Tom Dodson
Ryne Hunt
EE480 Project 2

This document is split into 6 parts, describing the five pipeline stages, and the control logic. For the purposes of this document, a stall in a certain pipeline stage is defined as the prevention of loading that stage's terminal buffers. That is, a stall in IF prevents the value currently in IF from being loaded into the IF/ID buffers. A squash in a certain stage is defined as the zeroing of the terminal buffer outputs for that stage. That is, a squash in IF does not affect the loading of the IF/ID buffers, but prevents the values in the buffers from reaching ID.

Overview

Features of this processor include value forwarding and early evaluation of branches and jumps. Features not included, but normally found in this kind of pipeline, are a fast register file (the need for which is eliminated by value forwarding), and any concept of exceptions or interrupts.

Overall, design decisions in this project were made with the goal of minimizing stalls, not maximizing clock rate. For example, early evaluation of branches and jumps adds significant path delay to the ID stage, as the data from the register file must pass back through ID.

Other than the ID stage, the longest path is likely the evaluation of the ONES instruction in EX. It seems likely that the synthesis tools are not optimizing the design, and are merely producing serial 1-bit adders.

Instruction Fetch (IF)

The instruction fetch stage is responsible for fetching a single word from code memory each cycle. This stage has no concept of the meaning of the instructions, but supports several control signals (stall, pc_ld, and reset).

Stall must be given whenever the IF stage is to stall: because IF always increments its program counter by one, if the IF stage is to be stalled, the program counter must also be locked so that instructions are not skipped.

Pc_load is a signal to load an external address into the PC, this is used for control instructions.

Reset is provided to set the PC to zero at initialization.

One optimization that could be included would be to evaluate JMP and JAL instructions in IF. Since the decode logic is so simplistic, it would add minimal delay, and eliminate stalls entirely on that instruction type. However, ID would still have to decode the JAL instruction in order to write the PC back to the register file.

Instruction Decode (ID)

Responsible for translating a data word into a set of control signals, and forwarding those signals to the pipeline control, as well as feeding them with their associated register data into the pipeline. A decision was made to make the opcodes as simple as possible, with two forms: one for those containing two registers, and one for those containing an 8-bit immediate value (of which LIW is a special case). This made the decode stage not only simple and massively parallel, but also extremely lazy. Most control signals can be generated with >5 logic gates, due to the partial one-hot encoding scheme used to differentiate instruction types within each opcode format.

This module has a very long path, first rs_sel and rd_sel must leave the stage, then rd and rs must reenter it, to which additional delay is added by value forwarding, then the data from rd must be checked to determine if a branch is taken. This does, however, introduce only one pipeline bubble on branches and jumps instead of two in the standard MIPS pipeline.

This stage generates several signals as inputs to control: is_control, for a control instruction (branch or jump), is_taken_control (taken branch or jump), and is_extended, when the opcode is the first word of an extended 32 bit instruction. Since this stage is completely combinatorial, and takes no inputs from the controller, it must rely on the control unit to properly manipulate the interstage buffers to store extended instructions, as well as to ignore any control signals generated by the second word of such an instructions. Also generates a target address, which is loaded into the PC when is_taken_control is high.

Execute (EX)

Performs binary arithmetic, also completely combinatorial. Supports addition, subtraction, logical and, or, and exclusive or, as well as a population count. The arithmetic and logical operators are straightforward; however, population count is not. It was decided that rather than attempting to implement an adder tree, a simple summation of each bit (x[0] + x[1] + … + x[15]) would give the synthesis tools the largest chance of optimizing the design. Whether this is true or not remains to be seen.

Memory (MEM)

Responsible for loads and stores from data memory. Because there are no indexed load/store instructions, EX and MEM could be parallel, however, in order to do so correctly, some manner of reorder buffer or scoreboarding would have to be implemented, further complicating the design. In fact, in order to see any speedup, IF and ID would have to be expanded in order to supply (possibly) two operands per cycle, and WB would have to be prepared to write 2 results per cycle to memory. Therefore, this stage is extremely simple: address, data in, and data out lines.

Writeback (WB)

Interface for writing to the register file. Completes in one cycle (i.e. no fast writes, wide interfaces, DDR transfers, etc). This would cause more hazards than in the typical 5-stage pipeline; however, those hazards were merely a subset of those eliminated by using value forwarding, which will be detailed in the control section.

Control

There are two control units in the pipeline: the flow control unit, which controls how instructions flow through the pipeline, and the value forwarding unit, which controls how and when data is forwarded between pipeline stages to eliminate hazards. The advantage of this design is that the VFU can signal the flow control unit whenever a hazard would cause a stall, and the flow control unit can then overlap stalls. For example, when the VFU would cause a stall in EX, the flow control unit can choose to allow IF and ID to continue processing a control instruction.

