Andy Hu
Kevin Shih

CS150 Project Design Document

Pipeline Stage Design

Where are your pipeline stages defined and why did you put them there?
1) Instruction Memory 2) RegFile and ALU 3) Data Memory and WriteBack
This configuration minimizes the average number of stalls we need to perform.
It also simplifies forwarding.
What happens in each pipeline stage?
1) Instruction Memory: Fetch instruction from memory
2) RegFile and ALU: Fetch data from regfile, select source for ALU
3) Data Memory and WriteBack: Load/Store from UART, Load/Store from DMEM, Write result back to RegFile.

Hazards

What kinds of hazards do you have to take care of?
Data Hazards and Control Hazards (branch delay slot)
Show a sequence of instructions that demonstrate each hazard and explain why it causes an error.
ADD $S0, …
AND $T0, $S0, ...
OR $T1, ...
$T0 gets old value of $S0

LW $S0, ...
AND $T0, $S0,...
OR $T1, $S0,...
SUB $T2, $S0,...
$T0 gets old value of $S0

24 BEQ ...
28 AND ...
32 ...
…
64 …
We don't know which address to branch to until the third stage.
What hardware in your design specifically deals with these hazards?
Data Hazards: Hazard Control Block (stalling)
Control Hazards: Forwarding branching results and new addresses

Under what circumstances would you need to forward data and between which stages?
Data dependency from ALUop to next instruction: Forward from Data/WriteBack stage to ALU input.
Branching: Forward result of comparison from beginning of Data/WriteBack stage to control unit.
Jump and Link: Forward from Data/WriteBack stage into register file.

Memory Map

How did you implement the memory map?
When loading/storing data, we check the first four bits of the target address. Then we load/store to the corresponding area (Data Memory, Instruction Memory, or I/O.
What hardware in your design is specifically used for with the memory map?
The memory map is specifically used for the UART.
Briefly explain how to operate the UART in order to send a byte over the serial line in the context of memory mapped I/O.
To send a byte sw,sh, or sb into memory location 0x80000008. The CPU will wait for 0x80000000 (DataInReady) to go high. Then it will send the byte to the UART.
To receive a byte, lw,lh, or lb from memory location 0x8000000c. The CPU will wait for 0x80000004 (DataOutValid) to go high. Then it will read the byte from the UART.

Critical Path Analysis

The critical path goes through the UART (sw,lw, etc), since the UART can stall the CPU for several clock cycles if it is receiving/transmitting data.
What modules do you expect to be the slowest in your design (list the slowest 3)?
Briefly justify your answer.
UART - have to wait several clock cycles to receive/transmit each bit
Data Memory - Memory Access is slow
Instruction Memory - Memory Access is slow

Stalling

How does your design implement stalling?
Our design disables the program counter, both pipeline registers, instruction memory, and data memory.
Under what circumstances would you want to stall the CPU?
Branches/Jump, Load/Store Words, Waiting for UART

Design Document

You will be required to submit a design document for review for this checkpoint that addresses the following aspects of your proposed design. This should not exceed more than 2 pages and should be written clearly and concisely.

Branch Handling
- bypass the ALU to reduce critical path
- check JAL handling top nibble is preserved and 2 LSB are zeroed
- PC+4 and PC+8 computations are handled for appropriate instructions
- BLTZ, BGEZ bit flag differentiation - they have the same opcode and are differentiated only by a bit in the instruction
- branch control logic unit is in design and datapath computes addresses for all J type destinations

Loading and Storing
- the write mask should be implemented correctly (endianness for block RAM is reverse of verilog indexing)
- a controller computes write bits based on instruction
- your memory map implementation is correct
- write enable bits are correctly supressed for appropriate accesses
- byte selection logic and sign extension are implemented for reads from memory map

Forwarding Data
- forwarding controller correctly computes forwarding for both register stage and execution stage (design dependent but both are most likely needed)
- datapath mutiplexors for forwarding address both the A and B src

Register File
- zero register is non-writable and should always output zero
- no negative edge write back logic

