Microarchitecture Project Final Report

- Design of 32-bit x86 microprocessor –

Abstract

In this project, we implemented single issue in-order 32-bit x86 microprocessor. Our processor runs at 62.5MHz and passed the all the cases. Due to fascinating memory system, our performance is comparable to high speed microprocessor design. Since it runs at relatively lower frequency, our processor still has benefit from lower power consumption.

1. Introduction

The x86 ISA (Instruction Set Architecture) has been adapted to the most of the high performance general purpose microprocessor since the advent of the Intel's 8086 microprocessor. In this project, we design a 32-bit microprocessor supports x86 architecture. Our processor is capable to support single issue in-order processing. It has 256 bytes single level 2-way set associative instruction cache, and 768 bytes single level 3-way set associative data cache. It exploits 6-stage pipeline architecture to improve throughput. Several techniques are used for improving performance such as non-blocking cache, wrapping addressing mode, and bypassing logic to minimize the bubble in the pipeline.

The processor should support 26 x86 instructions, which have large number of variants per each instruction. Each instruction utilizes the most of prefix and several addressing modes including SIB. It should support hierarchical memory structure consists of cache memory and main memory, and different I/O devices. It is also asked to support segmented address space, virtual address, exception/interrupt handler. In this work, we put more emphasis on designing a fully functioned x86 microprocessor for a given instruction set. Even though the most of the modern microprocessor exploits the out-of-order processing to improve performance, with considering a limited resource we have we firstly focus on implementing single-issue in-order microprocessor, and consider supporting the more sophisticated techniques to extended work. Instead, we put more effort on enhancing efficiency while maintaining simple in-order processing. As a result, our processor is able to run at 62.5MHz. By exploiting several optimization techniques, however, its execution time is comparable to that of the processors run at higher frequency. Therefore, our design benefits from lower power consumption while maintaining high throughput.

The rest of this paper is organized as follows. Section 2 describes the data path design, while section 3 explains about control path design. In the section 4, we analyze the critical path in our design. Section 5 describes the details of each pipeline stage and other major blocks such as cache memories and main memory. Finally, in section 6, we draw our conclusion.

2. Overview of data-path design

A simple pipeline structure typically consists of 5-stage, which are fetch, decode, execution, memory, and writeback respectively. It is mostly based on the load-store architecture, which has been the fundamental architecture of the RISC (Reduced Instruction Set Computing) based processor. However, the x86 ISA has different approach compared to the RISC based microprocessor. A single x86 instruction directly accesses the memory to compute data, so the RISC based microprocessor may need to execute multiple instructions to support it. Thus, we decided to modify the pipeline structure to support x86 instruction more efficiently. As a result, our processor can execute almost every instruction on every single cycle unless it suffers from dependency between other instructions. Figure 1 shows the block diagram of our data path design, which consists of 6 stages. These are fetch (FE), decode (DE), address generation (AG), memory read (MEM), execution (EX), and write back (WB) respectively.

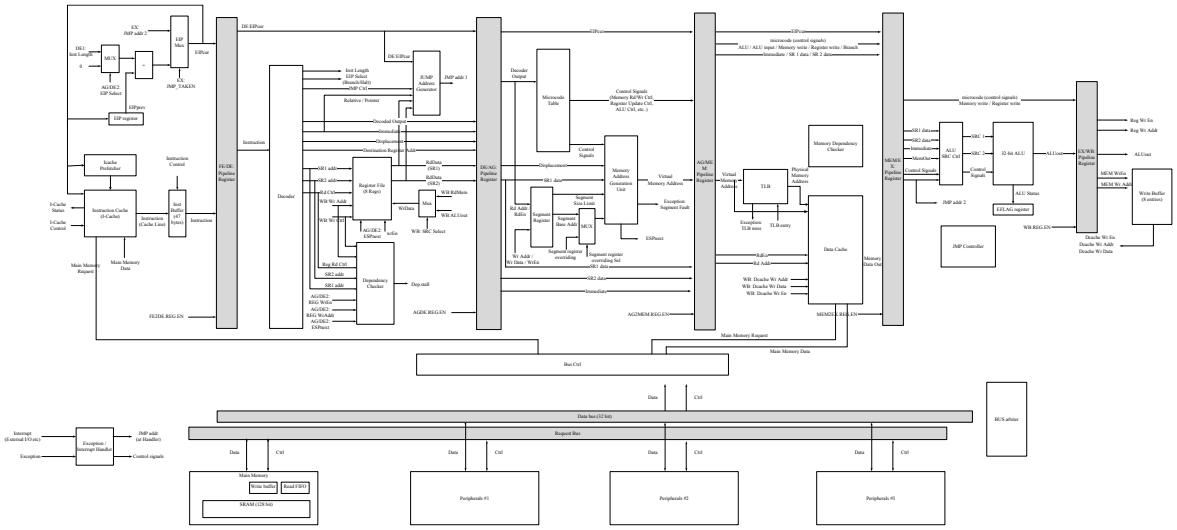

Figure 1 Data path diagram (The details are in the following sections)

Each stage is isolated from others by using pipeline registers, and every updated value will basically be written in the writeback stage to either memory or register. The register file in our design is located in the AG stage, so every read request to the register file should be done in the AG stage while the write requests are generated from the WB stage. It helps to reduce the design complexity of controllers including register dependency checkers. However, with some instructions like subsequent PUSH/POP operations and REP MOVS instruction, every subsequent instruction generates pipeline stall because the following instruction has to access the same register which would be updated by the current instruction. To resolve it, we utilize internal registers between AG and MEM stage, and it provides most recent value to the following instruction.

Since our data path includes two stages that send requests to memory but the available memory module in this project has only one port for either read or write, we need to resolve this conflict to make our design more efficient. Our approach is utilizing a controller between memory requests and data cache to manage the traffic between requests from different stages, MEM and WB. This controller is not only for managing conflict between requests from two different stages, but also enhancing write back performance by

buffering write requests to a small sized write back buffer. Consequently, our processor rarely experiences write back stalls while maintaining the correctness.

Our processor is asked to support a number of SIMD operations, which utilizing 64-bit MMX registers. After analyzing the requirements to support those SIMD instructions, we made a conclusion to use 64-bit interface between data cache and the pipeline. Since one data cache line size is more than 4 bytes, it does not incur any noticeable impact on our design but improves performance significantly. A given SIMD instruction requires 64-bit addition, but it is easily supported by adding one 32-bit adder that runs in parallel with the original 32-bit ALU logic.

In short, we can execute every x86 instruction except for IRETD, which needs to read 10 or 12 bytes from memory, on every single cycle unless it has dependency issue.

