Computer Science M151B, Homework 7

Michael Wu
UID: 404751542

May 21st, 2018

Problem 1

For the transfer using 4-word blocks, each transfer will take 1 cycle to send the address to memory, 24 cycles to access memory for the first set of words, 1 cycle to send the data from memory, and 2 idle clock cycles. So each 4-word block takes 28 cycles to read. This is 140 ns for each 4-word block. Thus the sustained bandwidth of the read is

$$4\,\frac{\text{words}}{\text{block}} \times 4\,\frac{\text{bytes}}{\text{word}} \times \frac{1\text{ block}}{1.4 \times 10^{-7}\text{ s}} = 1.1423 \times 10^{8}\,\frac{\text{bytes}}{\text{s}}$$

The latency to transfer 256 words is then

$$\frac{256\text{ words}}{4\,\frac{\text{words}}{\text{block}}} \times 140\,\frac{\text{ns}}{\text{block}} = 8960\text{ ns}$$

and the number of bus transactions per second is

$$\frac{1\text{ transaction}}{1.4 \times 10^{-7}\text{ s}} = 7.143 \times 10^{6}\,\frac{\text{transactions}}{\text{s}}$$

For the transfer using 16-word blocks, each transfer will take 1 cycle to send the address to memory, 24 cycles to access memory for the first set of words, 1 cycle for each 4-word block to send the data from memory, and 2 idle clock cycles for each 4-word block. So each 16-word block takes 37 cycles to read. This is 185 ns for each 16-word block. Thus the sustained bandwidth of the read is

$$16\,\frac{\text{words}}{\text{block}} \times 4\,\frac{\text{bytes}}{\text{word}} \times \frac{1\text{ block}}{1.85 \times 10^{-7}\text{ s}} = 3.4595 \times 10^{8}\,\frac{\text{bytes}}{\text{s}}$$

The latency to transfer 256 words is then

$$\frac{256 \text{ words}}{16 \frac{\text{words}}{\text{block}}} \times 185 \frac{\text{ns}}{\text{block}} = 2960 \text{ ns}$$

