Alex Gangwish and Jason Glass
EECS 361 Final Project
12/8/2016

Pipeline Processor Project

INTRODUCTION

In this assignment, we were tasked with creating a pipeline processor implementation, building upon our previous single-cycle processor. To do this, we needed to implement forwarding, hazard detection with stalling, and branching functionality. We also needed to modify our control.

CONTROL

To control our processor, we added room for control signals in our pipeline registers. For example, the ID/EX register includes signals for the WB, M, and EX controls; the EX/MEM register includes signals for the WB and M controls, and the MEM/WB register includes the signals for the WB controls. Once an instruction is decoded, the appropriate control signals are created and passed through the pipeline with their associated data. Table 1 shows the relevant control signals used.

	opcode hex	function	binary opcode	function binary	RegDst	ALUSrc	MemtoReg	RegWrite	MemWrite	Branch	Jump	ExtOp	ALUop <5:0>
add	0	20	000000	100000	1	0	0	1	0	0	0	x	100000
addu	0	21	000000	100001	1	0	0	1	0	0	0	x	100001
sub	0	22	000000	100010	1	0	0	1	0	0	0	x	100010
subu	0	23	000000	100011	1	0	0	1	0	0	0	x	100011
and	0	24	000000	100100	1	0	0	1	0	0	0	x	100100
or	0	25	000000	100101	1	0	0	1	0	0	0	x	100101
slt	0	2a	000000	101010	1	0	0	1	0	0	0	x	101010
sltu	0	2b	000000	101011	1	0	0	1	0	0	0	x	101011
sll	0	0	000000	000000	1	0	0	1	0	0	0	x	000000
lw	23		100011		0	1	1	1	0	0	0	1	100000
sw	2b		101011		x	1	x	0	1	0	0	1	100000
beq	4		000100		x	0	x	0	0	1	0	x	100010
bne	5		000101		x	0	x	0	0	1	0	x	100010
bgtz	7		000111		x	0	x	0	0	1	0	x	100010
addi	8		001000		0	1	0	1	0	1	0	1	100000

Table 1: Control Signals and Opcodes

Our pipeline control was effectively the same as our single cycle control except for the propagation of an instruction's corresponding control signals in the pipelined processor.

FORWARDING

We created a forwarding unit to determine when forwarding is necessarythat looks at the Rs and Rt fields of the instruction currently in the ID/EX register, and the Rd fields of the instructions currently in the EX/MEM and MEM/WB registers, and produces two two-digit opcodes for determining ALU input. It then attempts to determine if the ID/EX Rs or Rt registers are the same registers used in the EX/MEM or MEM/WB Rd register. If so, forwarding is necessary. If the forwarding unit determines there is a hazard between ID/EX and EX/MEM, it returns “10”, and if it finds a hazard between ID/EX and MEM/WB it returns “01.” If no hazard is detected, it returns “00.” If the forwarding unit determines there are hazards in both cases, it defaults to forwarding the information from EX/MEM (the most recent execution). The unit returns forwardA and forwardB opcodes, relating to the first and second inputs to the ALU, respectively.

The opcodes returned by the forwarding units are used in combination with two muxes to determine the appropriate inputs to the ALU. If the opcode is “10” then the ALU result currently stored in EX/MEM is input to the ALU; if the opcode is “01” then the ALU input is forwarded from MEM/WB; if the opcode is “00” the ALU input comes from the program register file. There is one additional mux used for the second input to the ALU--this is for choosing between the value selected by the forwarding logic and the sign-extended immediate field from the current instruction.