3. Overview of the control-path design

Our microprocessor does speculative execution assuming jump operation is always not taken. Considering that jump instruction needs to read data from memory, we put jump handling logic in EX stage. If the jump should be taken, the logic generates control signals, which flushes the pipeline. Then, new EIP or CS:EIP fed back to FE stage to restart with the new instruction. It enhances performance compared with stalling pipeline. Since the misprediction handling is done at EX stage, not WB stage, we can save one clock cycle time compared to the case processing it at WB stage when prediction becomes wrong. In addition, by putting jump handler at EX stage, we do not need to have EFLAGS dependency checking logic because EFLAGS is updated at EX stage. It simplifies our design. Figure 4 shows the diagram of jump handling logic.

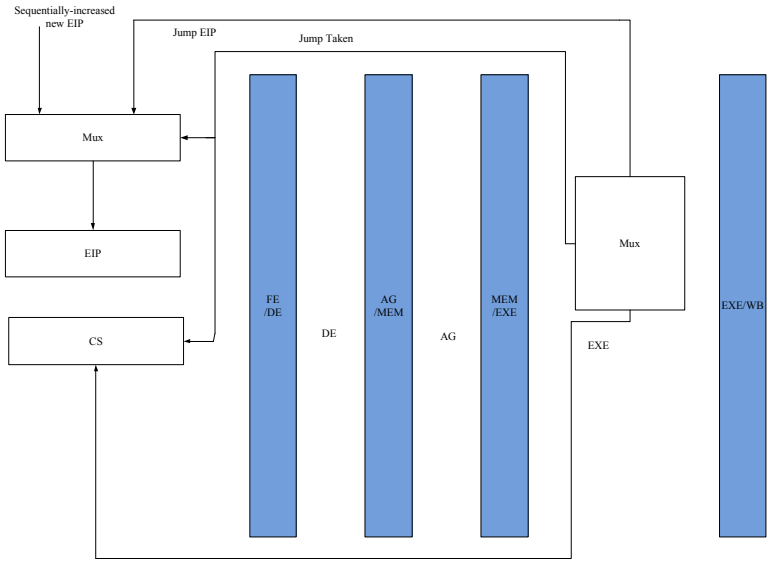

Figure 2 Block diagram of control path design

4. Critical path analysis

Our critical path resides in AG stage and we run our processor at 62.5MHz. Since we split one stage into two sub-stage to utilize limited resource in register file, we effectively have half cycle time in some case and there are subsequent addition to get memory address, so it incurs a critical path.

5. Description of the detailed design

5.1. Data path design

5.1.1. Fetch (FE) stage

The FE stage consists of several parts, which are the instruction cache (I-cache), instruction address generator and shifter, segment checking logic, and tail pointer for instruction buffer logic. Figure 3 shows the block diagram of FE stage. To fetch one instruction, it first calculates the address for sending a request to the I-cache, and then it shifts received data from I-Cache before putting into the instruction buffer. By doing this, the first byte in the instruction buffer always becomes the first byte of new instruction, which makes the design of decoder much simpler.

To shift the data from I-cache, we have 2 kinds of operations. First, when the buffer contains less than 32-byte data, it should push data read from I-cache into instruction buffer at the correct position which is indicated by tail point. Tail point indicates the last valid byte in the buffer. Second, when DE stage is not stalled, it should pop data from the instruction buffer by the length of the last instruction. In this way, it keeps moving bytes to the buffer and ensures the new instruction always starts from the beginning of the buffer. Since the size of the instruction buffer is 47 bytes, which is bigger than that of the longest instruction, it can request an instruction earlier than it needed. If the tail pointer, which points the last valid byte in the buffer, are less than a given threshold value, for example 32 bytes, it requests to get new data from I-cache to instruction buffer. In that way, it can almost always fill the instruction buffer.

According to the current buffer status, with the instruction length fed back from DE stage and the size of the data read from I-cache, the EIP and fetching address pointer are updated. At the same time, it checks segmentation fault by comparing the current EIP and CS limit value. Our design supports always not taken branch predictor. When the branch is taken, the processor updates the EIP (and CS if it needed) and flush the buffer according to the signal comes from jump handling logic in EX Stage.

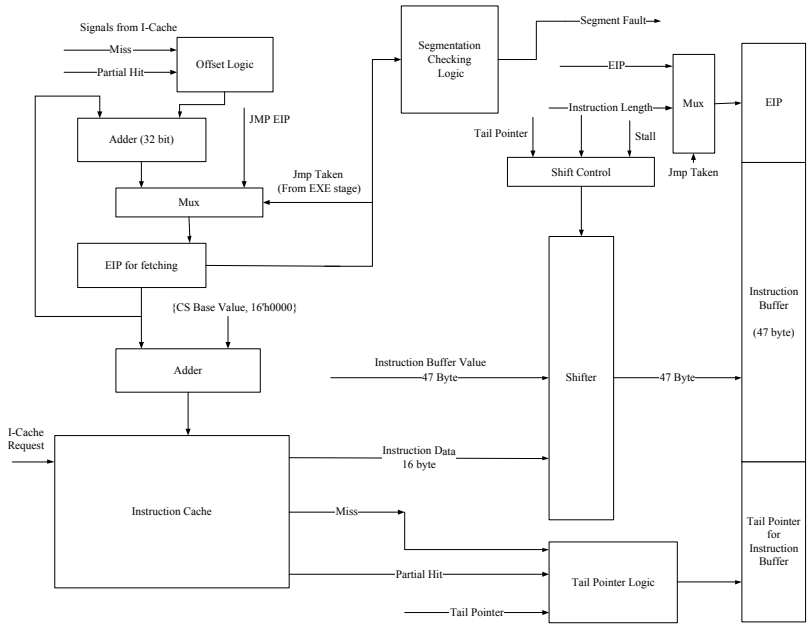

Figure 3 Block diagram of fetch stage

I-cache in our design has advanced features to support unaligned access. It is able to provide 16 bytes data even though they are across 2 cache lines. In addition, even though the second line is missing, it still provides data from the first cache line. In this way, the fetch logic can get as many bytes as possible, which helps to increase the performance by reducing pipeline stalls.

The critical path in FE stage is: Address Register - I-Cache Data - Shifter, which is 6.7 ns (1*MUX2$ + 4*MUX4$ + I-Cache read delay).

5.1.2. Decode

In the DE stage, it has to complete the following tasks. First, it has to calculate the length of current instruction and feed it back to fetch stage. Second, it has to figure out the each parts of the instruction. Lastly, from the preliminary decoded result, it should generate the appropriate micro-code.