Stalling
- must stall on positive and implementation cannot gate clock

Pipelining
- you must actually have 3 pipeline stages

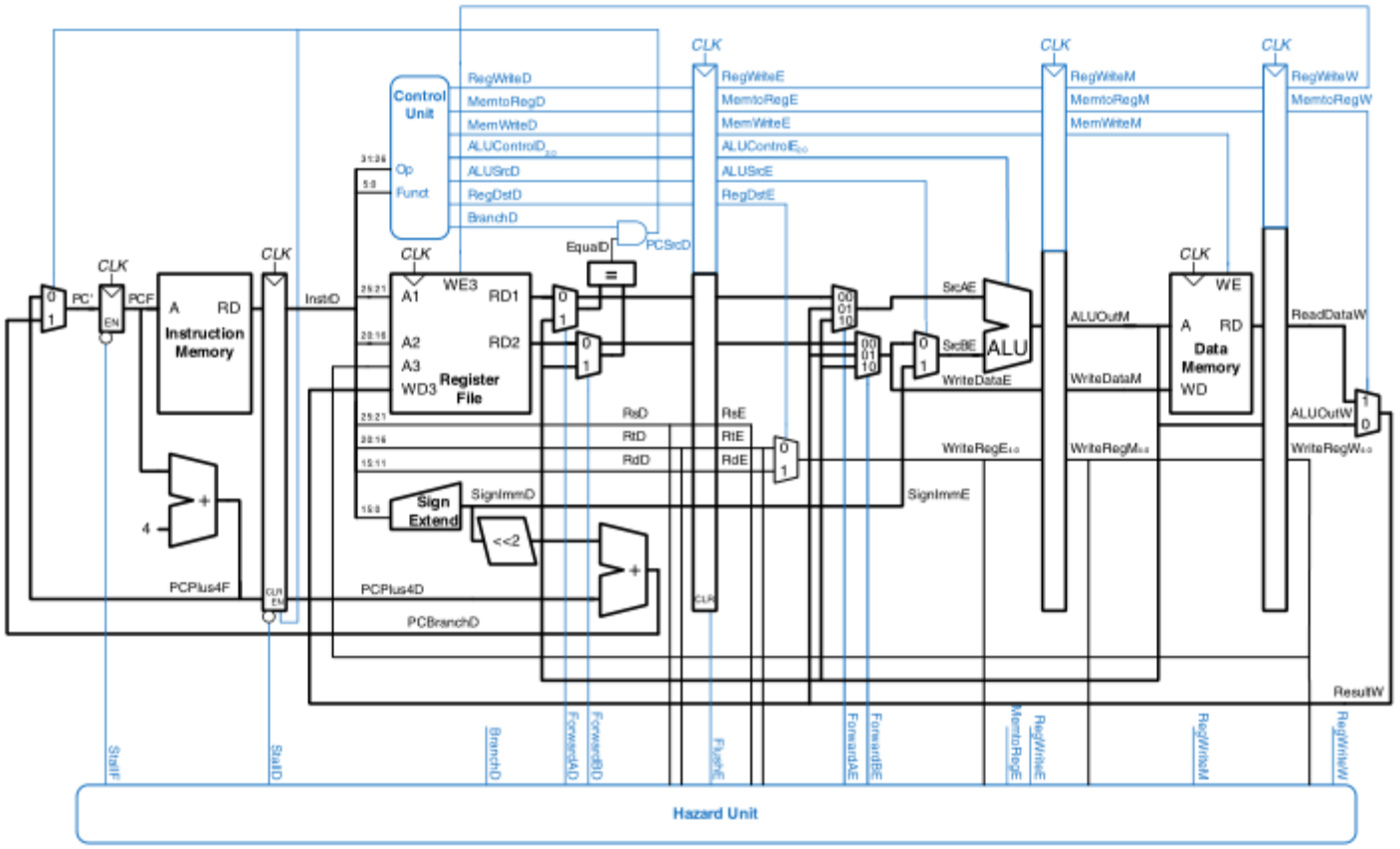


Figure 7.58 Pipelined processor with full hazard handling

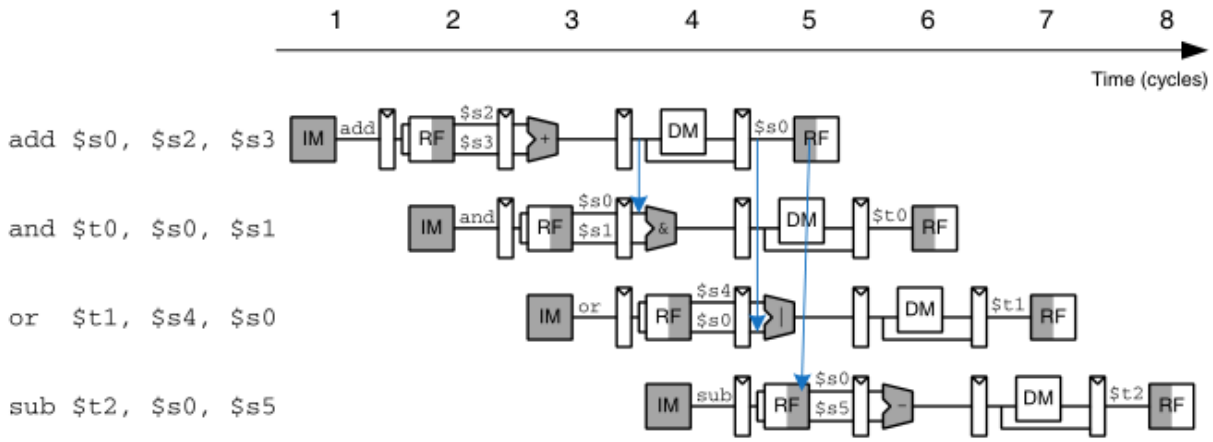


