Pentium Pro ISS modeling in C++

Amirhosein Yavarikhoo 810199514

September 2023

Faculty of Electrical and Computer Engineering

University of Tehran

Table of contents

1- Introduction

The Sixth-Generation Pentium Pro (P6) Processor is an x86 series microprocessor developed by Intel and introduced in November 1995. This single-core processor featured a higher transistor count and a newer microarchitecture compared to its predecessors. Improvements included enhanced pipelining and the utilization of techniques like register renaming. The Pentium Pro was primarily used in servers and high-performance computing systems, notably in the ASCI Red supercomputer.

The objective of this project is to create a system-level model of this processor using the SystemC library within the C++ programming language. This approach involves developing a high-level, software-based description of the hardware to facilitate a deeper understanding of its operation. By creating a software model, potential challenges for the hardware design team can be identified, offering new perspectives on problem-solving. Ultimately, the modeled processor will be capable of interfacing with a PCH unit, and its software modeling will allow for extensive testing and improved coordination between the design teams of these two components.

Given the sequential nature of programming languages, a platform capable of supporting the capabilities of Hardware Description Languages (HDLs) is required. The SystemC library provides the necessary framework for parallel and concurrent code execution. This report details the various stages of the system-level processor design and the associated C++ code.

2- CPU Architecture

To model a processor, one must first have a precise understanding of its structure. This structure includes system registers, processor-bus interactions, and more. *Note: This report assumes that the processor is operating in 32-bit mode.*

2-1- Registers

In the 32-bit architecture of Intel processors (IA-32), there are 16 registers used for executing programs and system activities. These registers are divided into four types.

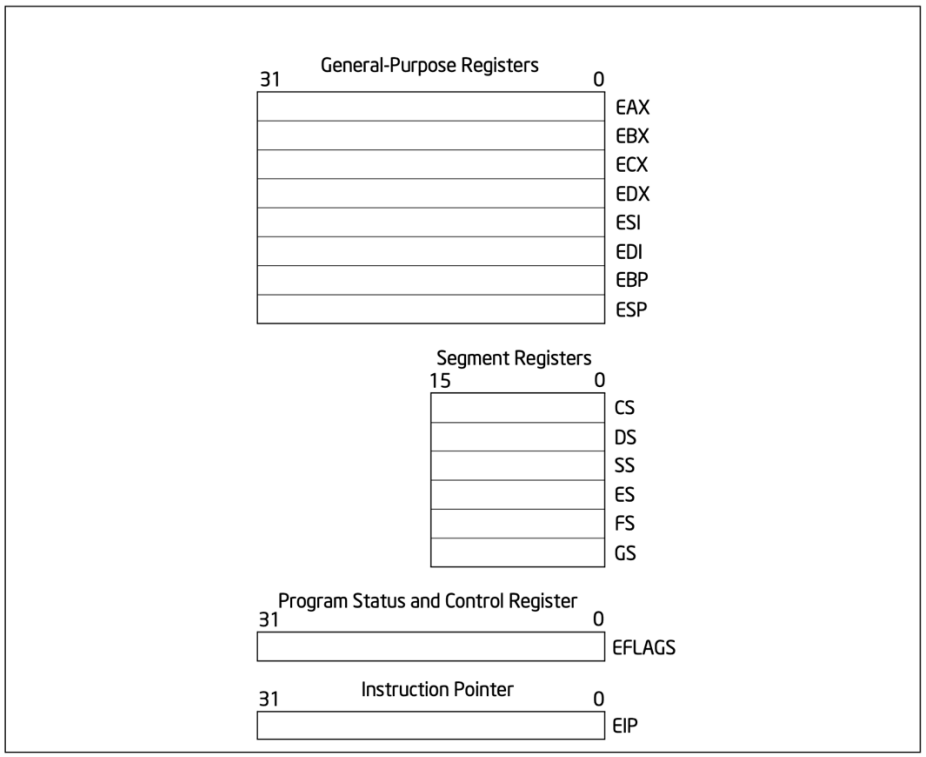


Figure 2-1-registers Pentium Pro

2-2- General-Purpose Registers

Eight 32-bit registers store operands and pointers. As shown in Figure 2-2, in different processor modes, the slices of each register will have new names. However, as mentioned earlier, we will focus on the 32-bit mode.

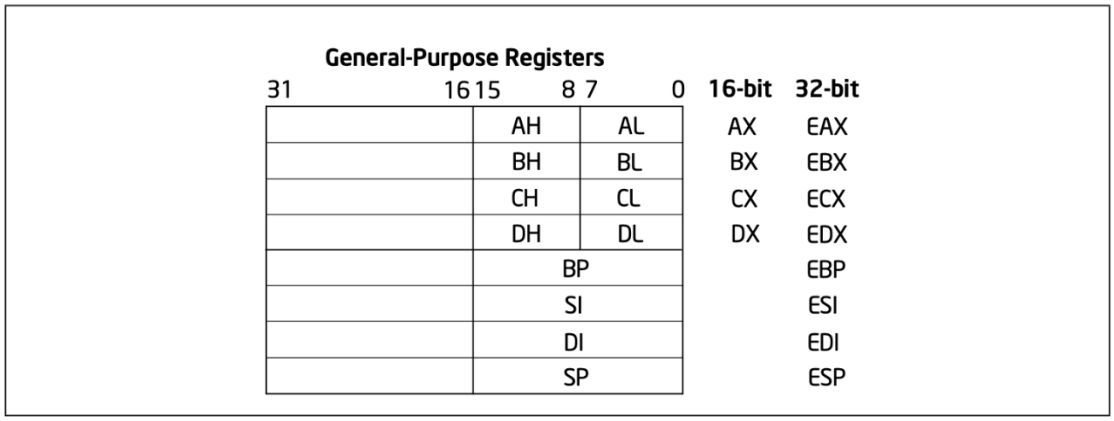


Figure 2-2 General-Purpose Registers

2-3- Segment Registers

Six 16-bit registers store segment selectors. These registers are used for addressing different memory segments.

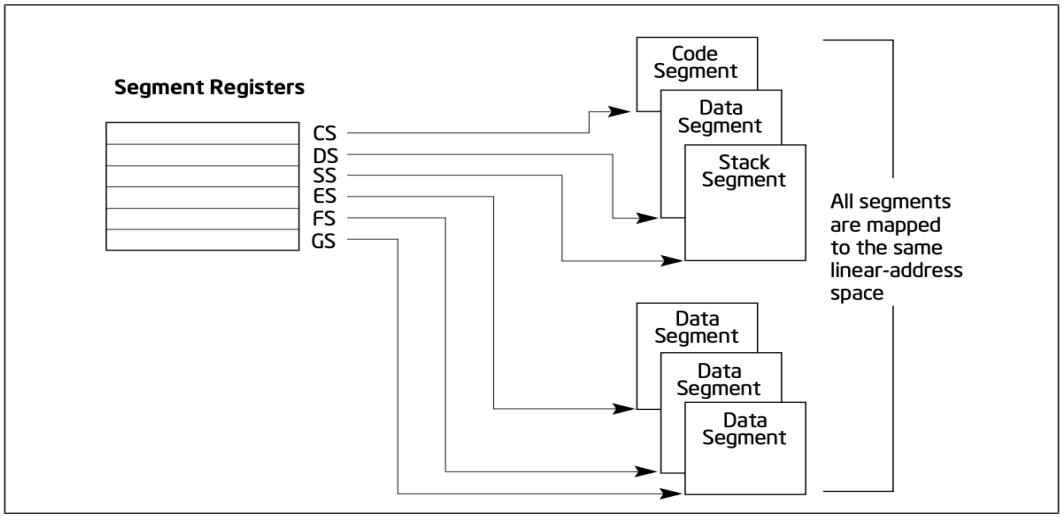


Figure 2-3- Segment Registers

2-4- EFLAGS Register

One 32-bit register in which the necessary system flags are stored. This register initially has the value 00000002H. Bits 1, 3, 5, 15, and 22 to 31 are pre-allocated and should not be modified by software.