Compared to RISC processors, decoding an x86 instruction is quite complicated because there are lots of variations in the instruction format. Since the instruction format is not fixed across the instruction set, it has to figure out the current instruction format before decoding information from the instruction. To achieve it quickly, we exploit parallel processing. Several decoder logics run in parallel with different input data to find out the appropriate value for each field.

Even though x86 instruction does not have fixed format, it still has some useful characteristic to reduce decoding overhead. The prefix can be existed within the first 2 byte and it decides the location of the opcode. The opcode should be within the first 4 bytes and it also determines the immediate size and possibly displacement size. ModR/M byte can be between the 1st and the 5th byte, and it determines the existence of SIB byte and displacement size. Therefore, to decode an instruction in parallel, we first built 3 basic decoders for prefix, opcode and ModR/M, and then put them at the every possible location to

decode input data. Since these sub-decoders can execute simultaneously, we can get the output from each logic much faster. After getting the results from each sub-decoder, it can generate control signals from that information to control the other logics, which select the correct value and the size of each part.

After figuring out each part of the instruction, we use that information to generate appropriate micro-code and propagate operands in the instruction into the following stages. We built a micro-code table and this table is accessed in DE stage. The index to access the entry of the micro-code table is generated from some information like opcode, ModR/M, REP control signal, and each micro-code entry has organized as a linked-list pointer to support multiple micro-code instructions. The linked list part in the micro-code table indicates if the current micro-code is the last one of current x86 instruction. When the current micro-code is not the end current instruction, it will stop loading decoded information for new instruction to pipeline register but repeat the same instruction with its next piece of micro-code. To support REP variant, the microcode sequencer has another control input, REP_END. When this signal is low, the logic will repeat the current instruction and its micro-code until this signal is changed to 0, which indicates the ECX becomes to 0. Figure 4 shows the diagram of microcode sequencer. The index is 8 bit value which mainly comes from the opcode. Since there are some collisions between instructions, we also mapped some opcodes to another index, and it is done by exploiting information on reg/op field in ModR/M byte or 2-byte-opcode indicator.

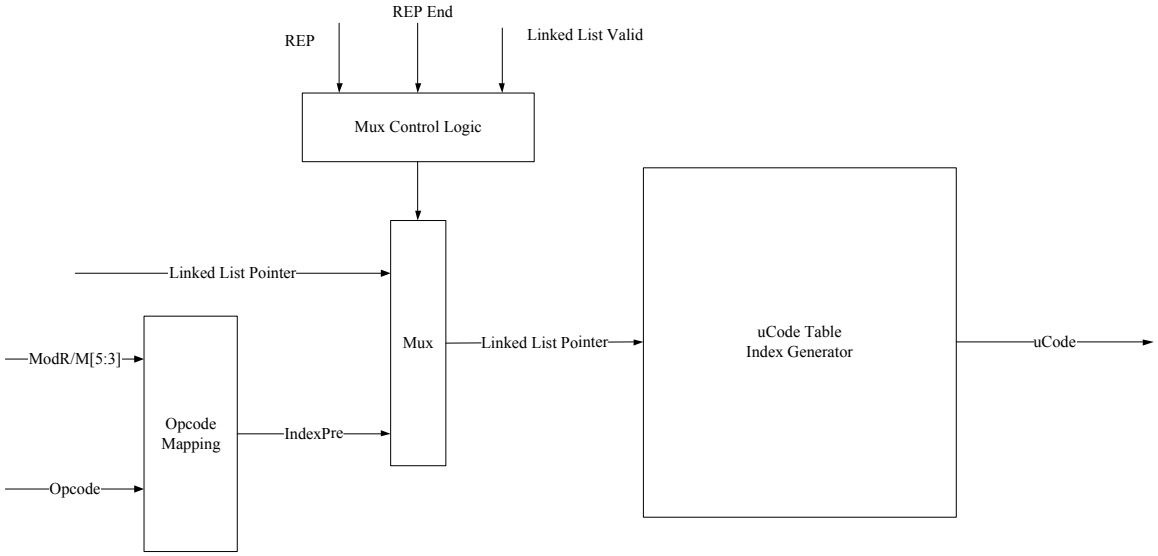

Figure 4 Block diagram of microcode sequencer

The good part of our decoder design is it is fully parallelized. We also pay much attention to minimize the logic delay by simplifying the logic because at the very beginning of the design we believed that this would be the critical path. From this parallel architecture and simplifying the logic at gate level, we minimize its critical path delay, which is roughly 8.3 ns (29 nand2$ + 3*mux4$ (0.5 ns) + ROM (1ns) = 8.3ns)

Those are all things the decode stage does and the overall architecture for Decode stage is schematic in Figure 5.

