CS552 Final Project

TripleSix

Xuyi Ruan & Yudong Sun

05/02/2016

Design Overview

The overall project was to design and implement a five stages pipeline processor. The project was incremental through different milestones along the entire semester. Since the overall work load of the processor project was considered heavy to complete within the semester along with other courses we were taking. For each milestone, we met as a team first went through and discussed design choices in a high hierarchy and planned out schematic diagram. After came up with detail schematic, our chief implementation engineer, Xuyi Ruan, was responsible for creating required modules and implemented the logics and functionality with Verilog. At the same time, our chief architect, Yudong Sun, was responsible for making decision for the rest of the design details, as well as fine grain adjustments on the design schematic. Some further verification was also performed by him to ensure our processer meet the basic requirements from the ISA specification.

Our processor design included the following five stages: Instruction Fetch, Decode, Execution, Memory, and Write Back stages. Each of the stage is responsible for only one specified task as indicate from the name of the stage. Instruction Fetch stage fetches the instruction indicated by current PC from the instruction memory and pass alone the instruction to the next stage in the pipeline. Decode stage parses the entire instruction into sub-portions to identify the specified instruction needs to be executing on. The decode stage generates control signals base on the op-code, and it also incudes the register file. The execution stage contains ALU and its job is mainly responsible for any arithmetic and logical operations, address calculation for LD and ST instructions and for test the condition of branch instructions. The result from the execution stage will propagate to the memory stage. Memory stage is relatively simpler compared to the previous stages. It handles instructions with load/store operation, which will read/write the memory. Compare to the non-pipelined version in the first demo, the benefit of pipeline design by breaking an instruction into five stages enable higher throughput and improve performance of the processor. The cache implementation before the final demo looses the perfect memory assumption, and try to reduce the amount of time that the processor stall waiting for the memory requests.

Optimizations and Discussions

Our optimization mainly focuses on the improvement of the data arrival time by improving the slack time from -2.98 ns to -0.89 ns to ensure our implementation will work on actual hardware. The timing synthesis reports tells us whether or not our processor design will work as expect on actual hardware. It takes time for human to transport from place to place and that analogy holds the same for data signal travel through wires and transistors. Different requirement for data arrival time will be reinforced depend on various hardware specification. For example, our processor design runs on a clock period of 100 ns (10 MHz), and the data arrival requirement is 0.95 ns. Under this requirement, the data travel time (from fetch stage through write back stage) should not exceed the 0.95 ns frame, otherwise, our

processor will not behave as expected as to the simulation. Since timing issue was not re-enforced in the previous demos, our demo3 result in a violation of -2.98 ns.

After carefully analyzing the timing report, we noticed that most of the extra time was spent at exestuation stage. We tracked down the serially ALUs we previously used for some simple subtraction. We used those ALUs to compensate the lack of rotate right function from the previous ALU design. We noticed that rotate right function could utilize the rotate left, which was currently embedded in our ALU. If we need rotate right for 2 bit positions that will be the same as we perform rotate left for 14-bit position (16-2=14). However, this required an additional subtraction of 16 with the require bit positions. Therefore, we serially added two ALUs into the data path. The result of the serially connection of multiple ALUs significantly affect the data arrival time and we need to come up with more efficient algorithm to improve the slack time.

Our chief architect realized instead of using the ALU for the simple subtraction, we could simply use a lookup table for this calculation. By switching out two ALUs we had in designed before with the LUT, the new design gains us 2 ns faster data arrival time compared to the arrival time of previous design.

We also implemented the extra credit instructions for exception handling. There is actually not much need to be done for the exception handling because most of the components required for exception handling was already done in the previous design. The behavior of siic and rti instruction is very similar to a branch or jump type instructions. So handing exceptions becomes trivial, just like handling a branch mis-prediction.

Design Analysis

Hazard Type	Stall cycle	Explanation
Data Hazard	1 cycle	Even most of the data hazard types could be solved with forwarding unit. The load instruction immediately follows an instruction whose data source has data dependency on load will need to stall for one cycle. The one cycle stall allows the data out from memory stably feed into the input for the exestuation stage for next instruction.
Control Hazard	2 cycle	Our processor use branch will not be taken as our branch prediction strategy. The condition test of the branch happens at execution stage. If the branch condition detect that the

		branch was actually taken, we need to flush both IF/ID, as well as ID/EX pipeline stage registers (waste of 2 cycles of work).
I-Cache Miss	cache-miss (with eviction of a line) 16 cycles. cache-miss (without any eviction) 9 cycles	On instruction cache miss, the processor will not be able to execute any new instructions until a new instruction is fetched from memory. During the I-Cache Miss, nop instruction will be inserted until new instruction is fetched.
D-Cache Miss	cache-miss (with eviction of a line) 16 cycles. cache-miss (without any eviction) 9 cycles	On data cache miss, stall signal will be asserted and processor will stall for 16/9 cycles until stall signal from cache module goes low again.

We used a Moore machine with about 16 states to implement the cache module. We utilized the characteristic of four-bank memory as well as the 4-cycle write and 2-cycle read feature to minimize the require clock cycles.