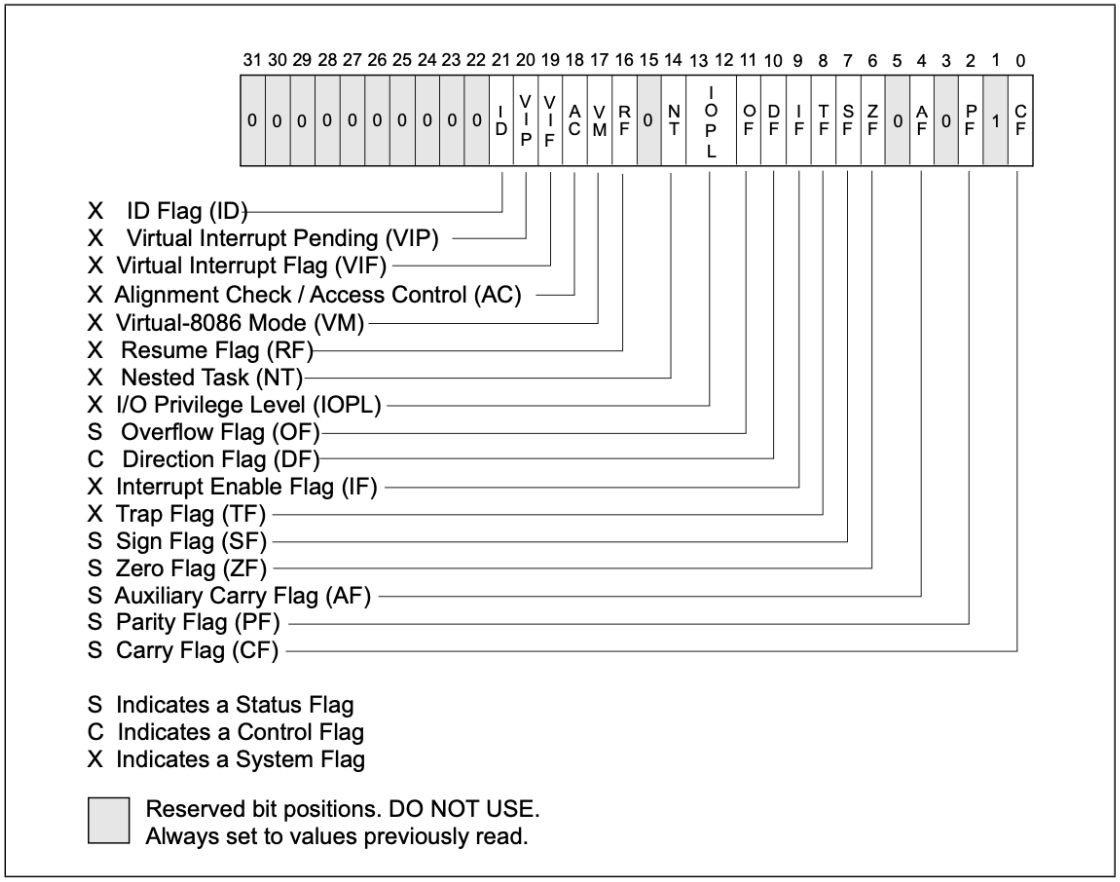


Figure 2-4- EFLAGS Register

2-5- EIP Register

A 32-bit register that stores the pointer to the next instruction. This register is not directly controlled by software; instead, it is implicitly controlled by control transfer instructions, interrupts, and exceptions.

3- Instruction Set

Given that the Pentium Pro processor is a type of CISC processor, its instruction lengths can vary from 1 to 15 bytes. Each instruction can include 1 to 6 main sections, with the presence of the Opcode section being mandatory for all instructions.

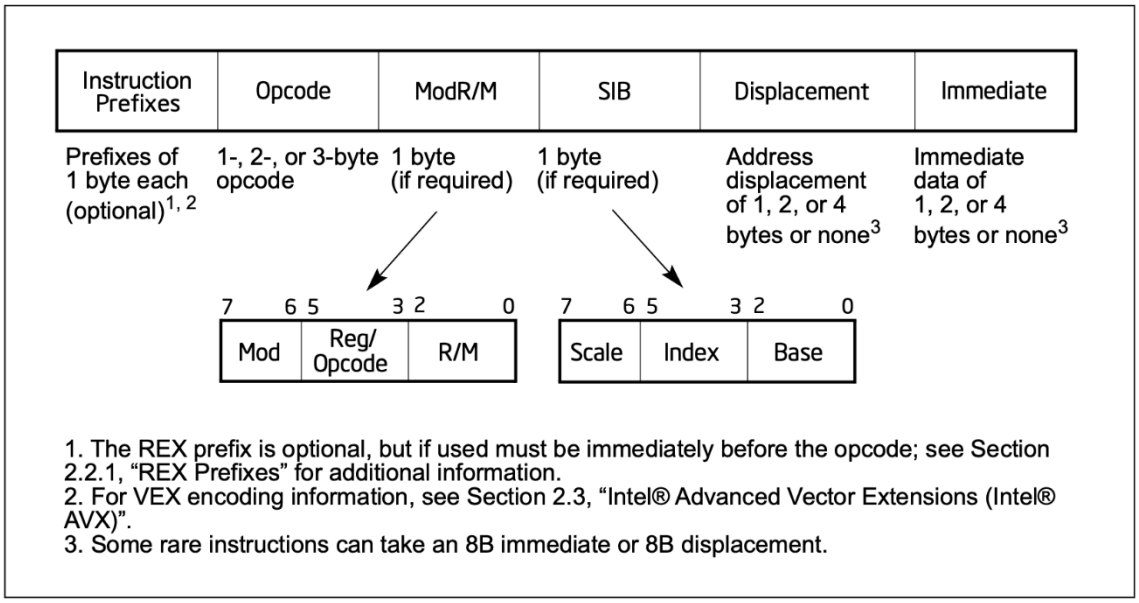


Figure 3-1- Instruction format

3-1- Instruction Prefixes

The first part of an instruction can be a prefix. Prefixes include four groups, which we will not implement at this stage of the project.

3-2- Opcode

The Opcode section exists in all instructions and can range from 1 to 3 bytes. As shown in Figure 3-2-1, bits 5 to 3 of the ModR/M section can be used as auxiliary bits of the Opcode section. In Figure 3-2-1, one-byte Opcodes are presented, with 4 bits representing the high-value Opcode of the row in the table and 4 bits representing the low-value Opcode of the

column in the table. The red square in the table indicates the Opcode, which, when observed, indicates that the length of the Opcode will be more than one byte.