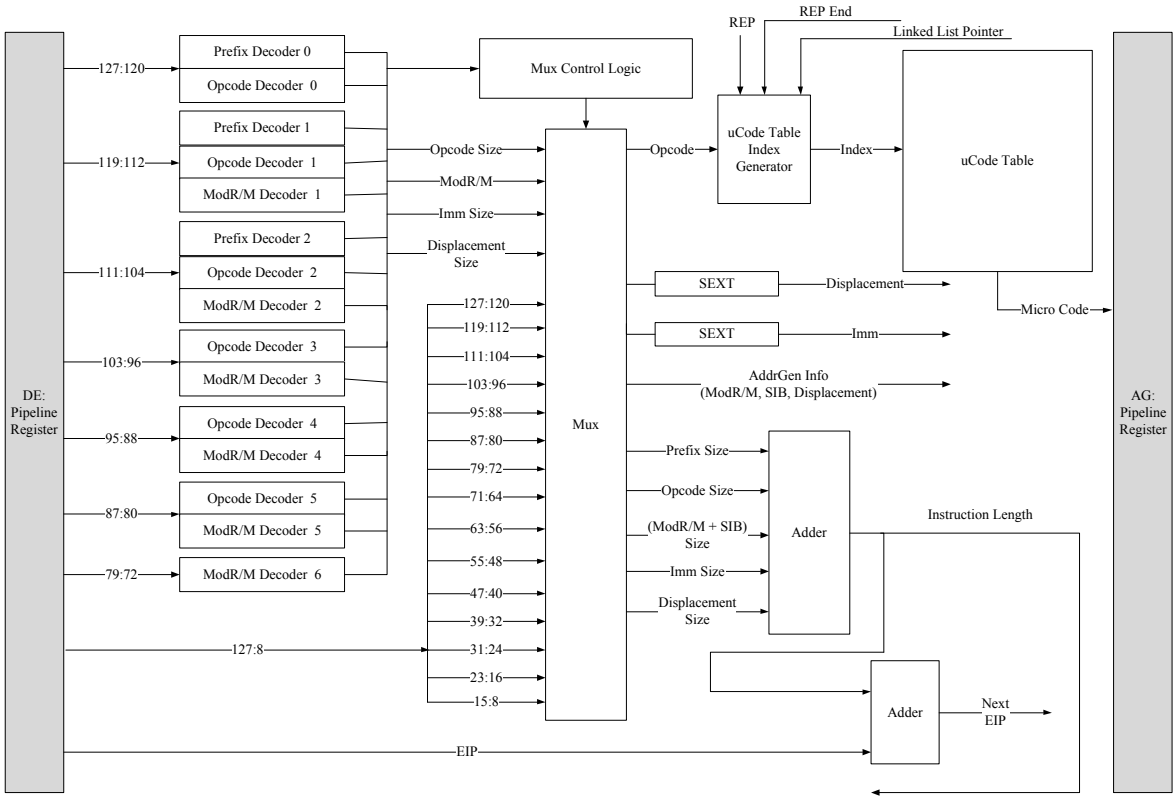

Figure 5 Block diagram of decode stage

5.1.3 Address generation

The AG stage is responsible to generate memory address by decoding the addressing mode information within a given instruction, in particular, ModR/M, SIB and displacement bytes. To generate address from addressing mode information, it needs to access general purpose register file. Thus, this stage is also responsible to access register files including general purpose registers (GPR), MMX registers (MMXR), and segment registers (SEGR), so AG stage also includes dependency checker logics for each the register file. The general purpose register file in our final design has 2 input ports and 3 write ports. With three write ports in the GPR, our processor can support REP MOVS, which needs update 3 registers, ESI, EDI, and ECX per every instruction, without stalls. Having more write ports increases the complexity of controller design, but in our design using the 3rd write port is restricted only for the special case like REP MOVS, so we use much simpler control logic for the 3rd write port and it helps to reduce the complexity while preserving correctness.

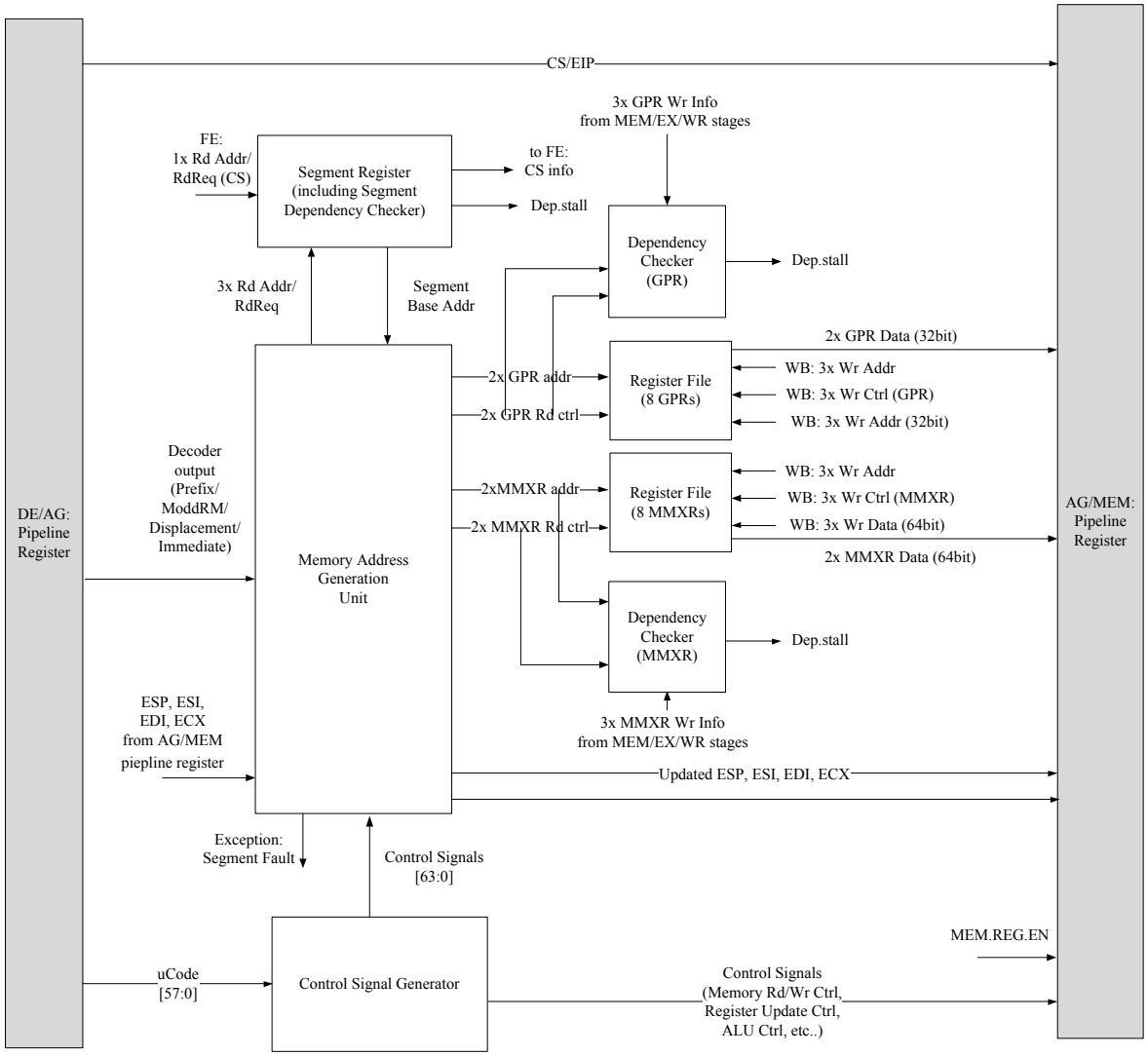

Figure 6 Block diagram of address generation stage

For reading values from the register file, we initially use the register file in a given library cells, which has only 2 read ports. Since we need to read two registers to get address when it uses SIB mode and one more register as operand, our register file needs to support more than two register values in a cycle. To overcome the limitation of a given register file in the library, we split the operation in a cycle into two sub-phases by the signal level of the clock. In the 1st half which the clock level is high, address generator requests registers that needed to generate memory address, and it requests up to two registers. After getting two register values in the 1st half, it starts to calculate memory address, which consists of sequential adders. In the 2nd half which the clock level is low, it requests registers as operands. Since we basically have combinational logic between pipeline registers, the register values that read in the 2nd half overwrite the value read from the 1st half, which are required to generate memory address. To resolve this, we exploit latch to capture the value while computing memory address. In addition to produce memory

address to get data to be processed, some instruction requires to produce secondary memory address, such as stack address and ES segment address to support string copy with MOVS instruction. Every case is supported by our two sub-phase structure while slightly increasing complexity of the access controller to register files.