The state diagrams for the flow control unit follow. There are no state diagrams for the value forwarding unit, as it is completely combinatorial. It works by using the register select lines in each stage to generate lists of instructions which need values from other stages. It can then signal for those values to be forwarded, or signal for a stall when, for example, an ALU instruction needs a value from a load instruction which has not yet completed.

There are actually three state machines in the flow control module, though two of them are extremely simplistic, that is, they only have two states, and current state is not a factor in determining next state. These two machines are used to produce squash signals from the VFU stall signals. It was decided to use these 1-bit state machines rather than add additional states to the control state machine because adding those states to the next state logic would have added a huge number of state transitions, as those operations are essentially parallel to much of the flow control logic.

In addition, part of the pipeline control logic, specifically, all of the logic related to stalls, is combinatorial because a stall must be asserted before the positive edge of the clock.

```
assign stall_id = vfu_stall_at_ex;
assign stall_if = (vfu_stall_at_ex & ~id_is_control) |
                  vfu_stall_at_id;
```

The following code sequence illustrates the properties of overlapping stalls:

```
        LIB $3, #0xFF
start:  LIB $2,#0
        ST $2,[$0]
        LD $2,[$0]
        ADD $2, $0
        BNEG $3,start
        HALT
```

A block diagram of the processor is provided below, with value forwarding lines excluded as the Forwarding Unit has inputs from every pipeline buffer, and outputs to multiplexers on the RD and RS lines entering ID, EX, and MEM. Reset is also excluded for the same reason.