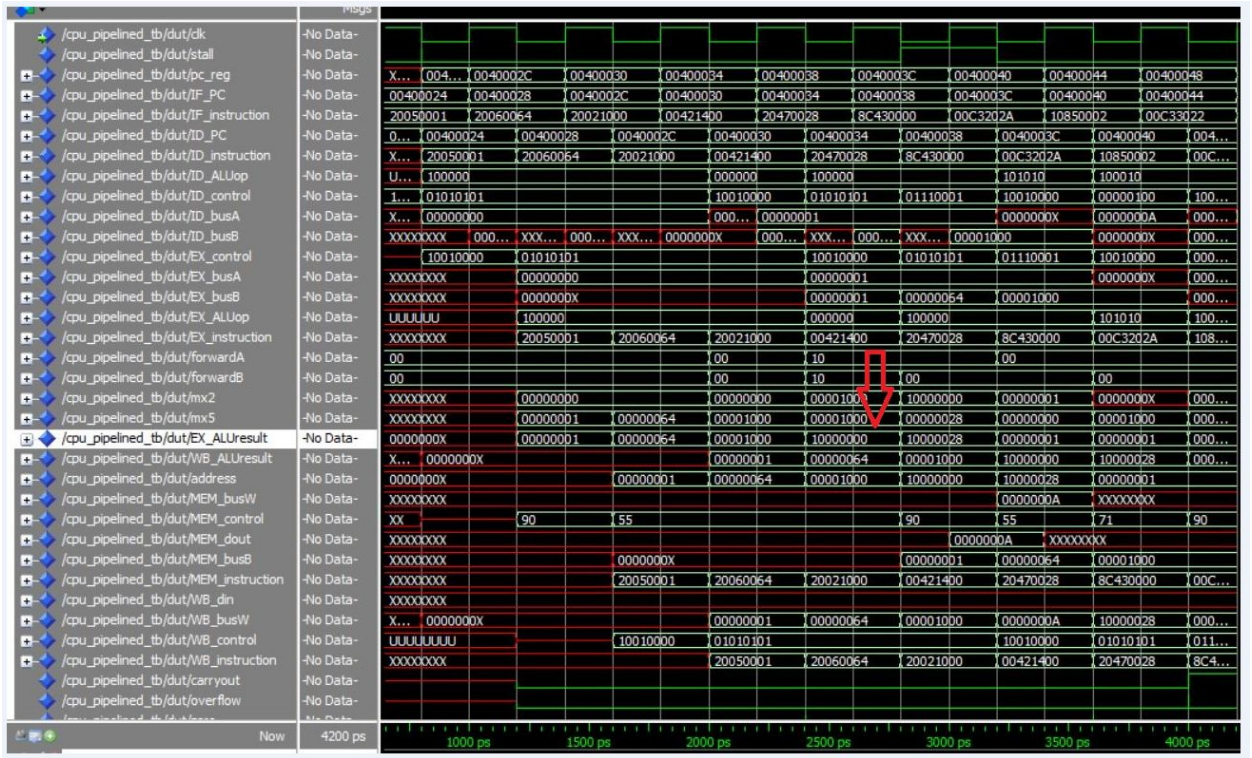

Figure 1: Trace showing data from the ALU output being forwarded to the ALU input in the next cycle

STALLING

To implement stalling, we created a stall unit that determines if a stall should happen based on the instructions. If the unit indicates a stall is necessary, the next address logic unit submits the same address again instead of incrementing to the next address, and the control signals are set to 0 to create a bubble in our pipeline.