AG stage is also responsible to generate signals to control the operation in the following stages, that is, MEM, EX, and WB stages. These signals are produced by analyzing the information from micro-code provided from DE stage, and addressing mode bytes, especially ModR/M and SIB bytes. This control signal is propagated to the following stages on every clock, and it is sometimes overridden in some conditions like CMOVC, MOVS, and CMPXCHG.

5.1.4. Execution

To design the execution logic to support a given instruction set, we first analyze the requirements of data processing for each instruction. We categorize most instructions into 8 categories, which are ADD/OR/NOT/SAL/SAR/SUB/DEC/MOV, and use other 8 cases to represent special instructions like CLD/STD/DAA/PADDD/PSUHFW/CALL/RET. We design 32-bit ALU logic and extend it to support 64-bit operation by adding one 32-bit adder and data shuffle logic. EFLAGS register is connected to the output of the 32-bit ALU because the instructions needs extra 32-bit adder or shuffler do not update EFLAGS register value. Even though the input operands of x86 instructions have lots of variants, we successfully organize them into two groups and the output of each group is connected to the input of the ALU logic.

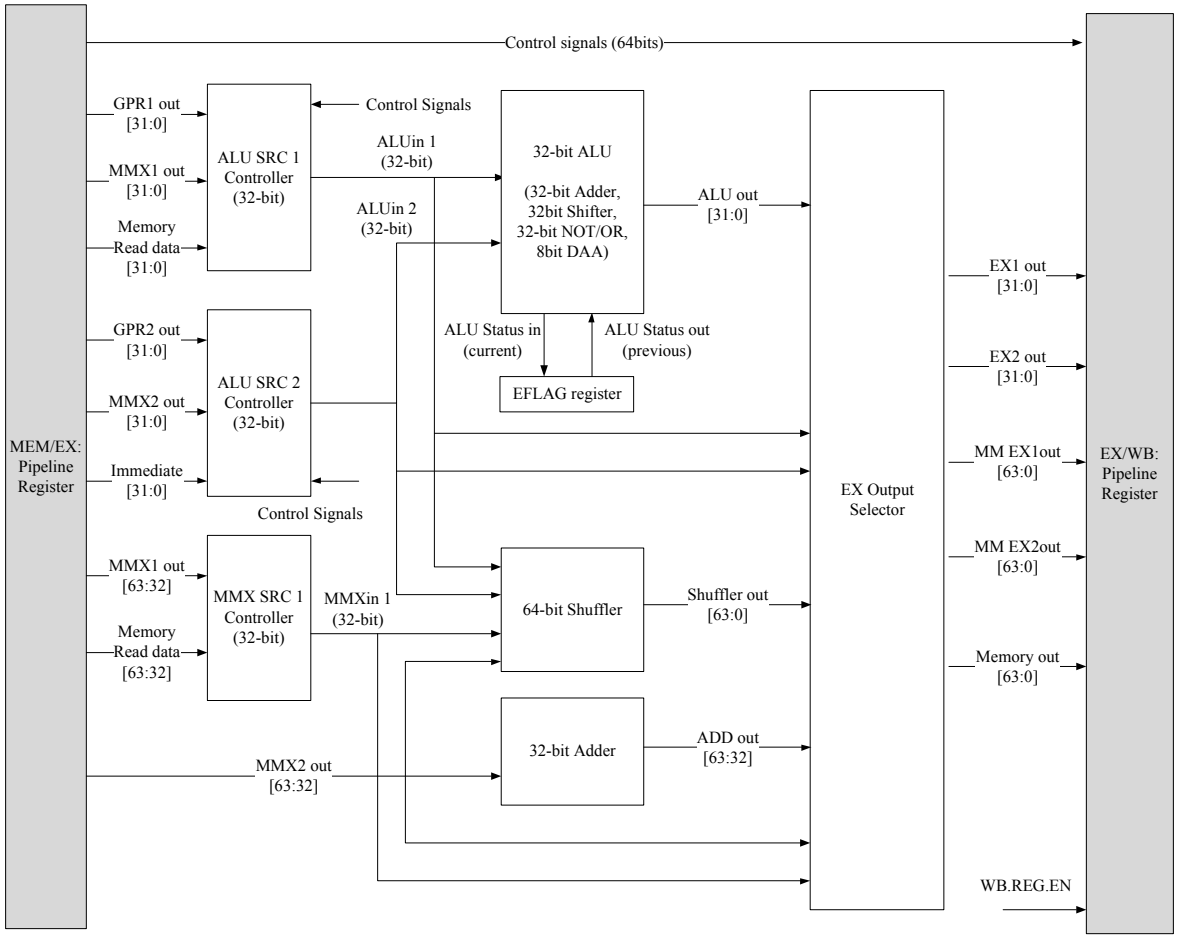

Figure 7 Block diagram of execution stage

Another part of the EX stage is for handling control flow instruction like JMP, CALL, and RET. For the control flow logic, we use 3 bits from micro-code to control it. In general, there are 7 types of jump instructions in our design, and their specifications (with according uCode) are listed below:

Table 1 Microcode for JMP instruction

Microcode	Description
000	not a jump
010	unconditional absolute far jump using address from memory (far return instruction)
011	unconditional absolute far jump using address in displacement and immediate
100	unconditional relative short jump
101	relative short jump when ZF = 0
110	unconditional absolute short jump using address from memory
111	relative short jump when ZF = 0 and CF = 0.

According to that information, we built the control flow logic which provides new EIP and CS as well as EIP write enable (Jump Taken) and CS write enable signal. When the enable signals are high, it has to change the control flow. In that case, it has to flush the pipeline before the EX stage. Its basic architecture is shown in Figure 8.