Cache Hit/Miss	Cycles	Explanation
cycles for a cache-hit	2 cycles	one cycle to detect read/write to cache, the other cycle to detect hit/miss base on the memory address.
cache-miss (with eviction of a line)	16 cycles	the line eviction operation need to write the entire four words in the current cache line back to memory. Each of the memory write has a cost of four cycles. By utilizing the four-bank memory, we are able to perform four memory writes in 7 cycles.
cache-miss (without any eviction)	9 cycles	By utilizing the four-bank structure of the memory system, we are able to retrieve four words from memory to cache with 7 cycles. Plus, the two cycles of determine read or write operation as well as hit/miss on cache. We are able to handle no eviction cache miss in 9 cycles.

Conclusions and Final Thoughts

By doing this project, we as a team learned in a hard way that the design of a processor takes time, especially when lots of design choice need to be made. The process of building our own processor allows us to have a deeper understanding of following concepts raised during the design. By utilizing 5-stage pipelining, we can achieve 5x higher clock frequency compare to the none pipelined version in the ideal case.

Hazard detection and forwarding is the second important idea in processor design. Our processor design mainly focuses on handling the data and control hazards. Data hazard happens when the data source of one instruction depend on the result of previous instruction, while the register file has not reflected the updated values from the previous instruction. We can solve data hazard by using forwarding unit, which directly wire the result from either execution/write back stage register even the update value has not yet been reflect to the register files.

Control hazard happens when there is any branch or jump instructions present. Here we take the approach of assuming the branch will always not take as our branch prediction strategy. When it comes to a mis-prediction, the processor needs to flush the incorrect instructions in the pipeline. At the same time, the correct branch target will be written to PC, and the correct instruction will be fetched.

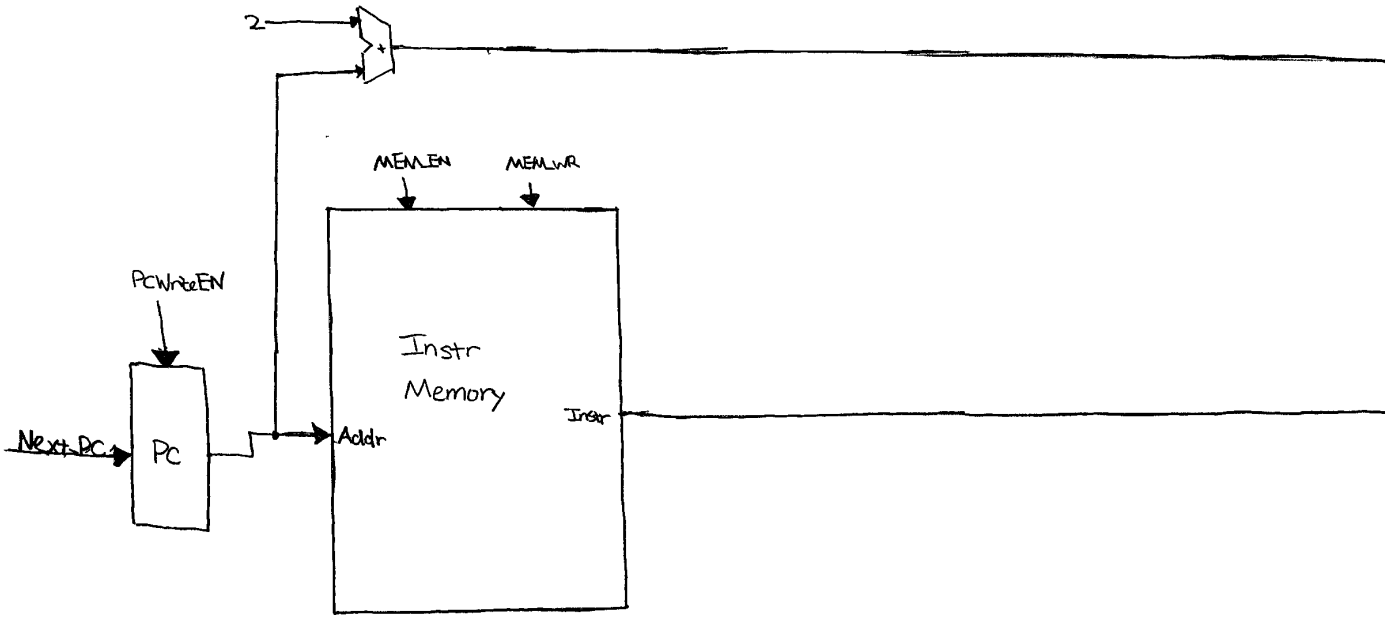
The final implementation of two-way set associative cache looses the perfect memory assumption that we used in the demo 1 and demo 2. Real memory access is very expensive with respect to the number of cycles, during those cycles the processor has to stall. Adding a cache allows the processor directly retrieve instruction/data from cache when a cache hit occur. If the request information is not currently in cache, we still need to fetch data from memory and hope for a cache hit for the next request. To further increase the cache hit, the idea of set-associative cache was introduced. Instead of one block per cache line, we can put more than one block into the same cache line to reduce the chances of conflict miss. The down side of having large set associative is the more complicated design, larger area and higher power consumptions.