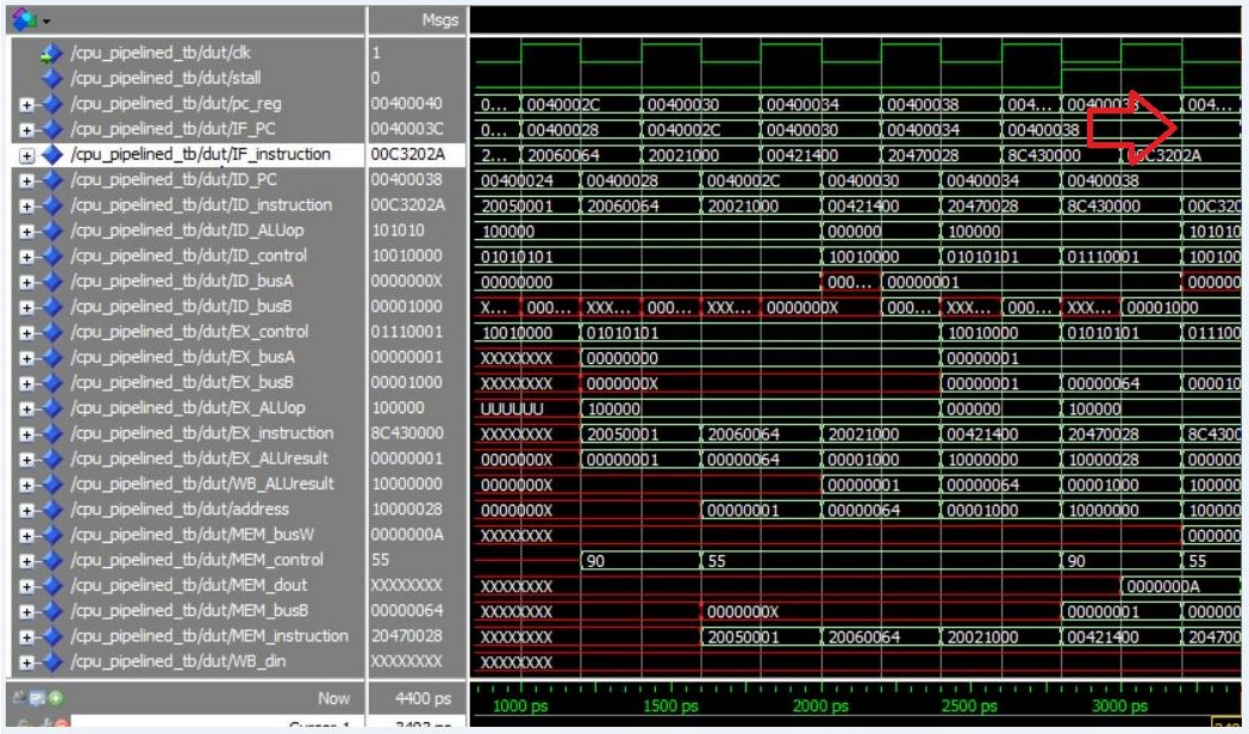

Figure 2: Trace showing the PC holding an instruction in place for a stall

The red arrow denotes where IF_PC, the PC signal currently being processed in the IFetch phase, has been stalled and the same value is available for two consecutive falling edges.

Our stall unit works by checking the instruction in the Decode phase for whether or not it is a branch or ALU instruction, our forwarding implementation and branch implementation makes loads and branches have only a 1 cycle delay at worst.

If the instruction is a branch or jump then the PC in the fetch phase is held and benign controls are set.

BRANCHING

Our branching logic is based off of having placed the Next Address Logic (NAL) unit during the decode phase instead of after the execution phase. This along with our forwarding implementation allows us to limit the worst case delay of a branch to 1 cycle. After that the next PC is set and on the following falling edge the PC is updated and the branch will be done if it executed at all. Our NAL unit simply outputs PC+4 if a branch is not taken. The NAL implementation is the same as our single cycle processor and uses muxs and control signals taken out of the instructions to detect branches.

Figure 3 shows a successful branch where PC_reg (the next PC from the NAL) is updated from x00400050 to x0040038, which is -24 bytes, 6 instructions. This is the correct calculation done during Bills Branch program. The offset in the BNE instruction is -7 which +1 is -6.