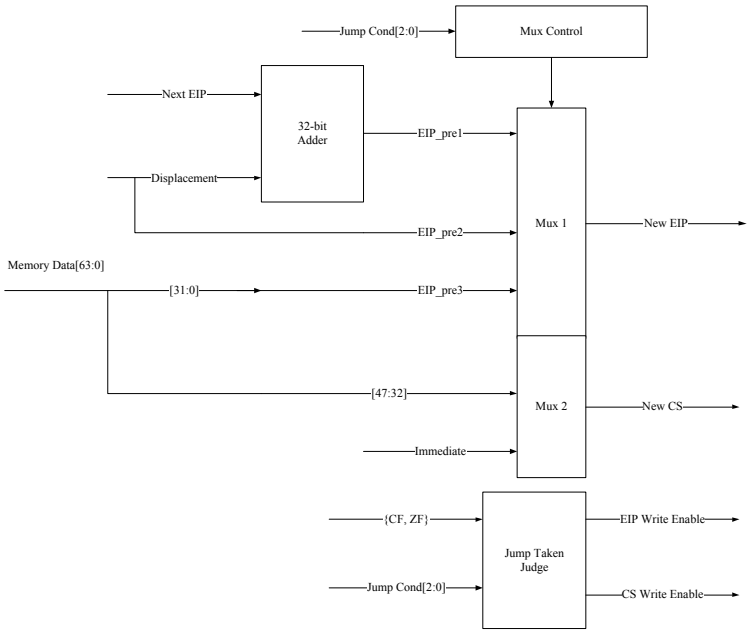

Figure 8 Block diagram of JMP logic

5.1.5. Write Back

Writing value to a register is done in a single cycle. However, a write operation to the memory may take longer based on the D-cache status. To reduce the stall from memory write request, we introduce a write buffer that has 8 entries. If there is an available space in the write buffer, the processor writes data into the write buffer instead of D-cache, and then continues executing the next instruction (non-blocking write operation). Later, if the write-allocate cache is able to receive that data, the data is popped from the write buffer and is written to the D-cache.

The memory dependency checker does the dependency checking for all the stages between the MEM stage to D-cache, which includes EX stage, WB stage, and Write Buffers. It also checks whether fetched data from main memory is actually written to the D-cache because the cache may not be updated with the new data yet after the write operation due to cache bypassing.

5.1.6. Instruction Cache

The instruction cache (I-cache) is implemented as 256 bytes, 2 way set associative cache. It has 2 banks and each line has 16 bytes width.

The 256 byte size derived from the cache size limitation in the project specification, which is 1024 bytes for I-cache and D-cache. We allocated 256 bytes for the I-cache, and the 768 bytes for the D-cache. The main design goal of the I-cache is to increase the hit rate and to decrease the latency.

To increase the hit rate, there are three major ways: increasing the size, increasing the associativity and improving the replacement policy. The cache size is limited by the project specification, and the associativity is 2, which is the maximum value achievable from the given SRAM library. In more details, our design has 256 bytes and 2 banks so that each bank can contain 128 bytes (256 bytes / 2 banks), which are 8 lines (128 bytes / 16 bytes per line). And the minimum row size of the SRAM in the library is 4 so we cannot have more than 2-way set associativity per bank. For the replacement policy, we use Least Recently Used (LRU) policy.

To decrease the latency, we use dual bank so each adjacent cache lines are on the different banks. The goal of this dual bank is to support unaligned requests efficiently. An unaligned request has its data spread over two cache lines (request size 16 bytes, line size 16 bytes). For an unaligned access, we fetch the first part of the data from bank0 and the other from bank 1 (or bank1 and bank0 respectively). If both banks are hit, the access time for the unaligned access is a single cycle. If only the first part is a hit and the second other is a miss, instruction cache provides the first part to FETCH stage in a hope that this partial data will help the FETCH stage to resume. In addition to that, the memory request for the second part is generated at the same cycle.

Another technique that we use to reduce latency is cache bypassing. When a new line is fetched from the main memory, the new line data is directly heading to FETCH stage without waiting to be stored into the cache line.

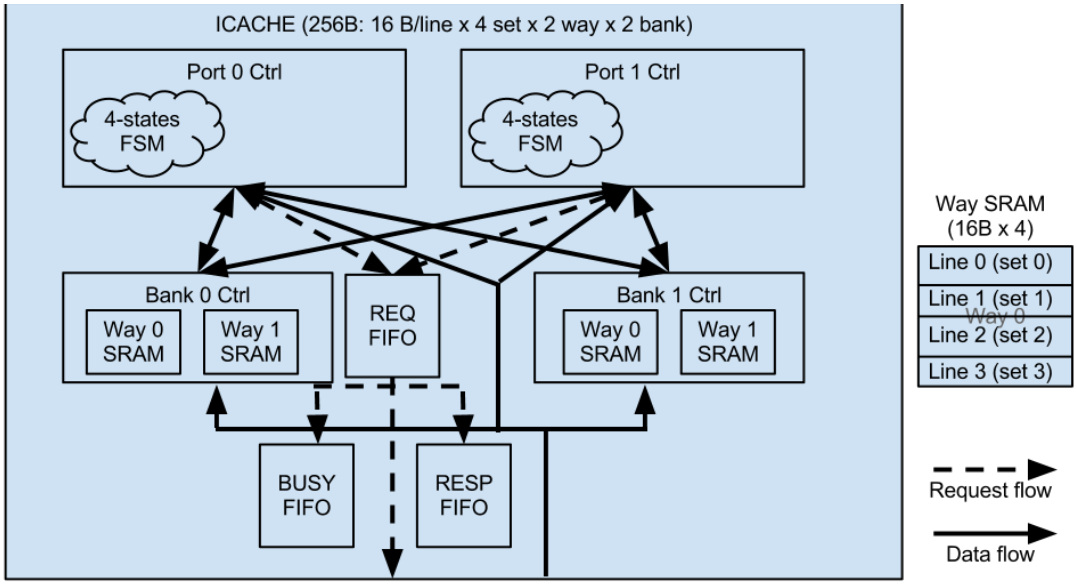

Figure 9 Instruction cache design

Outside the instruction cache, there is an instruction prefetcher. The instruction prefetcher monitors the address from FETCH stage and generates prefetching requests on the following cache lines (next two subsequent cache lines). It does not have internal data storage but its request will update the instruction cache with the following cache lines.

5.1.7. Data Cache

The Data cache (D-cache) is implemented as 768 bytes, 3 way set associative, write allocate cache. It has 4 banks and the line size is 16 bytes.

After allocating 256 bytes for the I-cache, the remaining 768 bytes are used for the D-cache. We allocated more size on D-cache because we assumed that the performance of test benches will depend on the D-cache much more.

Compared to the I-cache, we increase the bank number from 2 to 4. This is because there are more requesting stages for the D-cache. I-cache has a single requesting stage of FETCH (if we ignore the instruction prefetcher) whereas D-cache has two requesting stages: MEM stage (read) and WB stage (memory write). If we have more banks, there is less likelihood of a conflict between the ports so that we can provide the data for both stages at the same cycle. Also we increase the set associativity from 2 to 3, which is the maximum associativity achievable from the SRAM library (768 bytes for D-cache → 192 byte per bank (4 banks) → 12 line per bank (16 bytes per line)). We apply LRU replacement for the D-cache.