We are very happy our hard work finally results in a fully functioning 5 stages pipeline processor. If we have more time, we will further work on optimizing our processor in both area and power composition. We are thankful for all the helps from classmate, TAs and Professor Karu.

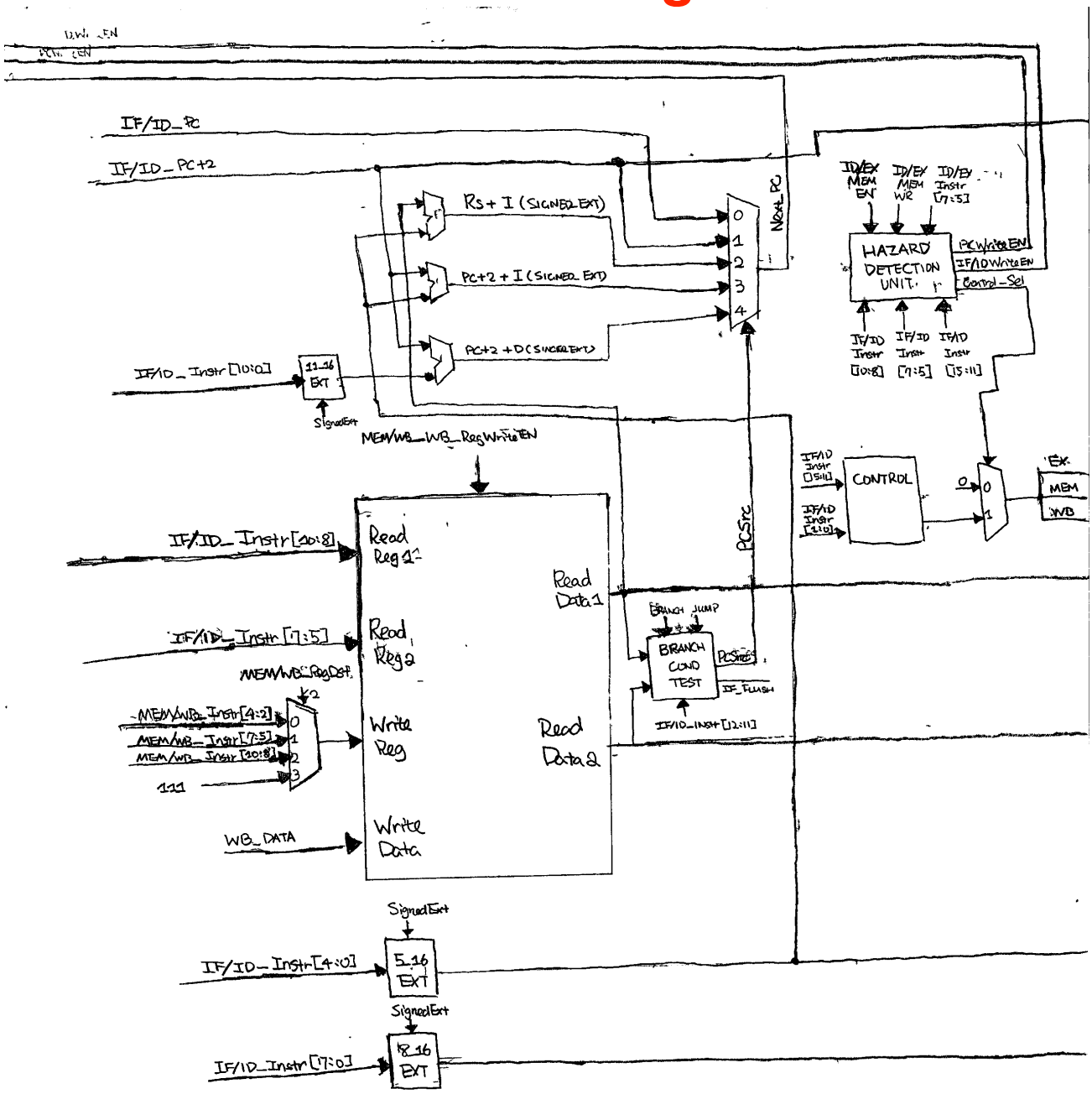
Schematic and FSMs Attached below:

- **Design Overview: high-level processor schematics**.
- **State diagrams and schematics for two-way set associative cache design.**