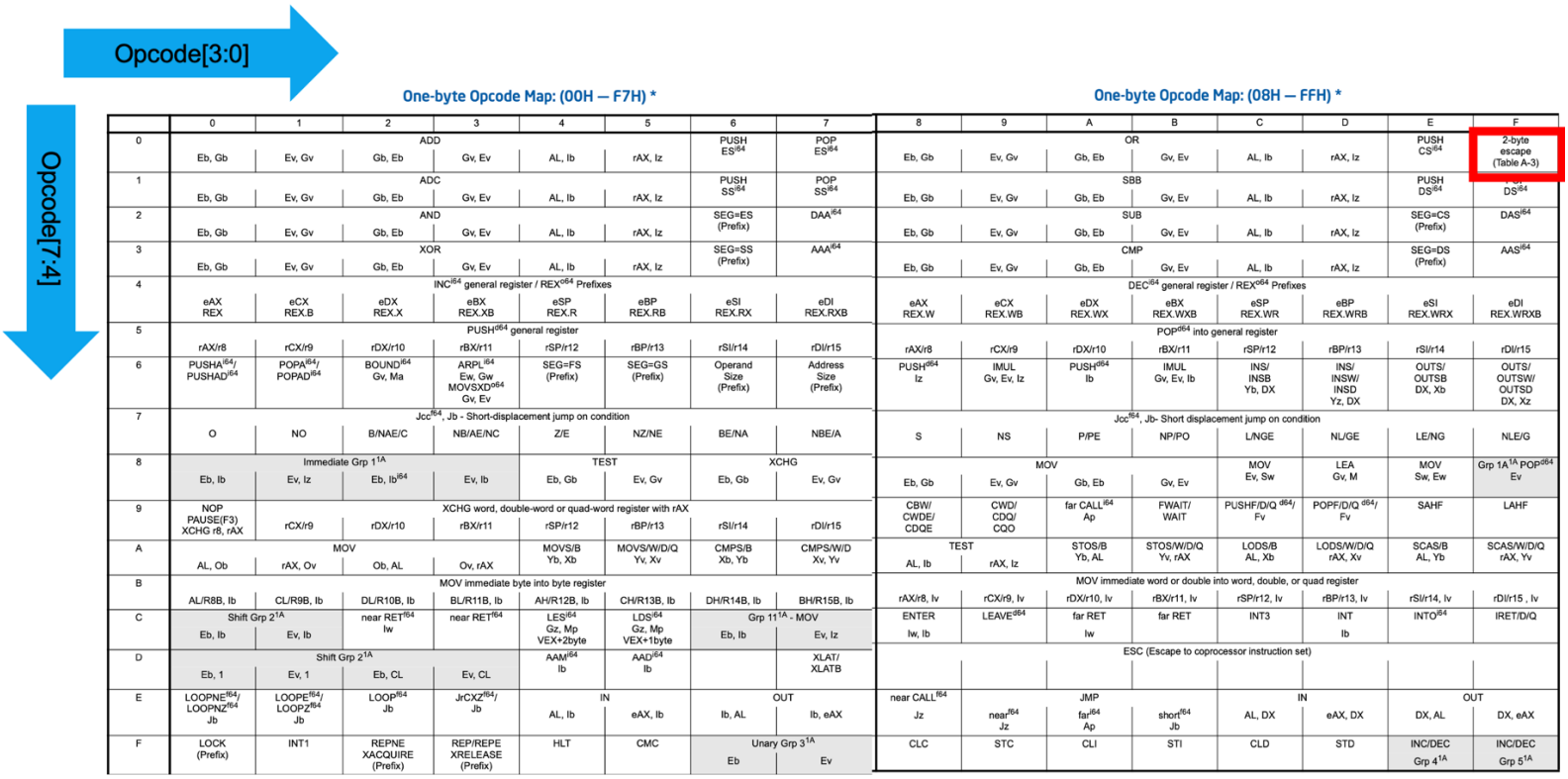
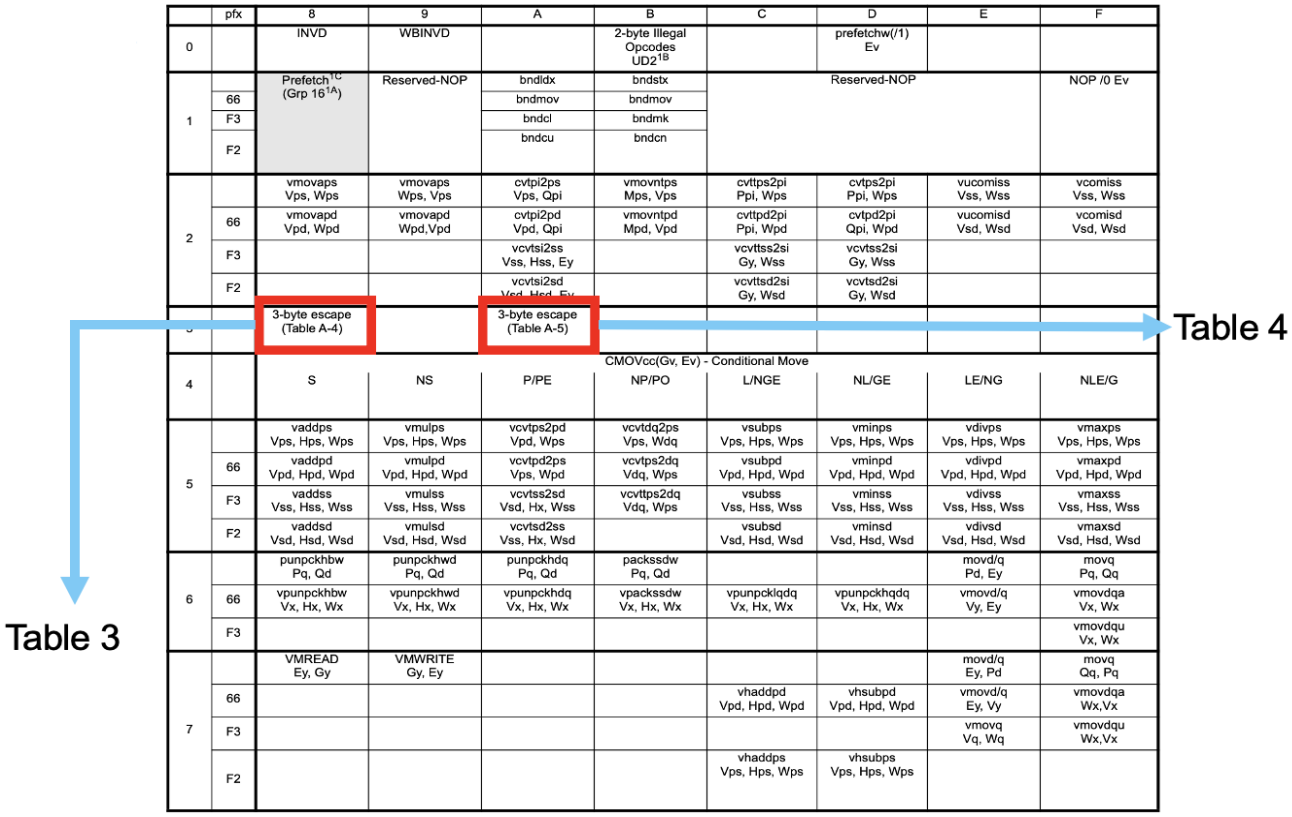


Figure3-2-1 1-byte opcodes

Figure 3-2-2 shows a portion of the 2-byte Opcode table, where, like in Figure 3-2-1, the red squares indicate a transition to a table with Opcodes of greater byte size.



	pfx	8	9	A	B	C	D	E	F
0		INVD	WBINVD		2-byte Illegal Opcodes UD2[1B]		prefetchw(/1) Ev		
1		Prefetch[1C] (Grp 16[1A])	Reserved-NOP	bndldx	bndstx	Reserved-NOP			NOP /0 Ev
	66			bndmov	bndmov				
	F3			bndcl	bndmk				
	F2			bndcu	bndcn				
2		vmovaps Vps, Wps	vmovaps Wps, Vps	cvtpi2ps Vps, Qpi	vmovntps Mps, Vps	cvttps2pi Ppi, Wps	cvtps2pi Ppi, Wps	vucomiss Vss, Wss	vcomiss Vss, Wss
	66	vmovapd Vpd, Wpd	vmovapd Wpd,Vpd	cvtpi2pd Vpd, Qpi	vmovntpd Mpd, Vpd	cvttpd2pi Ppi, Wpd	cvtpd2pi Qpi, Wpd	vucomisd Vsd, Wsd	vcomisd Vsd, Wsd
	F3			vcvtsi2ss Vss, Hss, Ey		vcvttss2si Gy, Wss	vcvtss2si Gy, Wss		
	F2			vcvtsi2sd Vsd, Hsd, Ey		vcvttsd2si Gy, Wsd	vcvtsd2si Gy, Wsd		
3		3-byte escape (Table A-4)		3-byte escape (Table A-5)					
4		CMOVcc(Gv, Ev) - Conditional Move							
		S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
5		vaddps Vps, Hps, Wps	vmulps Vps, Hps, Wps	vcvtps2pd Vpd, Wps	vcvtdq2ps Vps, Wdq	vsubps Vps, Hps, Wps	vminps Vps, Hps, Wps	vdivps Vps, Hps, Wps	vmaxps Vps, Hps, Wps
	66	vaddpd Vpd, Hpd, Wpd	vmulpd Vpd, Hpd, Wpd	vcvtpd2ps Vps, Wpd	vcvtps2dq Vdq, Wps	vsubpd Vpd, Hpd, Wpd	vminpd Vpd, Hpd, Wpd	vdivpd Vpd, Hpd, Wpd	vmaxpd Vpd, Hpd, Wpd
	F3	vaddss Vss, Hss, Wss	vmulss Vss, Hss, Wss	vcvtss2sd Vsd, Hx, Wss	vcvttps2dq Vdq, Wps	vsubss Vss, Hss, Wss	vminss Vss, Hss, Wss	vdivss Vss, Hss, Wss	vmaxss Vss, Hss, Wss
	F2	vaddsd Vsd, Hsd, Wsd	vmulsd Vsd, Hsd, Wsd	vcvtsd2ss Vss, Hx, Wsd		vsubsd Vsd, Hsd, Wsd	vminsd Vsd, Hsd, Wsd	vdivsd Vsd, Hsd, Wsd	vmaxsd Vsd, Hsd, Wsd
6		punpckhbw Pq, Qd	punpckhwd Pq, Qd	punpckhdq Pq, Qd	packssdw Pq, Qd			movd/q Pd, Ey	movq Pq, Qq
	66	vpunpckhbw Vx, Hx, Wx	vpunpckhwd Vx, Hx, Wx	vpunpckhdq Vx, Hx, Wx	vpackssdw Vx, Hx, Wx	vpunpcklqdq Vx, Hx, Wx	vpunpckhqdq Vx, Hx, Wx	vmovd/q Vy, Ey	vmovdqa Vx, Wx
	F3								vmovdqu Vx, Wx
7		VMREAD Ey, Gy	VMWRITE Gy, Ey					movd/q Ey, Pd	movq Qq, Pq
	66					vhaddpd Vpd, Hpd, Wpd	vhsubpd Vpd, Hpd, Wpd	vmovd/q Ey, Vy	vmovdqa Wx,Vx
	F3							vmovq Vq, Wq	vmovdqu Wx,Vx
	F2					vhaddps Vps, Hps, Wps	vhsubps Vps, Hps, Wps		

Figure 3-2-2 2-byte opcodes

3-3- SIB and ModR/M

The third part of an instruction can be the ModR/M section, which itself consists of three parts, as shown in Figure 3-3-1. We extract the operands of an instruction from this section (if necessary, we will also use the parts explained later). As mentioned in the Opcode section, bits 5 to 3 can either determine the register or be part of the Opcode.

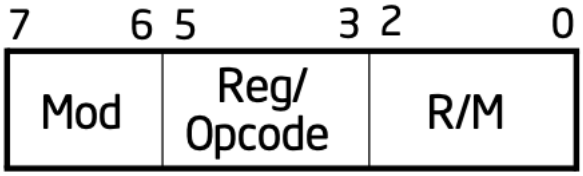


Figure 3-3-1 ModR/M placement

Figure 3-3-2 shows the method of determining the first and second operands of instruction operations. In this way, based on the values of the Mod, Reg, and R/M sections, the first operand is identified in the Effective Address column, and the second operand is specified in the r32 (/r) row.