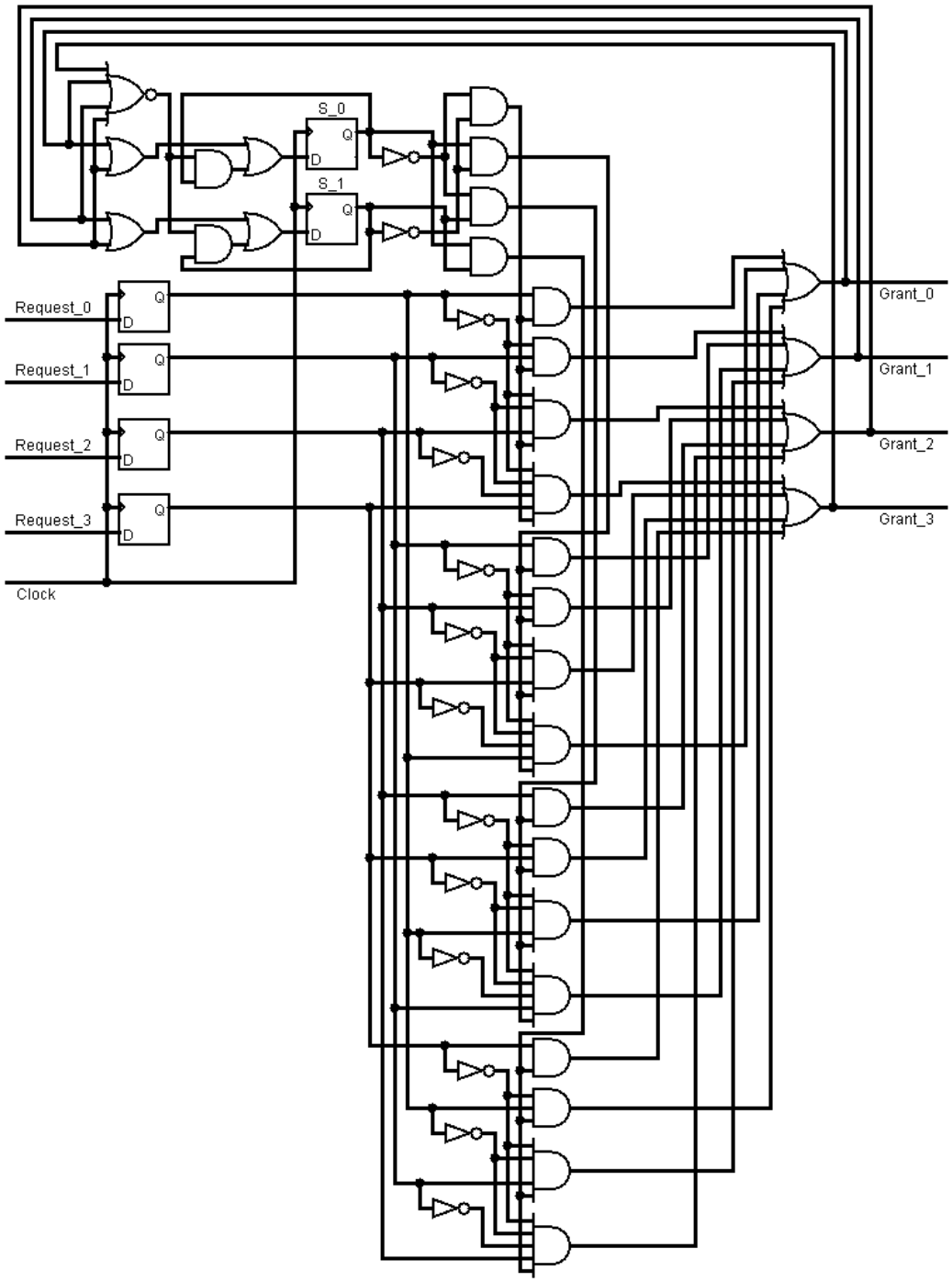
and the number of bus transactions per second is

$$\frac{1 \text{ transaction}}{1.85 \times 10^{-7} \text{ s}} = 5.405 \times 10^{6} \frac{\text{transactions}}{\text{s}}$$

Problem 2

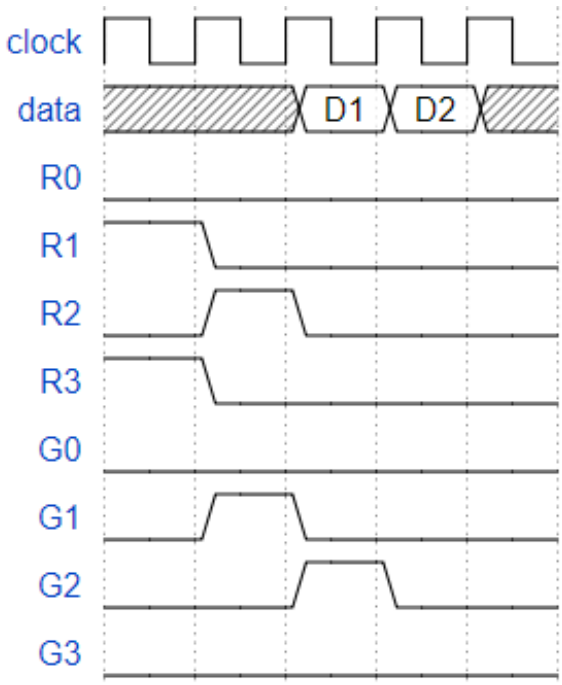
In DMA-based I/O, an interrupt must still be given to the CPU to indicate that the device controller has finished an I/O request and put some data in memory. The difference between interrupt-driven I/O and DMA-based I/O is that the interrupt-driven I/O will send an interrupt for every piece of data. The data in interrupt-driven I/O always passes through the CPU prior to going to memory, whereas in DMA-based I/O the device controller can directly modify memory. Thus DMA-based I/O introduces complexity into a computer's design because it may cause cache coherency problems. If the device controller modifies data in memory that has changed somewhere higher in the cache, the CPU may accidentally use the wrong data which would lead to incorrect behaviour.

