컴퓨터구조

이성원교수님

Project #2

학과 : 컴퓨터 정보 공학부

학번 : 2019202103

이름 : 이은비

제출 날짜 : 2023.05.07

<프로젝트 개요 및 내용 요약>

MIPS Inst Datasheet.pdf, Control Signals for MCPU.pdf를 참고하여서 주어진 zip파일 내에 있는ROM\_DISP.txt와 ROM\_MICRO.txt 파일의 Address와 Control signal을 설정하는 것입니다. 즉, ROM\_MICRO.txt 파일에서 설정한 Signal의 Address에 맞춰서 ROM\_DISP.txt 파일에 적는 것인데 이때 작성 되는 ROM\_MICRO.txt는 CPI =1 인 processor’s problem을 해결하기 위해 고안된 Multi-cycle Approach의 방법, 즉 Multi-cycle의 형태에서 instruction을 수행하며 txt파일의 각 bit도 그에 해당하는 signal로 작성하도록 합니다. 추가적으로 첨부된 M\_TEXT\_SEG.txt는 작성한 명령어를 테스트하기 위한 txt파일이므로 이를 이용해서 gtkwave 파형을 확인하여 작성된 명령어를 simulation을 통해 확인합니다..

<문제의 해석 및 해결 방향>

실험 내용에 대한 설명 ex) 자기가 구현한 FSM, 동작 및 특징

Multipath cycle CPU를 사용하여 instruction을 진행한다 하였을 때 아래와 같은 instruction의 FSM과 동작 및 특성을 설명하면 다음과 같습니다. Stage별로 IF->ID->EXEC->MEM->WB 순으로 이어지는 것을 알 수 있으며 FSM을 기반으로 아래서 설명합니다.

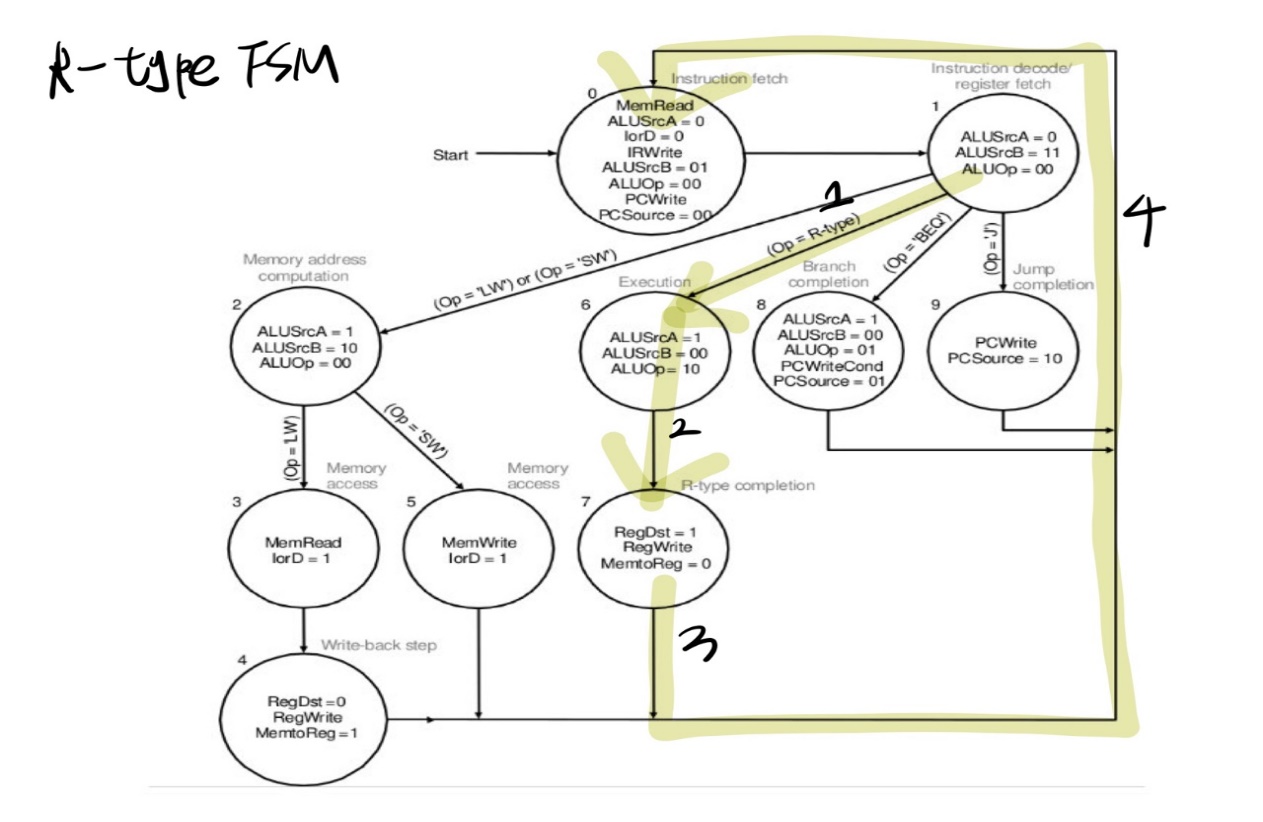
**ADDU, OR, ADDIU, XORI, SLL, SRAV, SH, LH, BLTZ, JAL**

**[각 명령어에 대한 FSM과 동작 및 특성]**

아래 모든 명령어의 공통인 부분을 보면 IF와 ID부분은 공통으로 이뤄지는 것을 알 수 있으며 각 instruction별로 fsm과 같이 설명하면 다음과 같습니다.