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) (In decimal) /digit (Opcode) (In binary) REG =			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX] [ECX] [EDX] [EBX] [--][--][1] disp32[2] [ESI] [EDI]	00	000 001 010 011 100 101 110 111	00 01 02 03 04 05 06 07	08 09 0A 0B 0C 0D 0E 0F	10 11 12 13 14 15 16 17	18 19 1A 1B 1C 1D 1E 1F	20 21 22 23 24 25 26 27	28 29 2A 2B 2C 2D 2E 2F	30 31 32 33 34 35 36 37	38 39 3A 3B 3C 3D 3E 3F
[EAX]+disp8[3] [ECX]+disp8 [EDX]+disp8 [EBX]+disp8 [--][--]+disp8 [EBP]+disp8 [ESI]+disp8 [EDI]+disp8	01	000 001 010 011 100 101 110 111	40 41 42 43 44 45 46 47	48 49 4A 4B 4C 4D 4E 4F	50 51 52 53 54 55 56 57	58 59 5A 5B 5C 5D 5E 5F	60 61 62 63 64 65 66 67	68 69 6A 6B 6C 6D 6E 6F	70 71 72 73 74 75 76 77	78 79 7A 7B 7C 7D 7E 7F
[EAX]+disp32 [ECX]+disp32 [EDX]+disp32 [EBX]+disp32 [--][--]+disp32 [EBP]+disp32 [ESI]+disp32 [EDI]+disp32	10	000 001 010 011 100 101 110 111	80 81 82 83 84 85 86 87	88 89 8A 8B 8C 8D 8E 8F	90 91 92 93 94 95 96 97	98 99 9A 9B 9C 9D 9E 9F	A0 A1 A2 A3 A4 A5 A6 A7	A8 A9 AA AB AC AD AE AF	B0 B1 B2 B3 B4 B5 B6 B7	B8 B9 BA BB BC BD BE BF
EAX/AX/AL/MM0/XMM0 ECX/CX/CL/MM/XMM1 EDX/DX/DL/MM2/XMM2 EBX/BX/BL/MM3/XMM3 ESP/SP/AH/MM4/XMM4 EBP/BP/CH/MM5/XMM5 ESI/SI/DH/MM6/XMM6 EDI/DI/BH/MM7/XMM7	11	000 001 010 011 100 101 110 111	C0 C1 C2 C3 C4 C5 C6 C7	C8 C9 CA CB CC CD CE CF	D0 D1 D2 D3 D4 D5 D6 D7	D8 D9 DA DB DC DD DE DF	E0 E1 E2 E3 E4 E5 E6 E7	E8 E9 EA EB EC ED EE EF	F0 F1 F2 F3 F4 F5 F6 F7	F8 F9 FA FB FC FD FE FF

Figure 3-3-2 finding effective address based on Mod R/M

Some rows in the table in Figure 3-5 refer to a new table, which is actually the SIB byte that specifies the corresponding column and row in the second table (Figure 3-3-3).

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) (In decimal) /digit (Opcode) (In binary) REG =			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--][1]		100	04	0C	14	1C	24	2C	34	3C
disp32[2]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8[3]	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[--][--]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[--][--]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

SIB

r32 (In decimal) Base = (In binary) Base =			EAX 0 000	ECX 1 001	EDX 2 010	EBX 3 011	ESP 4 100	[*] 5 101	ESI 6 110	EDI 7 111
Scaled Index	**SS**	**Index**	**Value of SIB Byte (in Hexadecimal)**							
[EAX]	00	000	00	01	02	03	04	05	06	07
[ECX]		001	08	09	0A	0B	0C	0D	0E	0F
[EDX]		010	10	11	12	13	14	15	16	17
[EBX]		011	18	19	1A	1B	1C	1D	1E	1F
none		100	20	21	22	23	24	25	26	27
[EBP]		101	28	29	2A	2B	2C	2D	2E	2F
[ESI]		110	30	31	32	33	34	35	36	37
[EDI]		111	38	39	3A	3B	3C	3D	3E	3F
[EAX*2]	01	000	40	41	42	43	44	45	46	47
[ECX*2]		001	48	49	4A	4B	4C	4D	4E	4F
[EDX*2]		010	50	51	52	53	54	55	56	57
[EBX*2]		011	58	59	5A	5B	5C	5D	5E	5F
none		100	60	61	62	63	64	65	66	67
[EBP*2]		101	68	69	6A	6B	6C	6D	6E	6F
[ESI*2]		110	70	71	72	73	74	75	76	77
[EDI*2]		111	78	79	7A	7B	7C	7D	7E	7F
[EAX*4]	10	000	80	81	82	83	84	85	86	87
[ECX*4]		001	88	89	8A	8B	8C	8D	8E	8F
[EDX*4]		010	90	91	92	93	94	95	96	97
[EBX*4]		011	98	99	9A	9B	9C	9D	9E	9F
none		100	A0	A1	A2	A3	A4	A5	A6	A7
[EBP*4]		101	A8	A9	AA	AB	AC	AD	AE	AF
[ESI*4]		110	B0	B1	B2	B3	B4	B5	B6	B7
[EDI*4]		111	B8	B9	BA	BB	BC	BD	BE	BF
[EAX*8]	11	000	C0	C1	C2	C3	C4	C5	C6	C7
[ECX*8]		001	C8	C9	CA	CB	CC	CD	CE	CF
[EDX*8]		010	D0	D1	D2	D3	D4	D5	D6	D7
[EBX*8]		011	D8	D9	DA	DB	DC	DD	DE	DF
none		100	E0	E1	E2	E3	E4	E5	E6	E7
[EBP*8]		101	E8	E9	EA	EB	EC	ED	EE	EF
[ESI*8]		110	F0	F1	F2	F3	F4	F5	F6	F7
[EDI*8]		111	F8	F9	FA	FB	FC	FD	FE	FF

Figure 3-3-3 Mod R/M table

The SIB section also consists of three parts, which are shown in Figure 3-3-4. Based on the values of Scale, Index, and Base, a cell from the right table in Figure 3-6 is selected, which provides the necessary values for memory access.

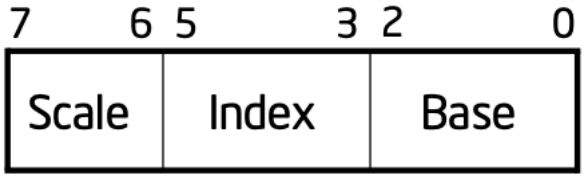


Figure 3-3-4 SIB placement

3-4- Immediate Placement

As seen in the table in Figure 3-5, the Displacement section is used in some memory addresses, and its value is used directly. Additionally, sometimes one of the operands of the instruction is located in the Immediate section, and we execute the instruction by reading it directly.

4- Software Model

In this section, the structure discussed in Section 2, along with other items, must be implemented.

4-1- Code Architecture

Since there are multiple files in this project, explaining the code hierarchy is essential. The processor is intended to operate alongside a memory unit. Therefore, the processor and the memory unit together form the entire system (File: *FullSystem.h*). The memory unit is simply an array of 2000 32-bit cells. Since, in a real system, the processor must ensure that the memory is ready for communication, a ready signal has been assigned for the memory, which is simply set manually to always be 1, meaning the memory is always ready to communicate with the processor. In the file main.cpp, the simulation duration, the creation of the vcd file, and monitoring the communication signals between the processor and memory are configured. The code hierarchy from the highest level to the lowest is as follows:

1. Test bench execution file and its settings (*Main.cpp*)
2. Overall system file comprising the processor and memory (*FullSystem.h*)
3. Main processor file (*ISS.h*) and main memory file (*Memory.h*)
4. Instruction execution file (*BaseFunctions.h*)
5. Library file and some general functions common between components (*Requirements.h*)

In the header files, only the function names are mentioned, while the function descriptions are included in the *.cpp* files. In addition to the above files, there is a separate file for testing memory named *MemoryTB*. To compile the files, the *makefile* provided in the folder is used.