Interrupt-driven I/O would be preferred for simpler devices where the performance penalty is small compared to DMA-based I/O. If a lot of data is not being transferred to memory at a time, then the advantage of DMA-based I/O being able to pass data quickly to memory is diminished. For example, if there is a sensor that generates one byte every second and sends it to the processor, then interrupt-driven I/O would be preferred. This is because the device does not use a lot of CPU time, so the performance improvement of DMA-based I/O would not be significant. If DMA-based I/O were used, this would add unnecessary complexity.


S_0
Q
D
S_1
Q
D
Request_0
Request_1
Request_2
Request_3
Clock
Q
D
Grant_0
Grant_1
Grant_2
Grant_3

a) My arbiter design is shown above. It uses D flip-flops to keep track of a state variable S which indicates which device has the highest priority. This signal gets decoded into four signals, each of which enables four of the sixteen `and` gates that control which grant line the arbiter will assert. I `or` together the outputs of the `and` gates that correspond to a grant for a particular device in order to get the output grant signals. Finally, if a grant is given then the state variable is updated to the next highest priority device. If no grant is given, the `nor` gate ensures that the state stays at the same value. I also use D flip-flops on the input request lines to ensure that the grant stays asserted for a whole cycle. The logic that this design implements is shown below.

Inputs						Outputs					
R_0	R_1	R_2	R_3	S_1	S_0	G_0	G_1	G_2	G_3	S_1	S_0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	1
0	0	0	0	1	0	0	0	0	0	1	0
0	0	0	0	1	1	0	0	0	0	1	1
1	x	x	x	0	0	1	0	0	0	0	1
0	1	x	x	0	0	0	1	0	0	1	0
0	0	1	x	0	0	0	0	1	0	1	1
0	0	0	1	0	0	0	0	0	1	0	0
x	1	x	x	0	1	0	1	0	0	1	0
x	0	1	x	0	1	0	0	1	0	1	1
x	0	0	1	0	1	0	0	0	1	0	0
1	0	0	0	0	1	1	0	0	0	0	1
x	x	1	x	1	0	0	0	1	0	1	1
x	x	0	1	1	0	0	0	0	1	0	0
1	x	0	0	1	0	1	0	0	0	0	1
0	1	0	0	1	0	0	1	0	0	1	0
x	x	x	1	1	1	0	0	0	1	0	0
1	x	x	0	1	1	1	0	0	0	0	1
0	1	x	0	1	1	0	1	0	0	1	0
0	0	1	0	1	1	0	0	1	0	1	1