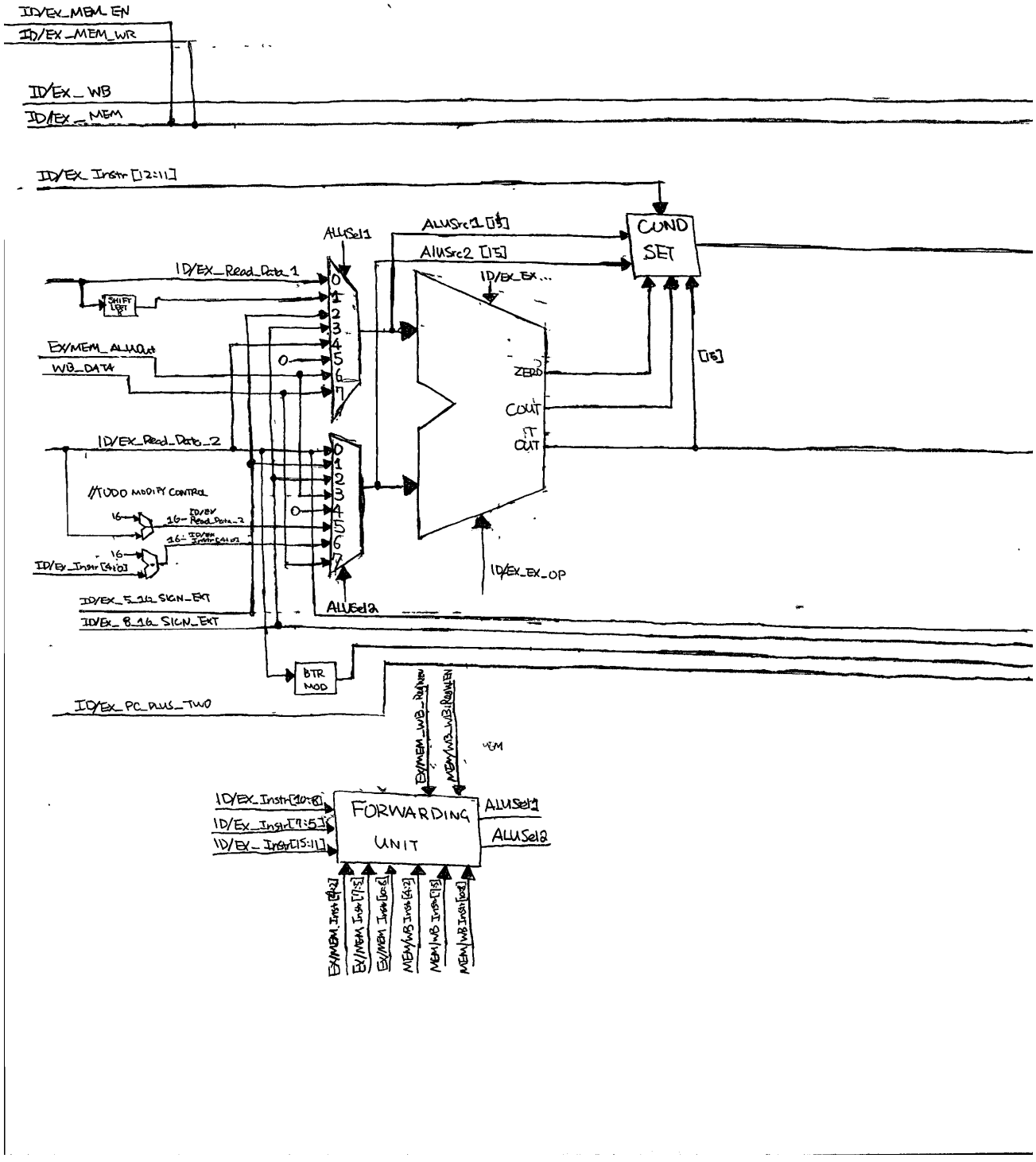
FETCH Stage



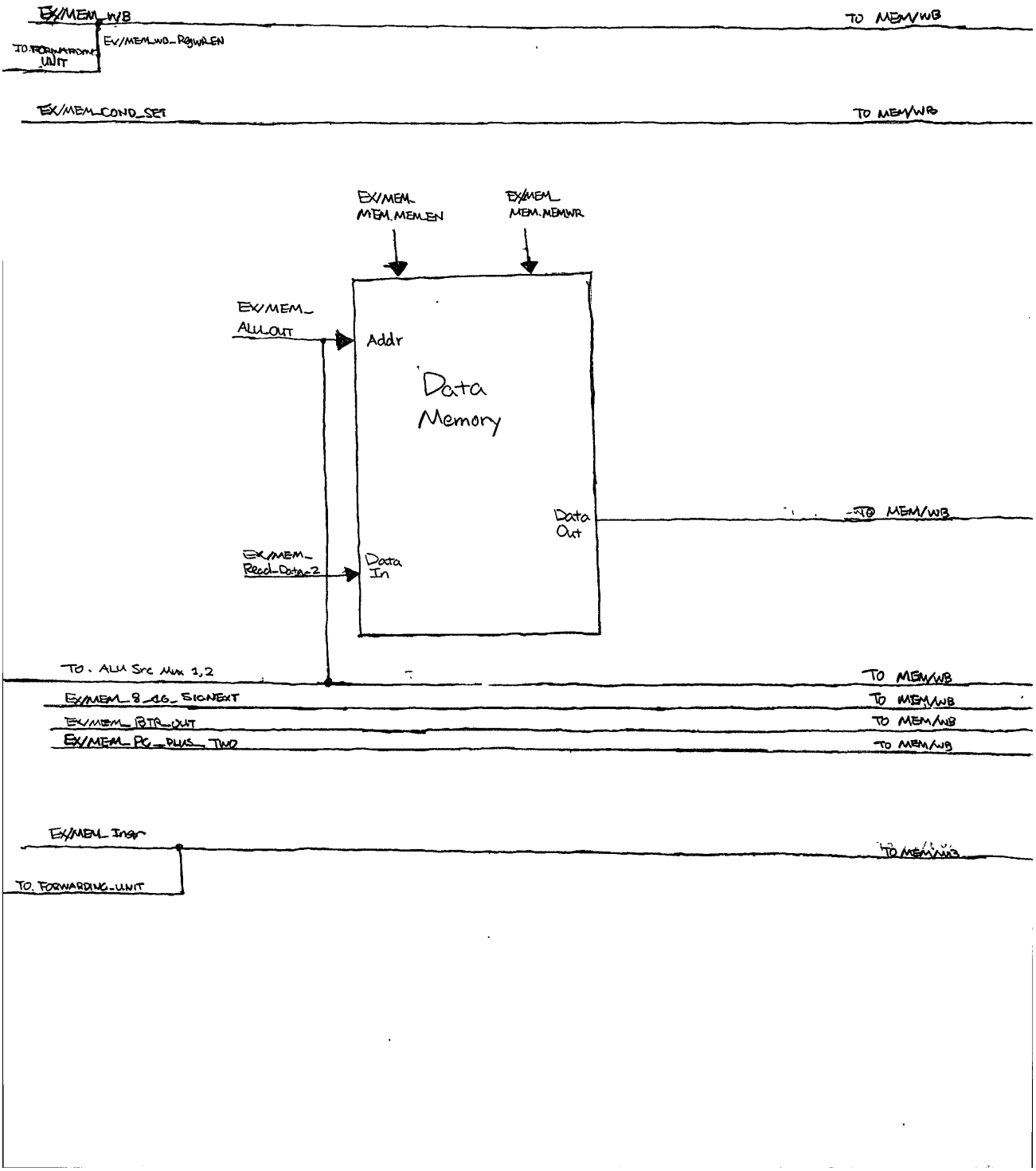
Decode Stage



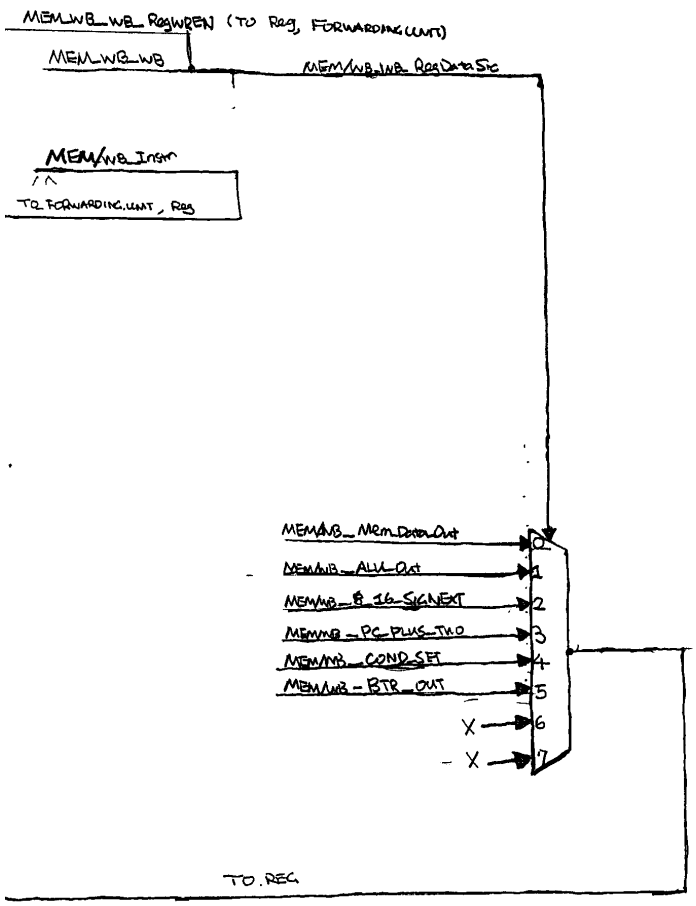
Exec Stage

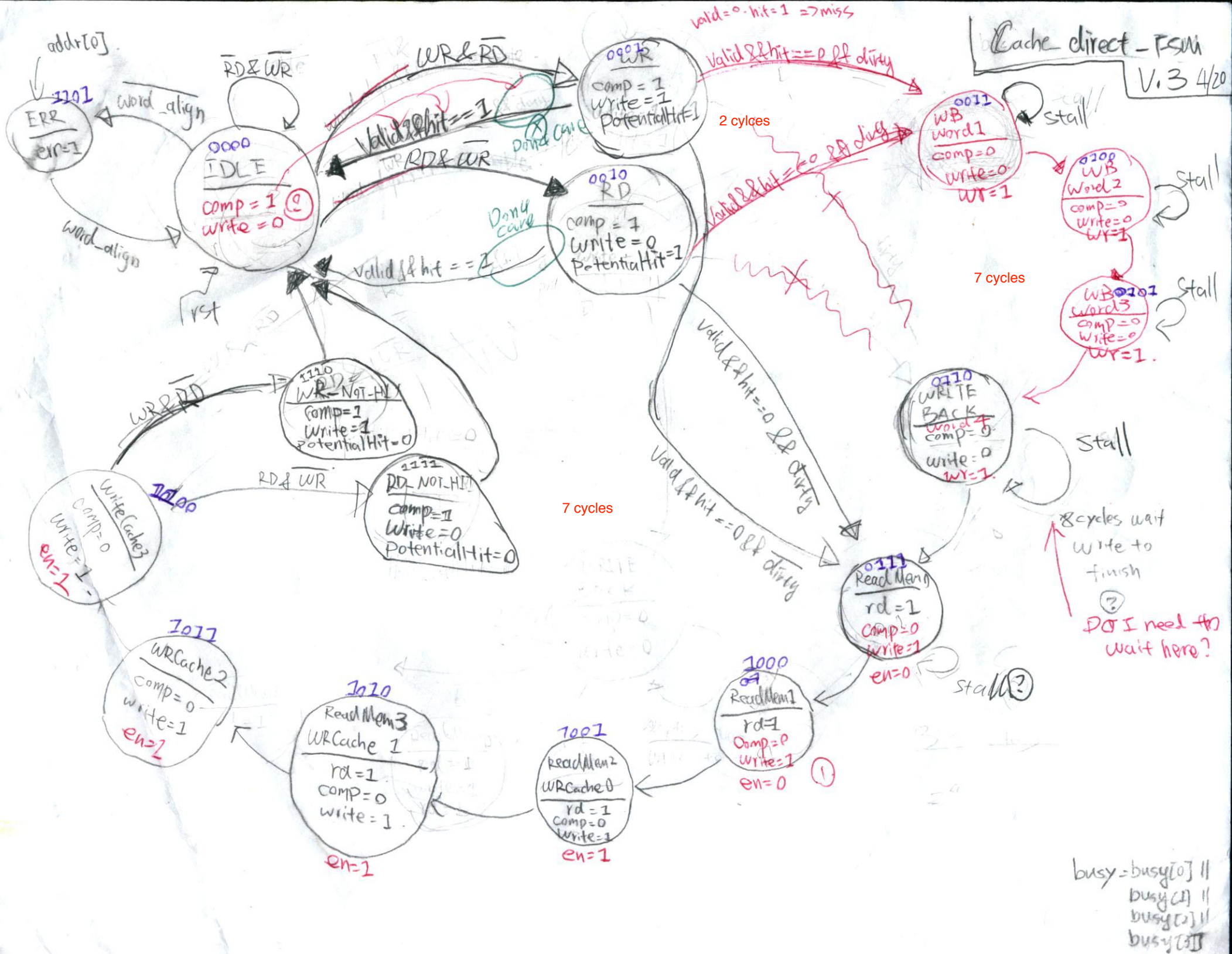

ID/EX_MEM_EN
ID/EX_MEM_WR
ID/EX_WB
ID/EX_MEM
ID/EX_Instr[12:11]
ALUSel1
ALUSrc1[15]
ALUSrc2[15]
COND SET
ID/EX_Read_Data_1
SHIFT LEFT 1
EX/MEM_ALUOut
WB_DATA
ZERO
COUT
ID/EX_Read_Data_2
//TODO MODIFY CONTROL
ID/EX_Instr[4:0]
ID/EX_5_16_SIGN_EXT
ID/EX_8_16_SIGN_EXT
ALUSel2
BTR MOD
ID/EX_PC_PLUS_TWO
ID/EX_EX_OP
EX/MEM_WB_RegWen
MEM/WB_WB_RegWEN
MEM
FORWARDING UNIT
ID/EX_Instr[10:8]
ID/Ex_Instr[7:5]
ID/EX_Instr[15:11]
ALUSel1
ALUSel2
EX/MEM_Instr[11:8]
EX/MEM_Instr[7:5]
EX/MEM_Instr[4:0]
MEM/WB_Instr[4:0]
MEM/WB_Instr[15:11]
MEM/WB_Instr[4:0]