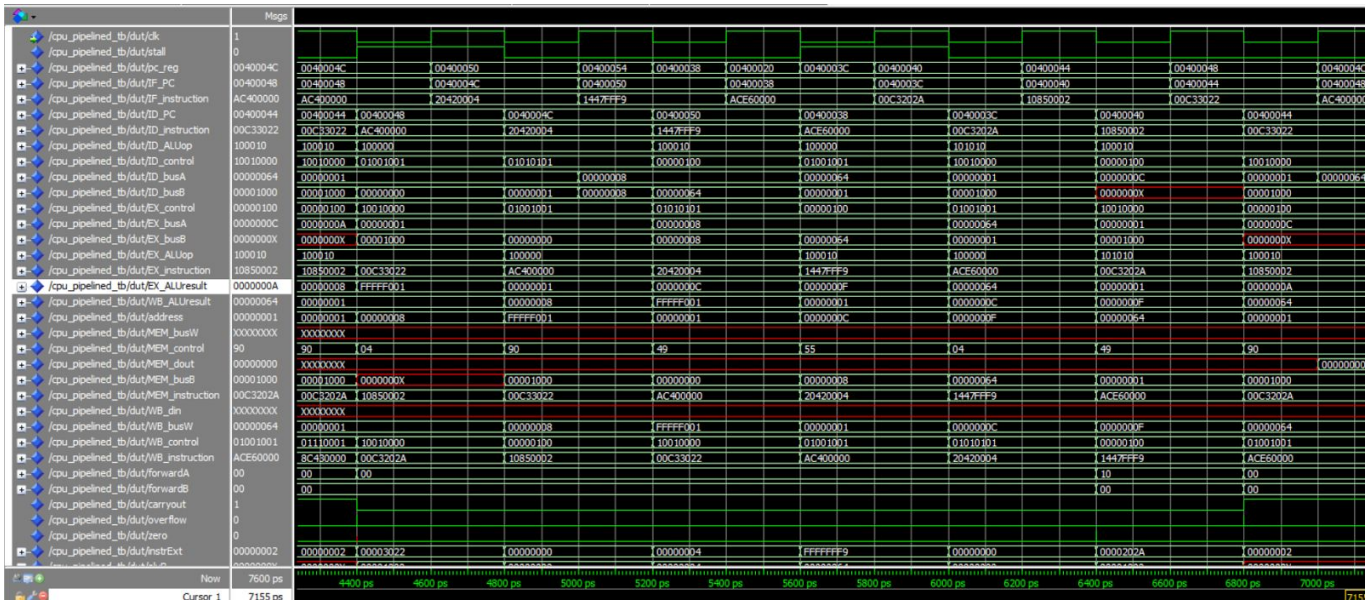

Figure 3: Trace showing successful execution of branch instructions

DESIGN CHALLENGES AND LIMITATIONS

Stalling

Our design does not seem to correctly implement stalling. It seemed that implementing our fully design stall detection was interfering with the pipeline for load instructions and corrupting the

data stream. We believe this was due to how the bubbles in the pipeline were implemented and a failure to control runaway write-enable signals. With more time we believe we could have solved the problems and fully implemented stalling.

Branching

Our branching logic was also at times broken by a data hazard created by our stall on load instruction. Improper writes caused branches to be taken and not taken incorrectly. However we did get the pipeline to successfully jump on multiple occasions and scenarios and this was a success in itself given the number of moving parts in a pipelined processor.

Forwarding

We had some trouble with getting our forwarding unit to work at first, specifically during the first few cycles of a program. We determined that the issue was that the logic in the forwarding unit relies on reading the Rd registers utilized in the instructions in the EX/MEM and MEM/WB registers, and these registers initialize to don't-cares, which seemed to produce undesirable results for our forwarding logic. We fixed this issue by temporarily stalling our forwarding unit until after the pipeline registers have had time to be written to. Another possible solution would be to have the registers automatically initialized to all 0s, instead of don't-cares.