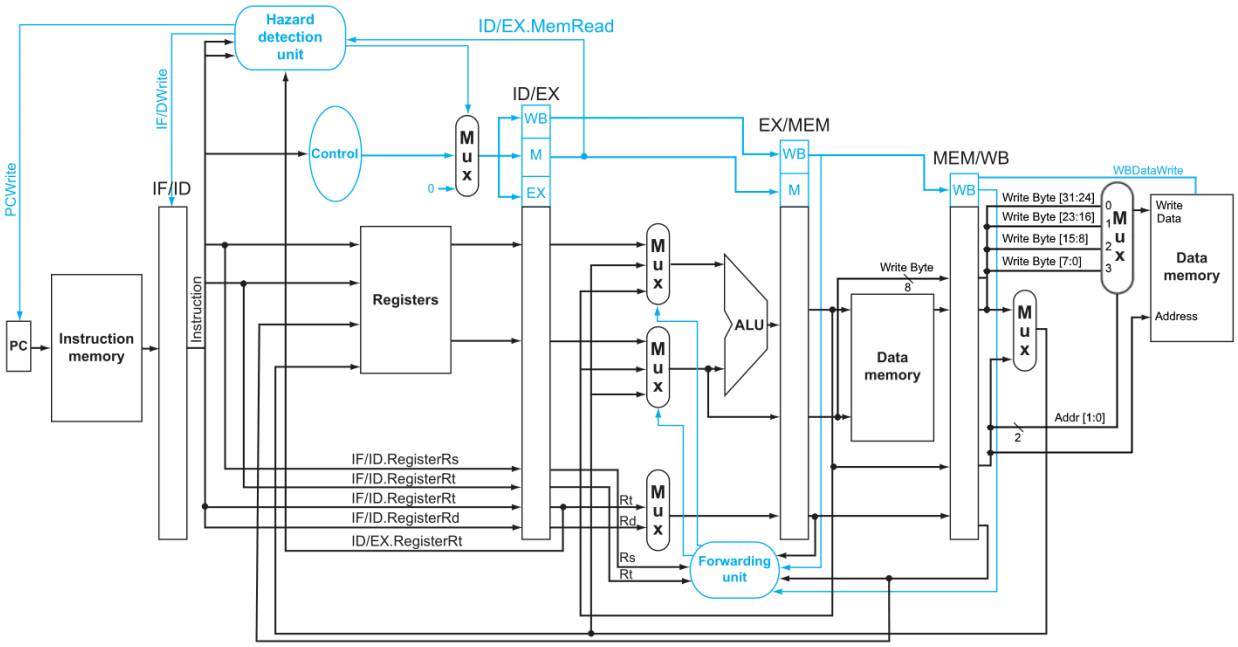


b) The timing diagram for my design is shown above. The arbiter first grants the bus to device 1, then device 2.

Problem 4

a) I will add another data memory unit in the `WB` stage. During the `sb` instruction in the `MEM` stage, the data memory performs a read from the given address. Then in the `WB` stage, the read data will be concatenated with the lowest 8 bits of the register that contains the data to be stored. There are four ways to perform this concatenation based on the lowest two bits of the store address, so a multiplexer will be used to choose which location to place the byte. Then the additional data memory unit is used to store the concatenated data in the store address.

b) The following figure shows my modifications. Note that I do not include forwarding between the `WB` and `MEM` stages to correct for any data dependencies between a `lw` or `sw` that follows a `sb` instruction. To implement



forwarding, I would have to check if the target address of the `lw` or `sw` is the same as the `sb`, and either replace the read contents for `lw` or have the write for `sw` override the write for `sb`.

Note that the label `Write Byte [31:24]` is a 32-bit signal that indicates the read data with bits 24 to 31 replaced by the 8 bit `Write Byte` signal, and similarly for the other four inputs to the multiplexer.

c) A new control signal called `WBDataWrite` is needed, which controls the write in the WB stage.

d) The changes to the control unit are shown in the following table.

Instruction	`RegDst`	`ALUOp1`	`ALUOp0`	`ALUSrc`	`Branch`	`MemRead`	`MemWrite`	`RegWrite`	`MemtoReg`	`WBDataWrite`
R-Format	1	1	0	0	0	0	0	1	0	0
`lw`	0	0	0	1	0	1	0	1	1	0
`sw`	x	0	0	1	0	0	1	0	x	0
`sb`	x	0	0	1	0	1	0	0	x	1
`beq`	x	0	1	0	1	0	0	0	x	0

Problem 5

Sequence A could be completed faster, as reads using DRAM only require a read-write cycle to ensure that the charge in the capacitors is refreshed, while writes require a read-modify-write cycle to change the data in the specified address. When writing, we must read before modifying because we can only write an entire row at once, so we must first know what the values stored in the row are. Then we change the specified bit, and write back. Thus writing has strictly more operations to perform than reading, so reading can be completed faster.

Problem 6

There are 32 instruction bits and 32 data bits that can be read every cycle, so this would require at least 8 of the given chips to supply the required 64 bits of output. Thus the size of our memory would be 2^{27} bits, or 128 Mibit.