To decrease the latency, D-cache uses quad-bank so that each adjacent line is on different banks. Like the I-cache, if the data is spread over two lines (i.e. unaligned access), the two lines are accessed simultaneously from different banks so that access time becomes one cycle (if both banks are hit). Also we support cache bypassing to reduce the latency.

Another optimization that we have is wrapping address at line boundary. If the request address starts from the middle of a line, we send a memory request starting from the address, not from the cache line start address. The strength of this approach is the reduction in memory access latency. The data under interest is transferred at the first cycle (or first and second cycle if the data is spread over two 32 bits) so that cache bypassing can provide the data to the pipeline stages earlier (i.e. removing the trailing-edge effect). Once starting from the middle of the cache line the address should wrap around to the line start if it reaches the end of the line.

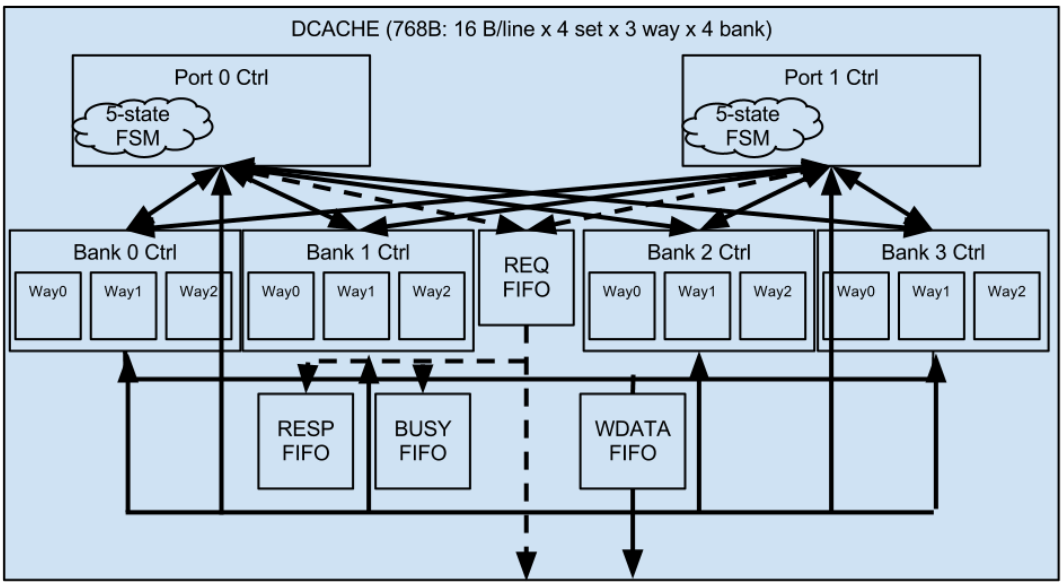

Figure 10 Data cache design

The address organization of D-cache is as follows. We implemented D-cache as a Virtually-Indexed-Physically-Tagged cache. It is virtually indexed because the cache index does not depend on the page number (address [31:12]). While the data / tag information is retrieved from SRAM using this index, the virtual address is translated into physical address by the Translation Look-aside Buffer (TLB) so that the output of the tag SRAM is compared with the translated physical address (physically tagged).

Table 2 Memory address format for data cache

[31:8]	[7:6]	[5:4]	[3:0]
Tag	SRAM index	bank	line offset

5.1.8. External Bus

We used shared data bus (32 bit) and shared address bus for the off-chip communication.

The control of the external bus is arbitrated by the bus arbiter and is composed of 3 phases. If a master wants to drive the external bus, it has to send a bus request to the arbiter and receive a bus grant from the arbiter first (arbitration phase). Once it received the grant for the bus, it drives the request information into the bus and receives an acknowledge signal (request phase). The slave which received the request during the request phase sends the read data into the data bus (data phase). The ownership of the data bus is also controlled by the bus arbiter. However there is no extra phase for the data bus control because the arbiter monitors the address bus and give the grant of data bus to the slave it is expected to response.

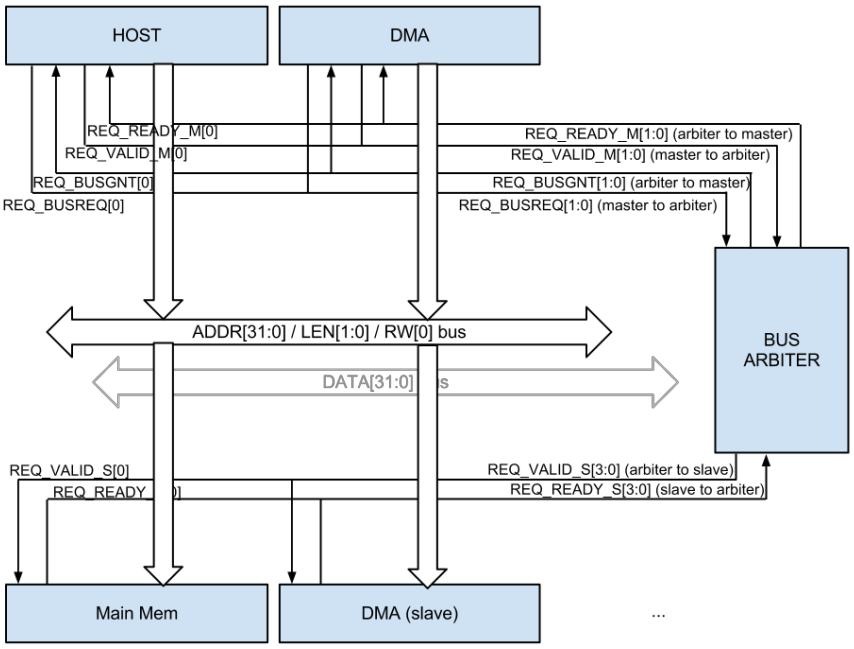

Figure 11 DMA operation (complicated I/O device)