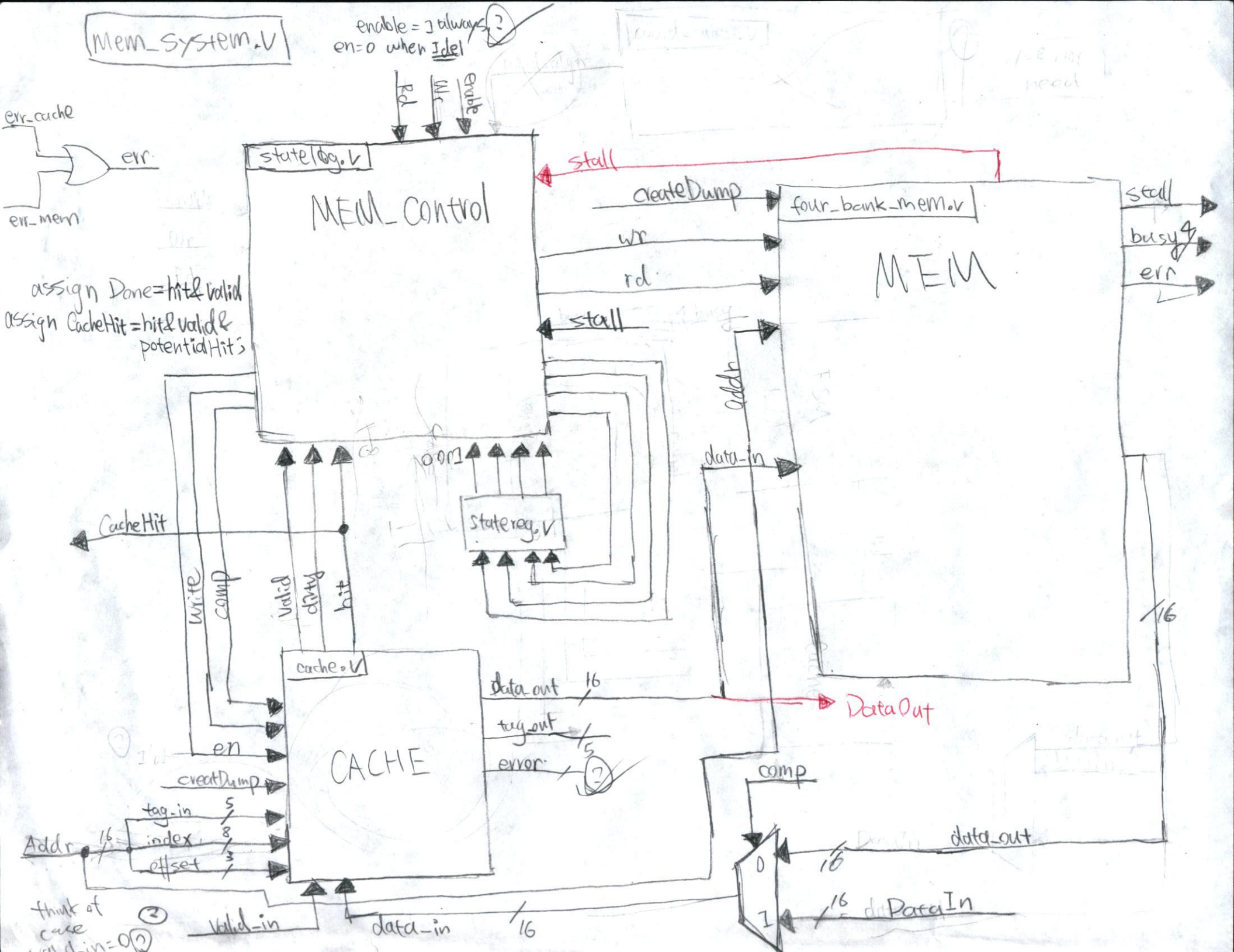
Mem Stage


EX/MEM_WB
TO MEM/WB
TO FORWARDING UNIT
EX/MEM_WB_RegWrEN
EX/MEM_COND_SET
TO MEM/WB
EX/MEM_ MEM.MemEN
EX/MEM_ MEM.MemWr
EX/MEM_ ALU_Out
Addr
Data Memory
Data Out
TO MEM/WB
EX/MEM_ Read_Data2
Data In
TO. ALU Src Mux 1,2
TO MEM/WB
EX/MEM_8_16_SignExt
TO MEM/WB
EX/MEM_BTR_Out
TO MEM/WB
EX/MEM_PC_PLUS_TWO
TO MEM/WB
EX/MEM_Instr
TO MEM/WB
TO. FORWARDING_UNIT