Figure 7.49 Abstract pipeline diagram illustrating forwarding

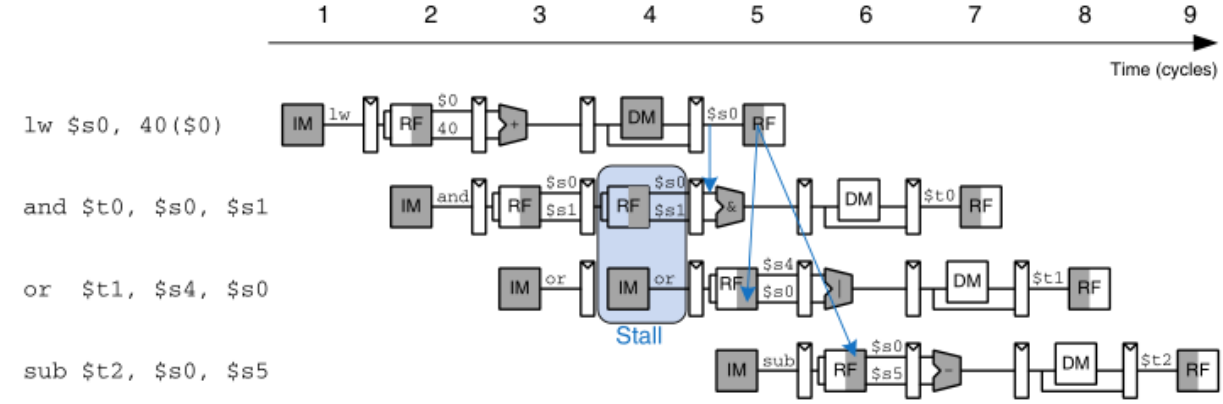


Figure 7.52 Abstract pipeline diagram illustrating stall to solve hazards

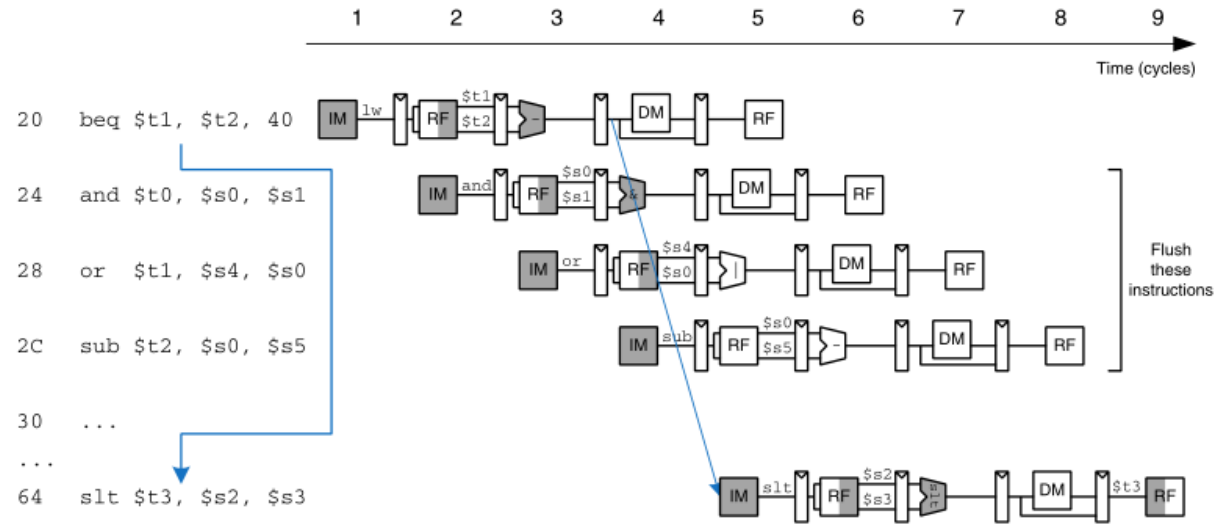
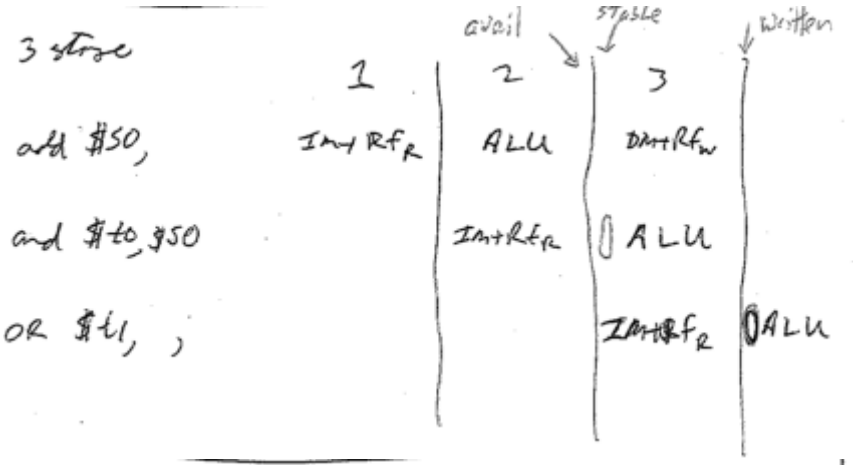
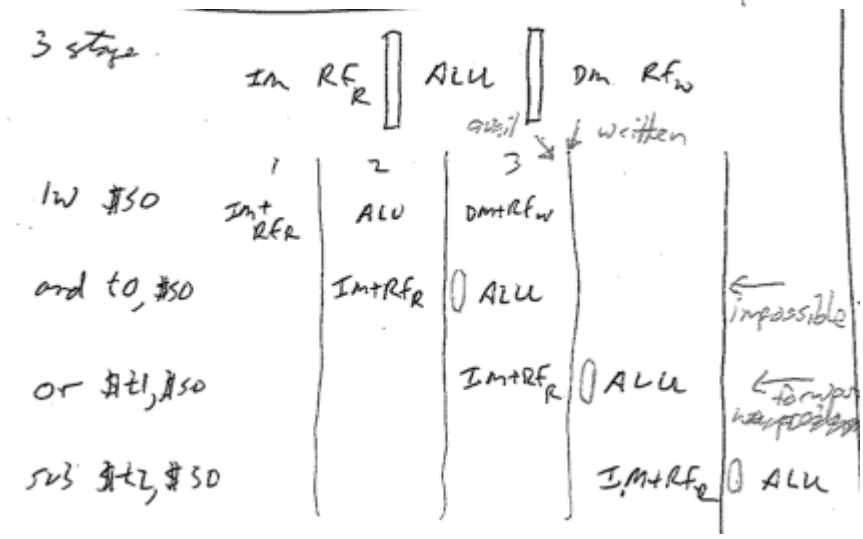