4-1-1 Modeling the Processor Structure

In the first phase, we will model the required ports, registers, and input/output signals. The mentioned registers include the main system registers (as shown in Figure 2-1) and the registers needed for temporarily storing variables (in the file *ISS.h*).

```
11      //----------------------------------PORTS---------------------------------
12      sc_in<sc_logic> clk;
13      sc_out<sc_lv<ADDRESS_SIZE>> AddressBus;
14      sc_in<sc_lv<DATA_SIZE>> DataBus_in;
15      sc_out<sc_lv<DATA_SIZE>> DataBus_out;
16      sc_out<sc_logic> read_request, write_request, input_valid;
17      sc_in<sc_logic> mem_ready;
18      //-------------------------------INTERNAL REGISTERS-------------------------------
19      sc_lv<REGISTER_WIDTH> EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP;   // General-Purpose Register
20      sc_lv<SEGMENT_REGISTER_WIDTH> CS, DS, SS, ES, FS, GS;   // Segment Registers
21      sc_lv<EFLAG_WIDTH> EFLAGS;  // EFLAGS Register
22      sc_lv<EFLAG_WIDTH> *EFLAGS_ptr;
23      sc_lv<EIP_WIDTH> EIP;   //EIP Register
24      sc_lv<MAX_INST_WIDTH> Instruction;
25      sc_lv<8> disp8;
26      sc_lv<32> disp32;
27      sc_lv<8> SIB;
28      sc_lv<ADDRESS_SIZE> effective_address;
29      sc_lv<32> memory_write_data;
30      sc_lv<32> memory_read_data;
```

Figure 4-1-1 Register Modeling

4-1-2 Execution Unit Modeling

In this stage, we need to model the Opcode table, specifically the table shown in Figure 3-2-1. For each cell in the table, we will write a function with the format:

“void T#table number_#row number#column number_instruction name”

Within each function, we will implement the specific actions that the corresponding instruction performs. For example, the function *T1_8B_MOV_DATA* indicates that in the table shown in Figure 4-1-2, at row 8 and column 11, the instruction *MOV_DATA* exists, and its function has been defined accordingly.

```
185     //-------------------------------------MOVE----------------------
186     void T1_89_MOV_DATA(vector<sc_lv<8>> Instruction_Bytes);
187     void T1_8B_MOV_DATA(vector<sc_lv<8>> Instruction_Bytes);
188     void T1_C7_MOV_DATA(vector<sc_lv<8>> Instruction_Bytes);
189     void T1_A1_MOV_DATA(vector<sc_lv<8>> Instruction_Bytes);
190     void T1_A3_MOV_DATA(vector<sc_lv<8>> Instruction_Bytes);
```

Figure 4-1-2 functions MOV_DATA

```
1746  template <int ADDRESS_SIZE, int DATA_SIZE, int REGISTER_WIDTH, int SEGMENT_REGISTER_WIDTH, int EFLAG_WIDTH, int EIP_WI
1747  void CPU_ISS<ADDRESS_SIZE, DATA_SIZE, REGISTER_WIDTH, SEGMENT_REGISTER_WIDTH, EFLAG_WIDTH, EIP_WIDTH, MAX_INST_WIDTH>:
1748      T1_A3_MOV_DATA(vector<sc_lv<8>> Instruction_Bytes)
1749  {
1750      int mod_location = 1;
1751      sc_lv<8> ModRM_Byte = Instruction_Bytes[mod_location];
1752      sc_lv<REGISTER_WIDTH> *EAX_;
1753      EAX_ = &EAX;
1754      load_disp32(Instruction_Bytes, 1);
1755      effective_address = disp32;
1756      memory_write_data = EAX;
1757      cout << "address is: " << endl;
1758      write_request = SC_LOGIC_1;
1759      wait(write_done);
1760      Instruction_done.notify();
1761  }
```

Figure 4-1-3function T1_A3_MOV_DATA

All of these functions, defined as **FunctionPtr*, will be placed in a two-dimensional array named opcode_1B[16][16] under the table1 function (Figure 4-1-4). (File: *ISS.h*)

```
568  template <int ADDRESS_SIZE, int DATA_SIZE, int REGISTER_WIDTH, int SEGMENT_REGISTER_WIDTH, int EFLAG_WIDTH, int EIP_WIDTH, int MAX_INST_WIDTH>
569  void CPU_ISS<ADDRESS_SIZE, DATA_SIZE, REGISTER_WIDTH, SEGMENT_REGISTER_WIDTH, EFLAG_WIDTH, EIP_WIDTH, MAX_INST_WIDTH>::table1()
570  {
571      opcode_1B[0][0] = &CPU_ISS<ADDRESS_SIZE, DATA_SIZE, REGISTER_WIDTH, SEGMENT_REGISTER_WIDTH, EFLAG_WIDTH, EIP_WIDTH, MAX_INST_WIDTH>::T1_00_ADD;
572      opcode_1B[0][1] = &CPU_ISS<ADDRESS_SIZE, DATA_SIZE, REGISTER_WIDTH, SEGMENT_REGISTER_WIDTH, EFLAG_WIDTH, EIP_WIDTH, MAX_INST_WIDTH>::T1_01_ADD;
573      opcode_1B[0][2] = &CPU_ISS<ADDRESS_SIZE, DATA_SIZE, REGISTER_WIDTH, SEGMENT_REGISTER_WIDTH, EFLAG_WIDTH, EIP_WIDTH, MAX_INST_WIDTH>::T1_02_ADD;
574      opcode_1B[0][3] = &CPU_ISS<ADDRESS_SIZE, DATA_SIZE, REGISTER_WIDTH, SEGMENT_REGISTER_WIDTH, EFLAG_WIDTH, EIP_WIDTH, MAX_INST_WIDTH>::T1_03_ADD;
575      opcode_1B[0][4] = &CPU_ISS<ADDRESS_SIZE, DATA_SIZE, REGISTER_WIDTH, SEGMENT_REGISTER_WIDTH, EFLAG_WIDTH, EIP_WIDTH, MAX_INST_WIDTH>::T1_04_ADD;
576      opcode_1B[0][5] = &CPU_ISS<ADDRESS_SIZE, DATA_SIZE, REGISTER_WIDTH, SEGMENT_REGISTER_WIDTH, EFLAG_WIDTH, EIP_WIDTH, MAX_INST_WIDTH>::T1_05_ADD;
577
578      opcode_1B[2][9] = &CPU_ISS<ADDRESS_SIZE, DATA_SIZE, REGISTER_WIDTH, SEGMENT_REGISTER_WIDTH, EFLAG_WIDTH, EIP_WIDTH, MAX_INST_WIDTH>::T1_29_SUB;
579      opcode_1B[2][11] = &CPU_ISS<ADDRESS_SIZE, DATA_SIZE, REGISTER_WIDTH, SEGMENT_REGISTER_WIDTH, EFLAG_WIDTH, EIP_WIDTH, MAX_INST_WIDTH>::T1_2B_SUB;
580      opcode_1B[2][13] = &CPU_ISS<ADDRESS_SIZE, DATA_SIZE, REGISTER_WIDTH, SEGMENT_REGISTER_WIDTH, EFLAG_WIDTH, EIP_WIDTH, MAX_INST_WIDTH>::T1_2D_SUB;
```

Figure 4-1-4 function table1

We will write the corresponding functions for the remaining instructions in the same way and add them to the opcode_1B table. The implemented instructions are listed below:

ADD – Add

AND – Logical AND

CMP – Compare Two Operands

DEC – Decrement By 1

DIV – Unsigned Divide

IDIV – Signed Divide

INC – Increment By 1

MOV – Move Data

MUL – Unsigned Multiply

OR – Logical Inclusive OR

SUB – Integer Subtraction

4-2- Steps for Executing Instructions in Code

To execute an instruction, several steps are followed. First, instructions must be read from memory in a way that 32 bytes are retrieved. Since each memory location is considered to be 32 bits, reading 32 bytes requires accessing 8 memory locations (equivalent to 256 bits). After reading the instructions from memory, the retrieved data is divided into two 16-byte segments, and each of these 16 bytes is examined separately. Then, it must be determined how many instructions are contained within each 16-byte segment, and these instructions need to be separated. After this stage, the instructions are executed in the order they were read.

The constructor for the main processor module in the code (*CPU_ISS*) is as follows.

```
131     //-----------------------------------CONSTRUCTOR---------
132     SC_CTOR(CPU_ISS)
133     {
134         IP = 1799;
135         initialize();
136         EFLAGS_ptr = &EFLAGS;
137         table1();
138         table1_length();
139         Address_init();
140         SC_THREAD(decoding);
141         sensitive << clk.pos();
142         SC_THREAD(read_mem);
143         sensitive << clk.pos();
144         SC_THREAD(write_mem);
145         sensitive << clk.pos();
146         SC_THREAD(read_Instruction);
147         sensitive << clk.pos();
148         SC_THREAD(pre_decode);
149         sensitive << clk.pos();
150         //  SC_METHOD(set_default);
151         //  sensitive << clk.pos();
152     }
```

