare \equiv$

$$
\begin{array}{l}
\quad Compare.WC!\mathbf{false};\ *[Compare.WC!(WC?)]\ \ \|\\
*[Compare.WC?wc;\ \ [wc \longrightarrow Compare.WA?wa;\ \ Core.WA!wa\ \ []\neg wc \longrightarrow skip],\\
\quad Compare.RC?rc;\ \ [rc \longrightarrow Compare.RA?ra\ \ []\neg rc \longrightarrow skip];\\
\quad bypass := rc \wedge wc \wedge (wa = ra);\ \ readNew := rc \wedge \neg bypass,\ \ justWrite := wc \wedge \neg bypass;\\
\quad c := makeCommand(rc, wc, bypass, readNew, justWrite);\\
\quad WSplit.C!c,\ \ Core.C!c,\ \ RMerge.C!c,\\
\quad [wc \longrightarrow Core.WA!wa\ \ []\neg wc \longrightarrow skip],\\
\quad [readNew \longrightarrow Core.RA!ra\ \ []\neg readNew \longrightarrow skip]\\
\ ]
\end{array}
$$

$WSplit \equiv$

$$
\begin{array}{l}
*[\,WSplit.C?c;\\
\quad [bypass(c) \longrightarrow WSplit.WD?d;\ \ Core.D!d,\ \ RMerge.DBypass!d\\
\quad []justWrite(c) \longrightarrow Core.D!(WSplit.WD?)\\
\quad []\neg wc(c) \longrightarrow skip\\
\quad]
\end{array}
$$

$Core \equiv$

$$
\begin{array}{l}
*[\,Core.C?c;\\
\quad [wc(c) \longrightarrow Core.D?regs[Core.WA?]\ \ []\neg wc \longrightarrow skip],\\
\quad [readNew(c) \longrightarrow RMerge.DRead!regs[Core.RA?]\ \ []\neg readNew \longrightarrow skip]\\
\quad]
\end{array}
$$

$RMerge \equiv$

$$
\begin{array}{l}
*[RMerge.C?c;\\
\quad [bypass(c) \longrightarrow RD!(RMerge.DBypass?)\\
\quad []readNew(c) \longrightarrow RD!(RMerge.DRead?)\\
\quad []\neg rc \longrightarrow skip\\
\ ]]
\end{array}
$$

- $WC \equiv Compare.WC$, $WA \equiv Compare.WA$, and $WD \equiv WSplit.WD$.
- $RC \equiv Compare.RC$, $RA \equiv Compare.RA$, and RD comes from $RMerge$.

Block Diagram

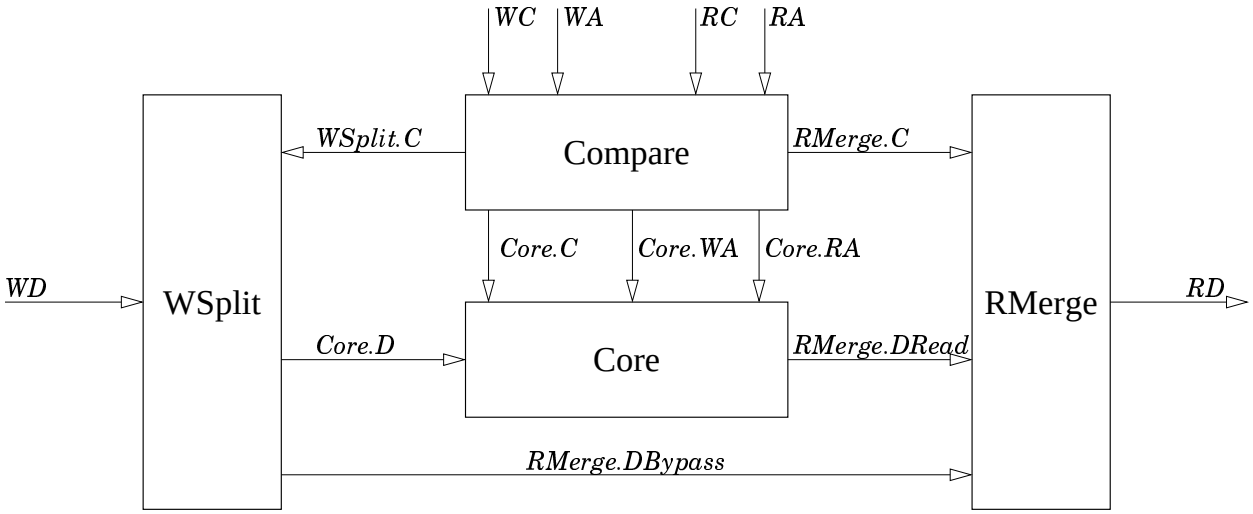

Figure 1. Block Diagram.

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.

$Compare \equiv$

$Compare.WC!\mathbf{false};\ *[Compare.WC!(WC?)]\ \|$
$*[Compare.WC?wc;\ [wc \longrightarrow Compare.WA?wa\ []\neg wc \longrightarrow skip],$
$Compare.RC?rc;\ [rc \longrightarrow Compare.RA?ra\ []\neg rc \longrightarrow skip];$
$bypass := rc \wedge wc \wedge (wa = ra);\ readNew := rc \wedge \neg bypass,\ justWrite := wc \wedge \neg bypass;$
$c := makeCommand(rc, wc, bypass, readNew, justWrite);$
$[wc \longrightarrow WSplit.C!c\ []\neg wc \longrightarrow skip],$
$[wc \vee readNew \longrightarrow Core.C!c\ []\neg wc \wedge \neg readNew \longrightarrow skip],$
$[rc \longrightarrow RMerge.C!c\ []\neg rc \longrightarrow skip],$
$[wc \longrightarrow Core.WA!wa\ []\neg wc \longrightarrow skip],$
$[readNew \longrightarrow Core.RA!ra\ []\neg readNew \longrightarrow skip]$
$]$

$WSplit \equiv$

$*[WSplit.C?c,\ WSplit.WD?d;$
$[bypass(c) \longrightarrow Core.D!d,\ RMerge.DBypass!d$
$[]justWrite(c) \longrightarrow Core.D!d$
$]$

$Core \equiv$

$*[Core.C?c;$
$[wc(c) \longrightarrow Core.D?regs[Core.WA?]\ []\neg wc \longrightarrow skip],$
$[readNew(c) \longrightarrow RMerge.DRead!regs[Core.RA?]\ []\neg readNew \longrightarrow skip]$
$]$

$RMerge \equiv$

$*[RMerge.C?c;$
$[bypass(c) \longrightarrow RD!(RMerge.DBypass?)$
$[]readNew(c) \longrightarrow RD!(RMerge.DRead?)$
$]]$

$RegFile \equiv\ Compare\ \|\ WSplit\ \|\ Core\ \|\ RMerge$

Bibliography

[MiniMIPS] **The Design of an Asynchronous MIPS R3000 Microprocessor.** Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nystrm, Paul Penzes, Robert Southworth, Uri Cummings and Tak Kwan Lee. *Proc. 17th Conference on Advanced Research in VLSI, 164-181, IEEE Computer Society Press, 1997.*