Write Back Stage



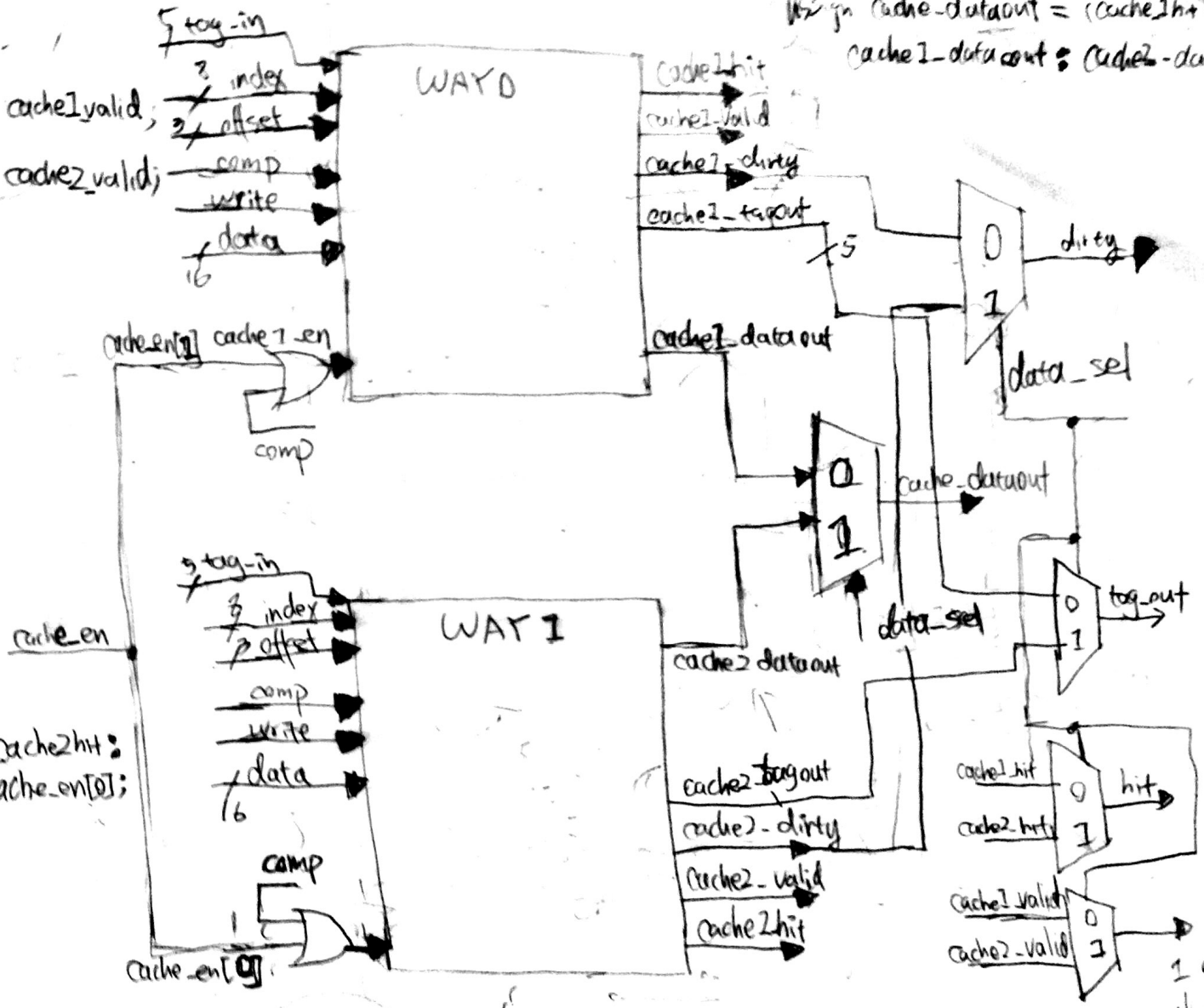

Cache direct - FSM
V.3 4/20
addr[0]
1101
ERR
err=1
word_align
RD&WR
0000
IDLE
comp = 1
write = 0
WR&RD
Valid&&hit==1
0010
RD
comp = 1
write = 0
PotentialHit=1
Dont care
Valid&&hit==1
rst
0011
WR
comp = 1
write = 1
PotentialHit=1
valid=0 hit=1 =>miss
Valid&&hit==0 && dirty
2 cylces
0011
WB
word1
comp=0
write=0
WT=1
stall
0100
WB
word2
comp=0
write=0
WT=1
stall
0101
WB
word3
comp=0
write=0
WT=1
stall
7 cycles
0110
WRITE
BACK
comp=0
write=0
WT=1
stall
Valid&&hit==0 && dirty
Valid&&hit==0 && dirty
1110
WR_NOT_HIT
comp=1
write=1
PotentialHit=0
WR&RD
1111
RD_NOT_HIT
comp=1
write=0
PotentialHit=0
RD&WR
7 cycles
1000
WriteCache3
comp=0
write=1
en=1
1011
WRCache2
comp=0
write=1
en=1
1010
ReadMem3
WRCache 1
rd=1
comp=0
write=1
en=1
1001
ReadMem2
WRCache0
rd=1
comp=0
write=1
en=1
0111
ReadMem1
rd=1
comp=0
write=1
en=0
1
stall?
8cycles wait
write to
finish
?
DO I need to
wait here?
busy=busy[0] ||
busy[1] ||
busy[2] ||
busy[3]


Mem_System.v
enable = 1 always ?
en=0 when Idel
rd
wr
enable
err_cache
err
err_mem
stateReg.v
MEM_Control
stall
createDump
four_bank_mem.v
MEM
wr
rd
stall
stall
busy
err
assign Done=hit& valid
assign CacheHit=hit& valid& potentialHit;
addr
data_in
com
stateReg.v
CacheHit
write
comp
valid
dirty
hit
cache.v
CACHE
data_out 16
tag_out 5
error
DataOut
en
creatDump
tag_in 5
index 8
offset 3
Addr 16
think of case wr_in=0 ?
Valid_in
data_in 16
comp
data_out
16
DataIn
16
0
1

assign cache1hit = cache1hit & cache1valid;
assign cache2hit = cache2hit & cache2_valid;

(V)

assign cache_dataout = (cache1hit == 1)? cache1_dataout : cache2_dataout;



data_sel = (cache1hit | cache2hit)? cache2hit : cache_en[0];

(V)

assign cache_en = (en == 1 && cache1_valid == 1 && cache2_valid == 0)? 2'b01 :
(en == 1 && cache1_valid == 0 && cache2_valid == 1)? 2'b10 :
(en == 1 && cache1_valid == 0 && cache2_valid == 0)? 2'b10 :
(en == 1 && cache1_valid == 1 && cache2_valid == 1)? pseudo :
2'b00;

(V)