Figure 4-2-1 constructor

As observed, the functions *initialize()*, *table1()*, *table1_length()*, and *Address_init()* are executed first for initialization. Next, the decoding function (Figure 4-2-2) is called. When this function is executed, two scenarios can occur at the beginning: if the instruction buffer

is empty, the *read_Inst* event is triggered (notified), and the instruction buffer is filled by executing the *read_Instruction()* function. After that, we proceed to the *pre_decode* stage. If the instruction buffer is not empty from the start, we directly execute the *pre_decode()* function.

```
806  void CPU_ISS<ADDRESS_SIZE, DATA_SIZE, REGISTER_WIDTH, SEGMENT_REGISTER_WIDTH, EFLAG_WIDTH, EIP_WIDTH, MAX_INST_WIDTH>::decoding()
807  {
808      int inst = 0;
809      while (true)
810      {
811          if (PC == 1811)
812          {
813              if (inst == 4)
814                  break;
815              else
816                  inst++;
817          }
818          wait(clk.posedge_event());
819          if (Instruction_Buffer.size() == 0)
820          {
821              read_Inst.notify();
822              wait(fetch_done);
823              pre_dec.notify();
824              wait(pre_dec_done);
825          }
826
827          Instruction = Instruction_Buffer[0];
828          int inst_len;
829          for (int i = MAX_INST_WIDTH; i >= 0; i--)
830          {
831              if (Instruction[i] == 'X')
832              {
833                  inst_len = (MAX_INST_WIDTH - i) / 8;
834                  break;
835              }
836              inst_len = (MAX_INST_WIDTH - i) / 8;
837          }
838          cout << "--------------------------------------------------------------------------------" << endl;
839          cout << "Instruction is :" << Instruction << endl;
840          vector<sc_lv<8>> Instruction_Bytes;
841          for (int i = 0; i < inst_len; i++)
842          {
843              Instruction_Bytes.push_back(Instruction.range(119 - 8 * i, 112 - 8 * i));
844          }
845
846          int opcode_location;
847          // will be completed later
848          opcode_location = 0;
849          int row = Instruction_Bytes[opcode_location].range(7, 4).to_uint();
850          int column = Instruction_Bytes[opcode_location].range(3, 0).to_uint();
851          (this->*opcode_1B[row][column])(Instruction_Bytes);
852          Instruction_Buffer = erase_front(Instruction_Buffer);
853          print_register_values();
854          wait(clk.posedge_event());
855      }
856  }
```

Figure 4-2-2 function decoding

Within the *pre_decode()* function (Figure 4-2-3), the 16 bytes retrieved are examined, and the length of each instruction is determined using functions that end with "_Len" (as shown in Figure 4-2-4). All instructions present in the 16-byte buffer are separated (different

instructions are stored in a vector called Instruction_Buffer). The storage method is such that the mentioned vector contains elements of type *sc_lv<120>*, with each instruction placed at the beginning of each *sc_lv*, followed by an 'X' until it reaches 120 characters. After that, based on the extracted opcode, the corresponding instruction is executed. In the next section, we will see an example of instruction execution and the process of calling functions.

```
726  void CPU_ISS<ADDRESS_SIZE, DATA_SIZE, REGISTER_WIDTH, SEGMENT_REGISTER_WIDTH, EFLAG_WIDTH, EIP_WIDTH, MAX_INST_WIDTH>::pre_decode()
727  {
728      while (true)
729      {
730          wait(pre_dec);
731          string Inst_buffer;
732          vector<sc_lv<8>> opcode_bytes;
733          if (second_pre_dec)
734          {
735              for (int i = 4; i < 8; i++)
736              {
737                  Inst_buffer += Instruction_parts[i].to_string();
738              }
739          }
740          else
741          {
742              for (int i = 0; i < 4; i++)
743              {
744                  Inst_buffer += Instruction_parts[i].to_string();
745              }
746          }
747          while (Inst_buffer.length() != 0) // main pre-decode
748          {
749              opcode_bytes.clear();
750              int inst_length = 0;
751              int opcode_location = 0; // location as byte
752              // byte count is from left (MSB)
753              sc_lv<8> first_byte = Inst_buffer.substr(0, 8).c_str();
754  >           if ((first_byte == 0x66) || (first_byte == 0x67) || (64 < first_byte.to_uint() < 79))…
760              // detecting opcode
761              sc_lv<8> opcode_first_byte = Inst_buffer.substr(opcode_location * 8, 8).c_str();
762              inst_length++;
763              opcode_bytes.push_back(opcode_first_byte);
764              if (opcode_first_byte == 0x0F)
765              {
766                  inst_length++;
767                  sc_lv<8> opcode_second_byte = Inst_buffer.substr(8 * (opcode_location + 1), 8).c_str();
768                  opcode_bytes.push_back(opcode_second_byte);
769                  if (opcode_second_byte == 0x38 || opcode_second_byte == 0x3A)
770                  {
771                      sc_lv<8> opcode_third_byte = Inst_buffer.substr(8 * (opcode_location + 2), 8).c_str();
772                      opcode_bytes.push_back(opcode_third_byte);
773                      inst_length++;
774                  }
775              }
776              switch (opcode_bytes.size())
777              {
778              case 1:
779              {
780                  int row = opcode_bytes[0].range(7, 4).to_uint();
781                  int column = opcode_bytes[0].range(3, 0).to_uint();
782                  inst_length += (this->*opcode_1B_length[row][column])(opcode_bytes, Inst_buffer, inst_length);
783                  break;
784              }
785              case 2:
786  >           {…
792              case 3:
793  >           {…
799              default:
800                  break;
801              }
802              string effective_instruction = Inst_buffer.substr(0, inst_length * 8);
803              int len = 120 - effective_instruction.length();
804  >           for (int i = 0; i < len; i++)…
808              sc_lv<120> temp = effective_instruction.c_str();
809              cout << "effective inst: " << temp << endl;
810              Instruction_Buffer.push_back(temp);
811              Inst_buffer.erase(0, inst_length * 8);
812          }
813          pre_dec_done.notify();
814          wait(clk.posedge_event());
```

Figure 4-2-3 pre_decoding

```
2101    int CPU_ISS<ADDRESS_SIZE, DATA_SIZE, REGISTER_WIDTH, SEGMENT_REGISTER_WIDTH, EFLAG_WIDTH, E
2102        T1_05_ADD_Len(vector<sc_lv<8>> opcode_bytes, string Inst_Buffer, int location)
2103    {
2104        // Immediate to AL,AX or EAX. Since we are doing 32 bit operation, we use EAX.
2105        // Since this instruction doesn't need ModR/M byte, the next Bytes are Immediate data.
2106        sc_lv<8> opcode = opcode_bytes[0];
2107        if (opcode[0] == SC_LOGIC_0)
2108            return 1; // immediate 8 bit
2109        else if (opcode[1] == SC_LOGIC_1)
2110            return 4; // immediate 32 bit
2111    }
```

Figure 4-2-4 instruction length finder

Assume that the "*ADD: memory to register*" instruction is detected in the decode function, represented as "0000 0000: mod reg r/m." Of course, if necessary, SIB and displacement bits can be added to this instruction (Figure 4-2-5).

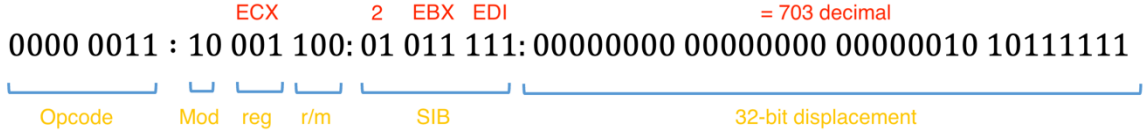


Figure 4-2-5 ADD: memory to register

Based on the second byte of the instruction, which is the ModRM, the operands of the instruction must be determined. By placing the values of the Mod, Reg, and R/M sections in the table (Figure 3-3-2), it is determined that the first operand is the EAC register. To find the second operand, attention must be paid to the SIB byte and also to the table in Figure 3-3-3. The entry pointed to by the SIB byte is identified in the figure, and using formula 4-1, the memory address from which the content should be read is determined.

$$memory\ address = scale * index + base + displacement$$

Formula 4-1 physical address generation formula

For example, suppose the following values are stored in the processor registers and memory:

ECX = 420

EDI = 30

EBX = 20

Mem [773] = 5043

The process for calculating the memory location from which we need to read the value is as follows:

scale = 2

index = EBX = 20

base = EDI = 30

displacement = 703

Mem [2*20 + 30 + 703] = Mem [773] = 5043

Now, the addition operation between the two operands is performed:

ECX + Mem [773] = 420 + 5043 = 5463

Finally, the result of the addition is stored in the ECX register, and the addition operation is completed.