TIME TRACES

Bills Branch

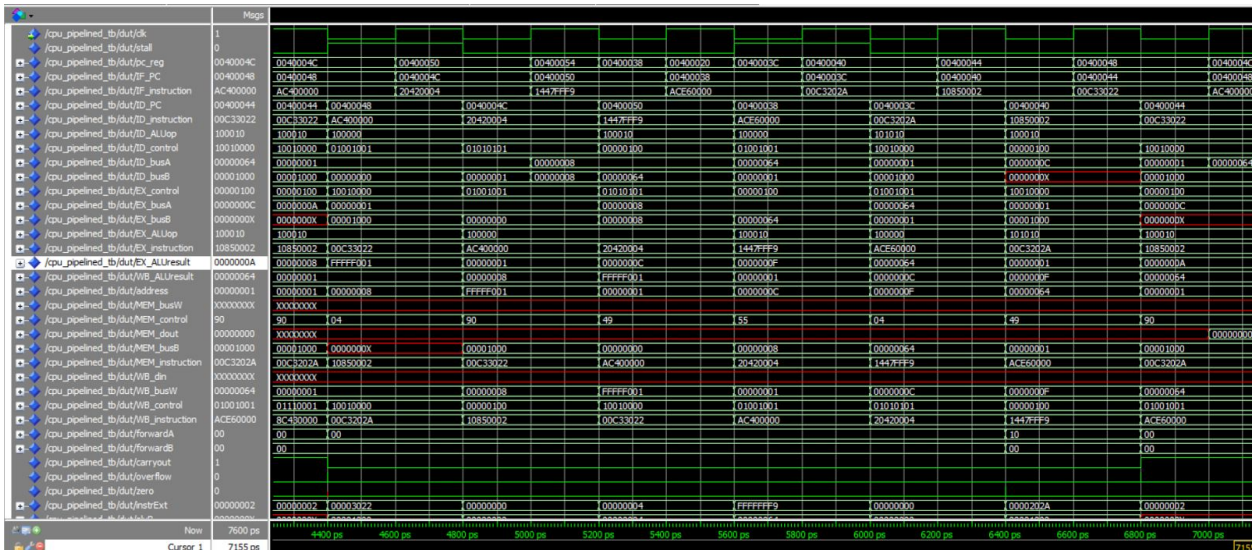

Figure 4: Trace for bills_branch.dat

Unsigned Sum

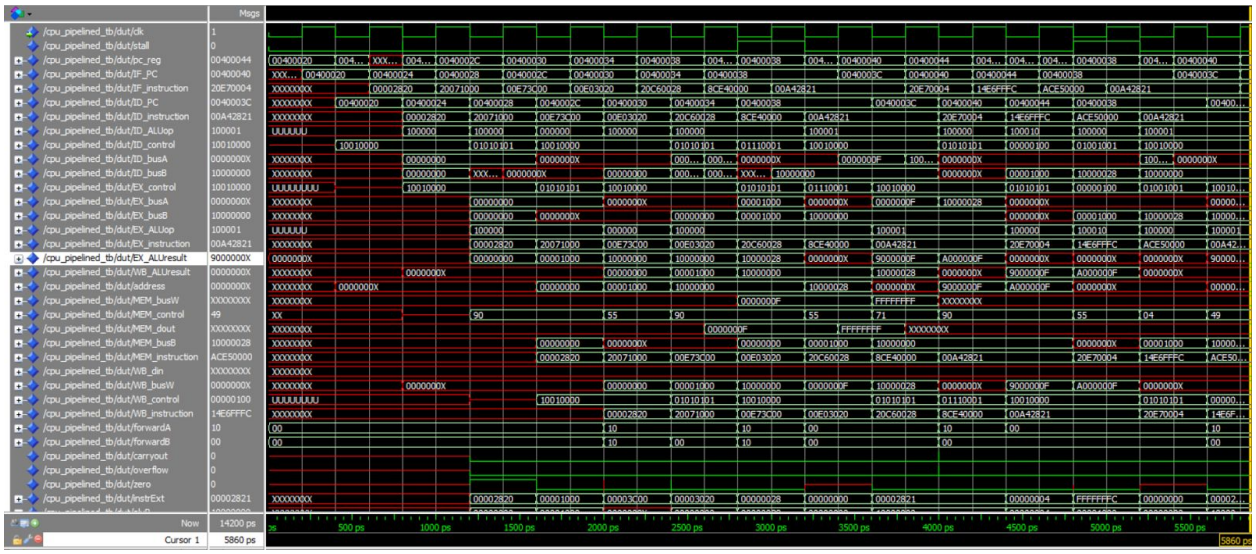

Figure 5: Trace for unsigned_sum.dat

Sort Corrected

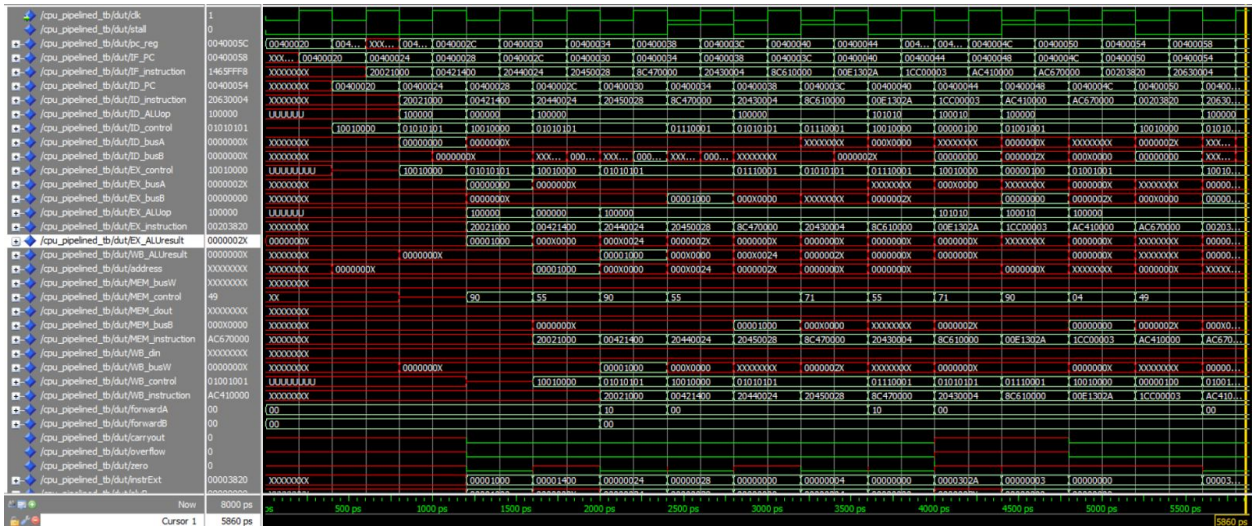

Figure 6: Trace for sort_corrected_branch.dat