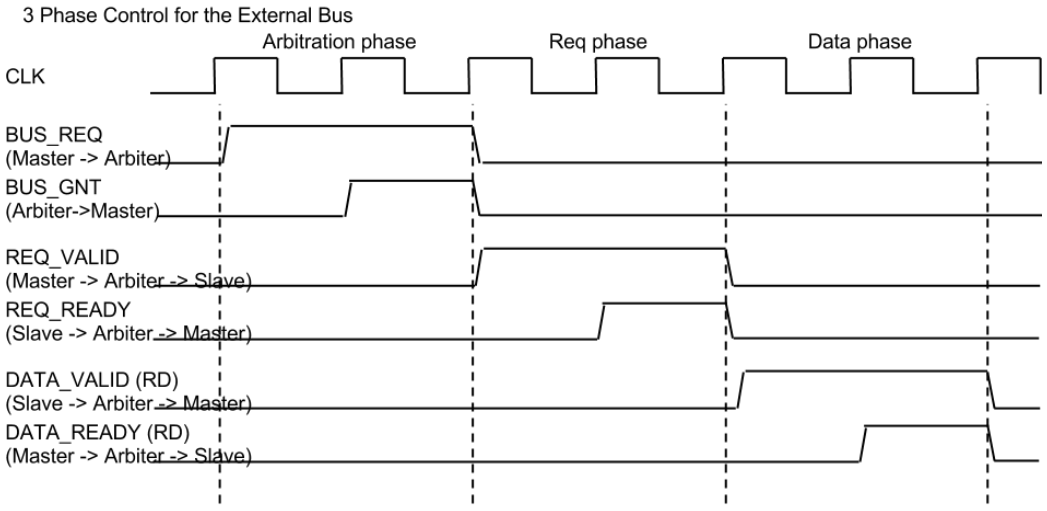

Figure 12 BUS signals to support DMA operation

To reduce the latency of external bus, we applied two optimizations. The first one is default master. When no master is requesting the bus, the arbiter gives the bus grant to the default master, which is CPU. Later, if the CPU wants to use the bus, it checks the bus grant signal. If it is already granted, it can skip the

arbitration phase and directly go to the request phase. The other optimization is using input buffers. Instead of generating acknowledge signal in the response to the request, acknowledge signal is high if there is a room in the input buffer. This approach can make the external bus interface to run with higher frequency even if we have some significant delays from the IO and PCB transmission lines (not modeled in this project).

5.1.9. Main Memory

The main memory has a size of 32KB and is configured as 128 bit x 2048 rows. It has 32 bit external data bus interface but it has 128 bit internal SRAM interface.

The reason for using the 128 bit internal SRAM is SRAM cycle time. After a write operation, SRAM has to wait 75 ns before another write operation. And this delay is critical when we are writing a cache line of 16 bytes into 32 bit (4 bytes) SRAM because we have to do 4 write operations. Instead of doing 4 separate write operations, we buffer the 32 bit data from the external bus into a write buffer to collect 128 bit data. And then we write this 128 bit data into the SRAM with a single write operation. This reduces the overhead of write operation significantly.

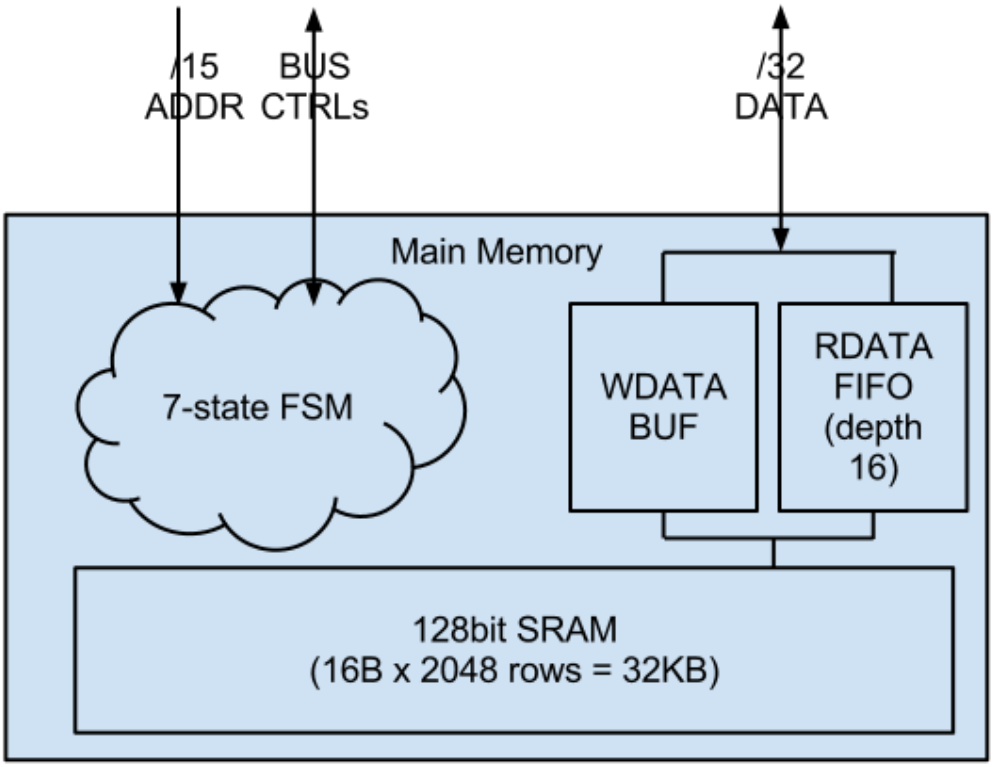

Figure 13 Block diagram of main memory

6. Conclusion

In this project, we have a good experience to design a microprocessor. Even though it eliminates many features in the real product, it still provides good lesson to understand the inside of the x86 processor and what is important to design an efficient device.

x86 architecture is one of the most popular architecture in the modern processor and has long history. Its architecture was built on 1970s and still maintaining major portion in microprocessor architecture. We believe that current form of x86 ISA is the results of long-time refinement of its predecessor. We expected to find something about these strong points, but unfortunately we could not. That is why we had hard time to design our decoder logic. Even though we had hard time to achieve the project goal, implementation of the x86 microprocessor, we are happy to finish our objective with good result.

We believe that the main reason for achieving good results is collaboration among team members. We actively discussed about the design point to enhance our processor design while having some difficulty due to conflict of each one's opinion, but it absolutely helps to improve our design. Especially we are proud of having good design of memory system, which was key factor to enhance the performance.

However, critical path is one thing we missed while implementing our processor. We had a hard time to fix the design of decoding part. It was started from 2 separated stages, but becomes collapsed into one stage for some reason, and then getting bigger while we finalize our design. Wrong decision about register file access is one of the important factors to mitigate our progress. Since we stuck in 2-port register file, it limits a lot in decoding stage. Also, it incurs lot of controller overhead. Even we noticed that issue later, it might be too late to recover the previous approach. It results in a little bit longer clock cycle time. We believe it gave us a good lesson: understands a given environment and constraint is very important to start a project.

7. Appendix

7.1. Microinstruction format, microcode listing.

- Attached as separate document.