Now let's examine the same instruction and its execution process in the code. We previously covered up to the decode section. We know that the Opcode "0000 0011" in hexadecimal becomes 03h. Therefore, the function "T1_03_ADD" should be executed here. (Figure 4-2-6).

```
1084  void CPU_ISS<ADDRESS_SIZE, DATA_SIZE, REGISTER_WIDTH, SEGMENT_REGISTER_WIDTH, EFLAG_WIDTH, EIP_WIDTH, MAX_INST_WIDTH>::
1085      T1_03_ADD(vector<sc_lv<8>> Instruction_Bytes)
1086  {
1087      int mod_location = 1; // needs completion
1088      sc_lv<8> ModRM_Byte = Instruction_Bytes[mod_location];
1089      int table_index = ModRM_Byte.to_uint();
1090      if (table_index > 192) // Mod = 11 : simple register to register Instruction
1091      {
1092          sc_lv<REGISTER_WIDTH> *operand1 = convert_to_reg(Operand_32b_MODRM[table_index][0]);
1093          sc_lv<REGISTER_WIDTH> *operand2 = convert_to_reg(Operand_32b_MODRM[table_index][1]);
1094          DO_Addition(operand1, operand2, operand1, EFLAGS);
1095
1096          Instruction_done.notify();
1097      }
1098      else if ((ModRM_Byte.range(7, 6) == "00" || ModRM_Byte.range(7, 6) == "01" || ModRM_Byte.range(7, 6) == "10") && ModRM_Byte.range(2, 0) == "100") // SIB needed
1099      {
1100          effective_address = convert_to_SIB_address(Instruction_Bytes, mod_location);
1101          cout << "address is : " << effective_address << endl;
1102          AddressBus = effective_address;
1103          wait(clk.posedge_event());
1104          read_data.notify();
1105          wait(read_done);
1106
1107          sc_lv<REGISTER_WIDTH> *operand = convert_to_reg(Operand_32b_MODRM[table_index][1]);
1108          DO_Addition(operand, &memory_read_data, operand, EFLAGS);
1109
1110          Instruction_done.notify();
1111      }
1112      else // simple memory to register without SIB byte
1113      {
1114          find_disp(Instruction_Bytes, mod_location);
1115          effective_address;
1116          int addr;
1117          addr = convert_to_address(Operand_32b_MODRM[table_index][0]);
1118          cout << "address is : " << addr << endl;
1119          effective_address = addr;
1120          sc_lv<REGISTER_WIDTH> *operand = convert_to_reg(Operand_32b_MODRM[table_index][1]);
1121          AddressBus = effective_address;
1122          wait(clk.posedge_event());
1123          read_data.notify();
1124          wait(read_done);
1125          DO_Addition(operand, &memory_read_data, operand, EFLAGS);
1126          wait(clk.posedge_event());
1127          Instruction_done.notify();
1128      }
1129  }
```

Figure 4-2-6 function T1_03_ADD

Here, since our operation is "memory to register," the second part of the conditions in the function *T1_03_ADD*, specifically line 1098, is executed. Upon closer inspection of this section, we see that the address of the memory location to be read is obtained using the *convert_to_SIB_address* function. Then, the memory read operation is performed. After

that, the second operand is identified using the *convert_to_reg* function. In the next step, the *Do_Addition* function is called. (Figure 4-2-7 and *BaseFunctions.cpp* file).

```
66  void DO_Addition(sc_lv<32> *op1, sc_lv<32> *op2, sc_lv<32> *result, sc_lv<32> &EFLAGS)
67  {
68      cout << "Addition Called" << endl;
69      cout << "operand 1 is : " << op1->to_int() << endl;
70      cout << "operand 2 is : " << op2->to_int() << endl;
71      *result = op1->to_int() + op2->to_int();
72      cout << "result is : " << result->to_int() << endl;
73      set_Eflags(*op1, *op2, *result, EFLAGS, 1, "11111100000");
74  }
```

Figure 4-2-7 function Do_Addition

Further explanation of the code in Figure 4-2-7:

This function, specific to row 0 and column 3 of Figure 3-2, performs the addition operation. In the initial lines of code, the position of the ModR/M bits among other bits is determined, and a variable named *table_index* is used to store the integer value of the ModR/M bits. Then, by analyzing the different parts of the ModR/M bits, we identify which section of the table in Figure 3-5 the operation falls under.

If the *table_index* value is greater than 192, it indicates that the two operands of the instruction are stored in the processor's registers, meaning we just need to read the register values and perform the operation. Otherwise, if memory access is required to reach the necessary operands, and the condition in line 1098 is true, the SIB bits will be needed for memory access. If the condition in line 1098 is not met, memory access is done without SIB bits.

Within the condition of line 1098, the address referenced by the SIB bits is first determined. The address is then placed on the bus, and memory is read. Once the memory data is read, the other operand is fetched from the processor's registers, and the addition operation is performed.

If the condition in line 1112 is met, the *find_disp* function is called to find the location of the displacement bits. These bits are used to determine the memory address that should be

read. After that, the remaining steps are executed in the same way as in the previous condition.

As can be seen, in this function, the addition between the two operands is performed, and after that, the *set_Eflags* function is called to change the necessary flags based on the instruction and its result. (Figure 4-2-8 and the *BaseFunctions.cpp* file)

```
15  void set_Eflags(sc_lv<32> operand1, sc_lv<32> operand2, sc_lv<32> result, sc_lv<32> &EFLAGS, int mode, string flags_status) //--> flags: false: 0, true: 1, clear: 2
16  {                                                                                                                          //--> mode: 1: ADD, 2: SUB, 3: AND, 4: OR, 5: MUL
17      int OF = flags_status[0], SF = flags_status[1], ZF = flags_status[2], AF = flags_status[3], PF = flags_status[4],
18          CF = flags_status[5], TF = flags_status[6], IF = flags_status[7], DF = flags_status[8], NT = flags_status[9], RF = flags_status[10];
19      // ZF (Zero Flag)
20      if (ZF)
21      {
22          if (result.to_uint() == 0)
23              EFLAGS[6] = 1;
24          else
25              EFLAGS[6] = 0;
26      }
27      // SF (Sign Flag)
28      if (SF)
29      {
30          if (result[31] == 1)
31              EFLAGS[7] = 1;
32          else
33              EFLAGS[7] = 0;
34      }
35      if (CF)
36      {
37          // CF (Carry Flag)
38      }
39      if (PF)
40      {
41          EFLAGS[2] = check_parity(result);
42      }
43      // OF (Overflow Flag)
44      if (OF)
45      {
46          if (mode == 1) // Add
47          {
48              if ((operand1[31] == 0) && (operand2[31] == 0) && (result[31] == 1))
49                  EFLAGS[11] = 1;
50              else if ((operand1[31] == 1) && (operand2[31] == 1) && (result[31] == 0))
51                  EFLAGS[11] = 1;
52              else
53                  EFLAGS[11] = 0;
54          }
55          else if (mode == 2) // Subtract
56          {
57              if (((operand1[31] ^ operand2[31]) == 1) && ((result[31] ^ operand1[31]) == 1))
58                  EFLAGS[11] = 1;
59              else
60                  EFLAGS[11] = 0;
61          }
62      }
63      cout << "EFLAGS UPDATE : " << EFLAGS << endl;
64  }
```

Figure 4-2-8 function set_Eflags

Note: If we look at line 1108 in Figure 4-2-6, we see that the result is written to the location of the "*operand*," which is the pointer to the register. The reason for this is that the sum is supposed to be stored in the register.
Once this instruction is completed, the *Instruction_done* event is triggered to allow the processor to move on to execute the next instructions.

4-5- Testbench

To test this system, we first read 8 consecutive memory locations, starting from address 1799. Then we pre-decode 4 of them and execute the instructions in order.

The first instruction: Add the content of memory to the register and store the result in the destination register ESP. The memory address is calculated based on the instruction as follows:

[EBX] + disp32 = 20 + 60 = 80

```
Instruction is :00000011101000110000000000000000000000000111100XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
disp32 is :60
address is : 80
Addition Called
operand 1 is : 80
operand 2 is : 390880194
result is : 390880274
EFLAGS UPDATE : XXXXXXXXXXXXXXXXXXXX0XXX00XXX0XX
EAX Register value: 1
EBX Register value: 20
ECX Register value: 30
EDX Register value: 40
EBP Register value: 50
ESI Register value: 60
EDI Register value: 70
ESP Register value: 390880274
```

Figure 4-5-1 Test result

The second instruction: Add the content of the register to memory and store the result in memory. The source register is EBX. The memory address is obtained through the SIB byte and displacement (disp).

```
Instruction is :00000001010111001100111011000000XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
disp8 is :206
scale : 8
index is : 30
disp8 is :192
Addition Called
operand 1 is : 20
operand 2 is : -1401842134
result is : -1401842114
EFLAGS UPDATE : XXXXXXXXXXXXXXXXXXXX0XXX10XXX0XX
data[-1401842114] written in address[492] at 550 ns
EAX Register value: 1
EBX Register value: 20
ECX Register value: 30
EDX Register value: 40
EBP Register value: 50
ESI Register value: 60
EDI Register value: 70
ESP Register value: 390880274
```

Figure 4-5-2 Test result

The third instruction: Add the content of the immediate value to memory and store the result in memory. The memory address is obtained from the instruction as follows:

[ESI] + disp8

```
Instruction is :10000011010001101000101101110000XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
disp8 is :139
imm is: 112
AND Called
operand 1 is memory data : 01110001010000000001111001101001
operand 2 is Immediate : 00000000000000000000000001110000
result is : 00000000000000000000000001100000
EFLAGS UPDATE : XXXXXXXXXXXXXXXXXXXX0XXX00XXX1XX
data[-1401842114] written in address[199] at 670 ns
data[96] written in address[199] at 690 ns
EAX Register value: 1
EBX Register value: 20
ECX Register value: 30
EDX Register value: 40
EBP Register value: 50
ESI Register value: 60
EDI Register value: 70
ESP Register value: 390880274
```