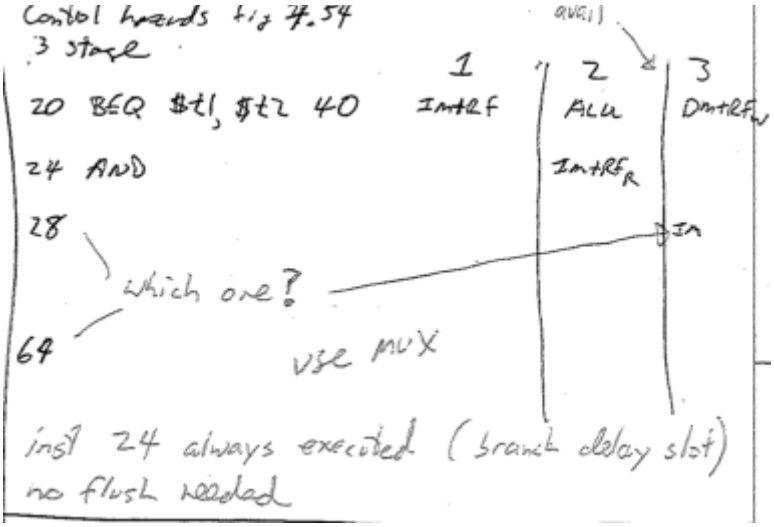


Figure 7.54 Abstract pipeline diagram illustrating flushing when a branch is taken






Control hazards fig 7.54
3 stage
1
2
3
20 BEQ $t1, $t2 40
IntRF
ALU
24 AND
28
which one?
64
use MUX
inst 24 always executed (branch delay slot)
no flush needed