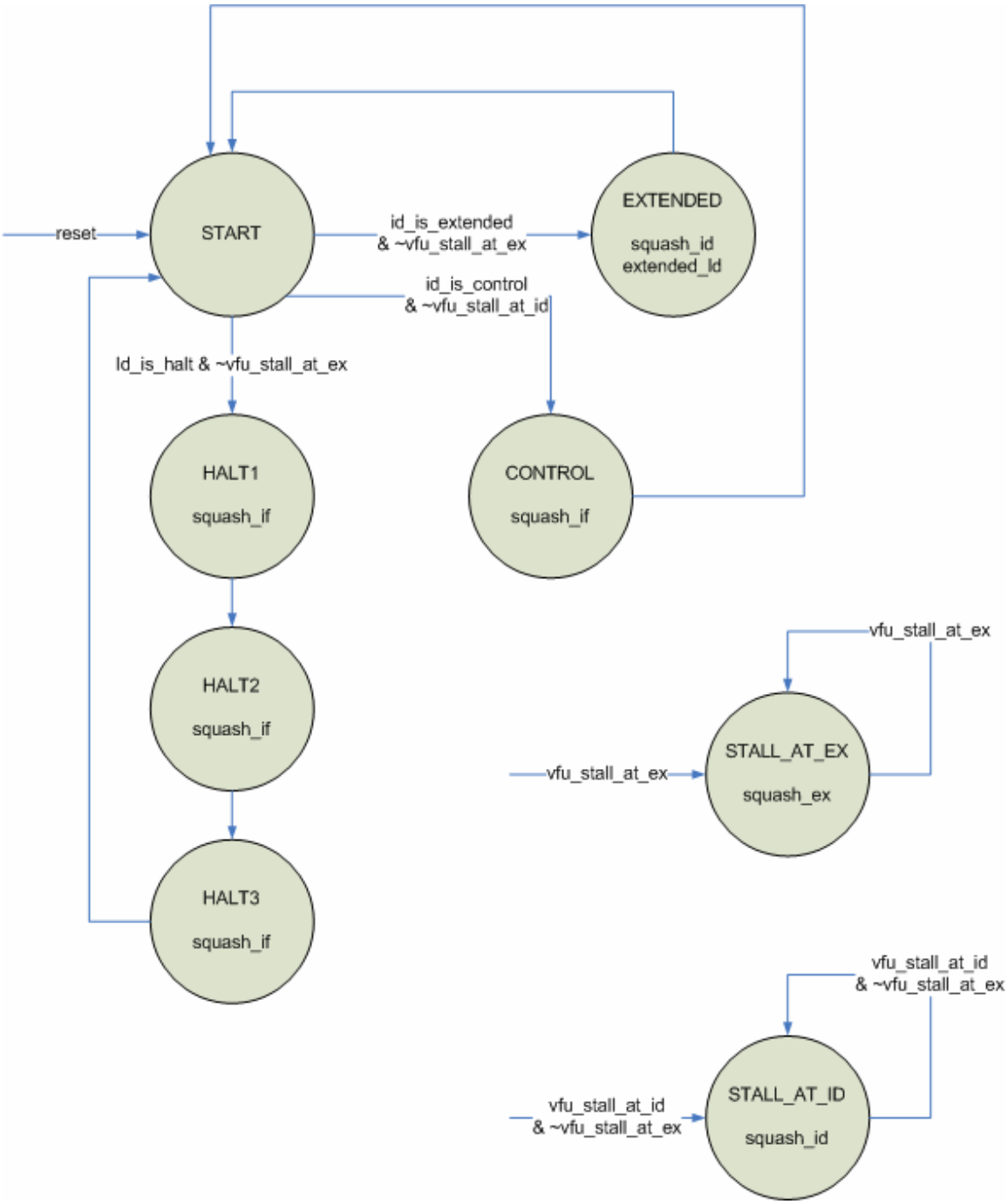

Fig 1. Pipeline Control state machine

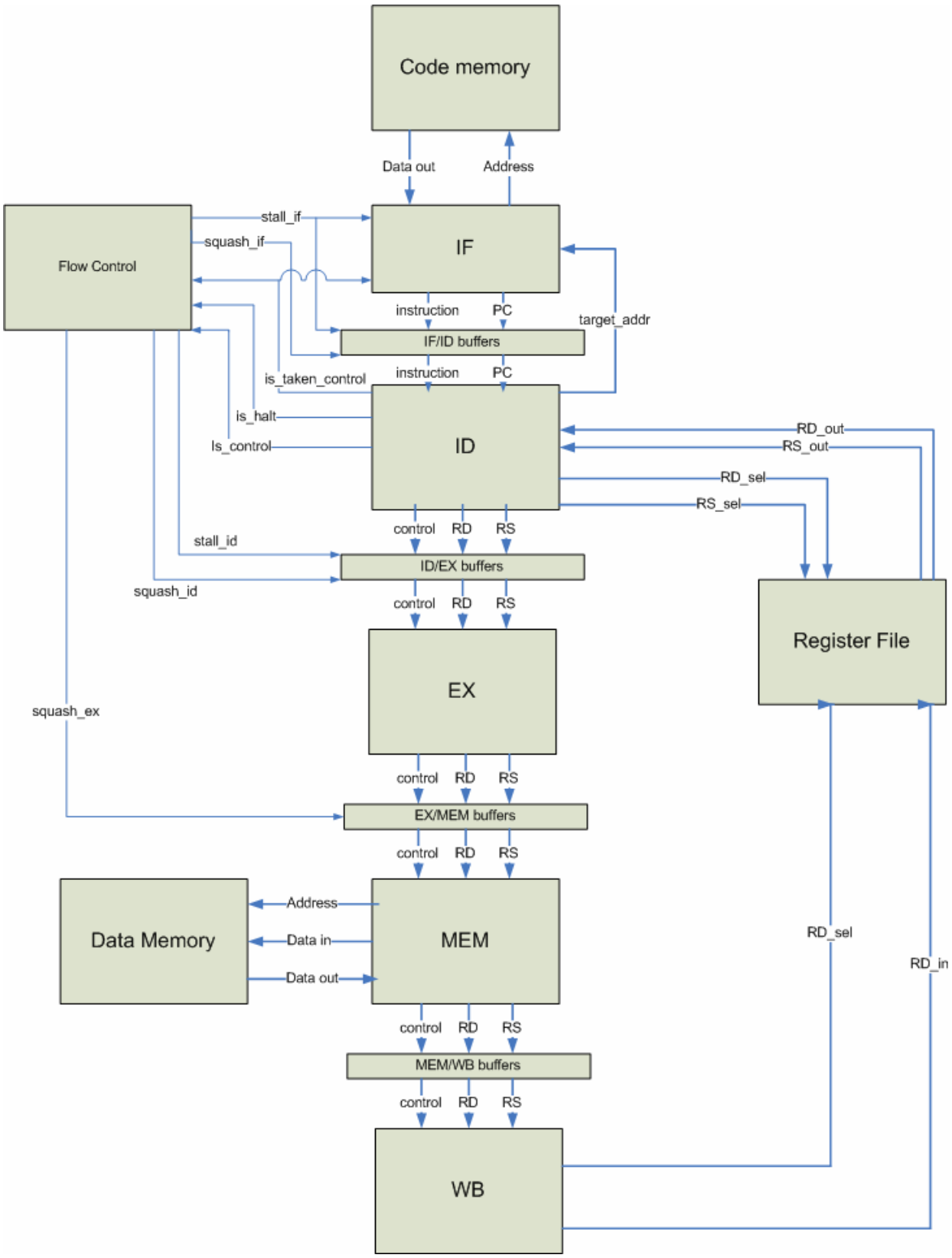

Fig 2. Block diagram, sans reset and value forwarding