Figure 4-5-3 Test result

The fourth instruction: The contents of two registers are ANDed together, and the result is stored in the second register. The first register is ESI, and the second register is ECX.

```
Instruction is :0010000111110001XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
AND Called
operand 1 is : 00000000000000000000000000011110
operand 2 is : 00000000000000000000000000111100
result is : 00000000000000000000000000011100
EFLAGS UPDATE : XXXXXXXXXXXXXXXXXXX0XXX00XXX0XX
EAX Register value: 1
EBX Register value: 20
ECX Register value: 30
EDX Register value: 40
EBP Register value: 50
ESI Register value: 28
EDI Register value: 70
ESP Register value: 390880274
```

Figure 4-5-4 Test result

Now that the internal instruction buffer is emptied, since there are still 4 unprocessed locations in the processor's memory, we will pre-decode them.

```
effective inst: 000010110011110100000000000000000000011100000000XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
effective inst: 001010111000010001001101000000000000000000000000100000001111XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
effective inst: 111111110000010000001010XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Figure 4-5-5 Test result

Fifth instruction: The content of memory is going through OR gate with a register and stored in memory. The source register is EDI, and the memory address is calculated from the instruction as follows: [disp32] = 1792

```
Instruction is :000010110011110100000000000000000000011100000000XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
disp32 is :1792
address is : 1792
OR Called
operand 1 is : 00000000000000000000000001000110
operand 2 is : 11110110000010110001110001111101
result is : 11110110000010110001110001111111
EFLAGS UPDATE : XXXXXXXXXXXXXXXXXXX0XXX10XXX0XX
EAX Register value: 1
EBX Register value: 20
ECX Register value: 30
EDX Register value: 40
EBP Register value: 50
ESI Register value: 28
EDI Register value: -167043969
ESP Register value: 390880274
```

Figure 4-5-6 Test result

Sixth instruction: The content of memory is subtracted from the content of a register, and the result is stored in memory. The source register is EAX, and the memory address is calculated from the instruction as follows:

[ECX]*2 + [EBP] + disp32 = 1149

```
Instruction is :00101011100001000100110100000000000000000000000010000001111XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
scale : 2
index is : 30
disp32 is :1039
address is : 00000000000000000000010001111101
Subtraction Called
operand 1 is : 1
operand 2 is : -290361455
EFLAGS UPDATE : XXXXXXXXXXXXXXXXXXXX0XXX00XXX0XX
result is : 290361456
EAX Register value: 290361456
EBX Register value: 20
ECX Register value: 30
EDX Register value: 40
EBP Register value: 50
ESI Register value: 28
EDI Register value: -167043969
ESP Register value: 390880274
```

Figure 4-5-7 Test result

Seventh instruction: One is added to the content of memory. The memory address is calculated from the instruction as follows:

[ECX] + [EDX] = 70

```
Instruction is :1111111100000100000001010XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
scale : 1
index is : 30
address is : 00000000000000000000000001000110
Increment Called
operand is memory data : -1784701158
result is : -1784701157
EFLAGS UPDATE : XXXXXXXXXXXXXXXXXXXX0XXX10XXX1XX
data[-1784701157] written in address[70] at 1390 ns
EAX Register value: 290361456
EBX Register value: 20
ECX Register value: 30
EDX Register value: 40
EBP Register value: 50
ESI Register value: 28
EDI Register value: -167043969
ESP Register value: 390880274
```

Figure 4-5-8 Test result

The instructions have been cleared from the buffer, so we read eight consecutive memory locations and pre-decode four of them.

```
effective inst: 1111111111001011XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
effective inst: 11110111011001000110011000100011XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
effective inst: 1111011111111001XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
effective inst: 11000111110001000000000000000000001111111100000000XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
effective inst: 1000100100010010XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
-------------------------------------------------------------------------------
```

Figure 4-5-9 Test result

Eighth instruction: One is subtracted from the content of the register. The register in question is EBX.

```
Instruction is :1111111111001011XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Decrement Called
operand is : 20
result is : 19
EFLAGS UPDATE : XXXXXXXXXXXXXXXXXXXX0XXX00XXX0XX
EAX Register value: 290361456
EBX Register value: 19
ECX Register value: 30
EDX Register value: 40
EBP Register value: 50
ESI Register value: 28
EDI Register value: -167043969
ESP Register value: 390880274
```

Figure 4-5-10 Test result

Ninth instruction: The content of the register is multiplied by the memory and stored in memory. The register in question is EAX, and the memory address is calculated from the instruction as follows:

[ESI] + disp8 = 35 + 28 = 63

```
Instruction is :11110111011001000110011000100011XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
scale : 2
index is : 0
disp8 is :35
Multiplication Called
operand 1 is : 290361456
operand 2 is memory data : 1843654154
result is : 2057946208
EFLAGS UPDATE : XXXXXXXXXXXXXXXXXXXX0XXX00XXX1XX
data[2057946208] written in address[63] at 1890 ns
EAX Register value: 290361456
EBX Register value: 19
ECX Register value: 30
EDX Register value: 40
EBP Register value: 50
ESI Register value: 28
EDI Register value: -167043969
ESP Register value: 390880274
--------------------------------------------------------------------------------
```

Figure 4-5-11 Test result

Tenth instruction: The contents of the two registers are divided. The first register is EAX, and the second register is ECX. This division is performed as signed.

```
Instruction is :1111011111111001XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Signed Division Called
operand 1 is : 290361456
operand 2 is : 30
result is : 9678715
EAX Register value: 290361456
EBX Register value: 19
ECX Register value: 9678715
EDX Register value: 40
EBP Register value: 50
ESI Register value: 28
EDI Register value: -167043969
ESP Register value: 390880274
```

Figure 4-5-12 Test result

Eleventh instruction: This is an immediate transfer instruction to the register. The target register is ESP.

```
Instruction is :110001111100010000000000000000001111111100000000XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
imm is: 65280
Move Called
Source value : 65280
Destination previous value: 390880274
Destination current value: 65280
EAX Register value: 290361456
EBX Register value: 19
ECX Register value: 9678715
EDX Register value: 40
EBP Register value: 50
ESI Register value: 28
EDI Register value: -167043969
ESP Register value: 65280
```

Figure 4-5-13 Test result

Twelfth instruction: The content of the register is written to memory. The register in question is EDX, and the memory address is also the content of EDX.

```
Instruction is :1000100100010010XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
data[40] written in address[40] at 2050 ns
EAX Register value: 290361456
EBX Register value: 19
ECX Register value: 9678715
EDX Register value: 40
EBP Register value: 50
ESI Register value: 28
EDI Register value: -167043969
ESP Register value: 65280
```

Figure 4-5-14 Test result

The instruction buffer is empty again. We will pre-decode the next 4 cells that have been previously fetched.

Thirteenth instruction: A comparison between the register and memory that changes the EFLAGS. The memory address is obtained as follows:

[ESI] + disp 32 = 28 + 1280 = 1308

The content of memory cell 1308: 10001001110011001010100110000011, which is signed and equal to -1983075965.

```
Instruction is :00111011100111100000000000000000000001010000000XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
disp32 is :1280
Compare Called
operand 1 is : 19
operand 2 is : -1983075965
EFLAGS UPDATE : XXXXXXXXXXXXXXXXXXX0XXX00XXX0XX
EAX Register value: 290361456
EBX Register value: 19
ECX Register value: 9678715
EDX Register value: 40
EBP Register value: 50
ESI Register value: 28
EDI Register value: -167043969
ESP Register value: 65280
```

Figure 4-5-15 Test result

Fourteenth instruction: This instruction is a transfer from memory to register. The destination register is EAX. The memory address is obtained as follows:

Scale * index + base + disp32 = 1 * none + [EBX] + 1024 = 1043

The content of this memory cell: 00010011110110100110101101110010, which is signed and equal to 333081458.

```
Instruction is :10001011100001000010001100000000000000000000100000000000XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
scale : 1
index is : 0
disp32 is :1024
Move Called
Source value : 333081458
Destination previous value: 290361456
Destination current value: 333081458
EAX Register value: 333081458
EBX Register value: 19
ECX Register value: 9678715
EDX Register value: 40
EBP Register value: 50
ESI Register value: 28
EDI Register value: -167043969
ESP Register value: 65280
```

Figure 4-5-16 Test result

Fifteenth instruction: This instruction is the subtraction of a register from memory and storing the result in memory. The register in question is EAX, and the memory address is obtained as follows: [ESI] + disp8 = 28 + 200 = 228

```
Instruction is :001010010100011011001000XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
disp8 is :200
Subtraction Called
operand 1 is : 333081458
operand 2 is : 440197849
result is : -107116391
EFLAGS UPDATE : XXXXXXXXXXXXXXXXXXXX0XXX10XXX0XX
Addition Called
operand 1 is : 333081458
operand 2 is : 440197849
result is : 773279307
EFLAGS UPDATE : XXXXXXXXXXXXXXXXXXXX0XXX00XXX1XX
data[773279307] written in address[228] at 2710 ns
EAX Register value: 333081458
EBX Register value: 19
ECX Register value: 9678715
EDX Register value: 40
EBP Register value: 50
ESI Register value: 28
EDI Register value: -167043969
ESP Register value: 65280
```

Figure 4-5-17 Test result

Reference

[1] Intel® 64 and IA-32 Architectures Software Developer’s Manual