**ADDU(R-type instruction),**

[FSM]



일반적인 Rtype instruction으로 볼 수 있습니다. State는 1->6->7->0 순으로 진행되는 것을 알 수 있습니다. 그리고 각 state마다 해당 되는 bit 변화를 변화시키는 작업을 진행합니다.

Execution: ALU operation 진행합니다. 즉, add operation: $d = $s + $t 에서 $s + $t, 즉 add operation을 진행합니다.

R-type completion: register에 연산결과를 저장하는 과정을 거칩니다. $d = $s + $t 에서 $d에 저장하는 과정이라고 할 수 있습니다.

(연산과정을 상세히 풀어보면 다음과 같습니다.)

Fetch instruction: Instruction ← MEM[PC]

Fetch operands: data1 ← Reg(Rs), data2 ← Reg(Rt)

Execute operation: ALU\_result ← func(data1, data2) ($d = $s + $t)

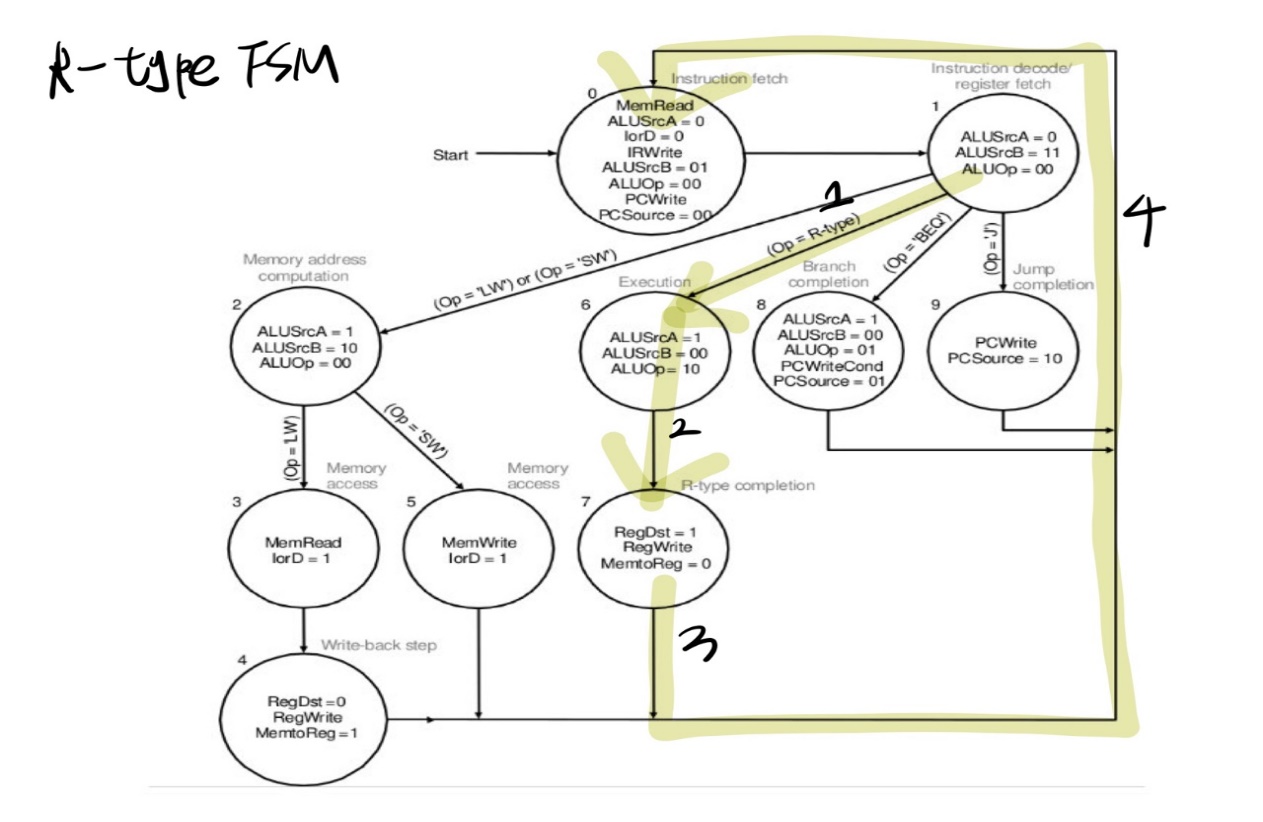
Write ALU result: Reg(Rd) ← ALU\_result

Next PC address: PC ← PC + 4

Operation: $d = $s + $t

**OR(R-type instruction)**

[FSM]



일반적인 Rtype instruction으로 볼 수 있습니다. State는 1->6->7->0 순으로 진행되는 것을 알 수 있습니다. 그리고 각 state마다 해당 되는 bit 변화를 변화시키는 작업을 진행합니다.

Execution: ALU operation 진행합니다. 즉, or operation: $d = $s | $t 에서 $s | $t, 즉 or operation을 진행합니다.

R-type completion: register에 연산결과를 저장하는 과정을 거칩니다. $d = $s | $t 에서 $d에 저장하는 과정이라고 할 수 있습니다.

(연산과정을 상세히 풀어보면 다음과 같습니다.)

Fetch instruction: Instruction ← MEM[PC]

Fetch operands: data1 ← Reg(Rs),

data2 ← Reg(Rt)

Execute operation: ALU\_result ← func(data1, data2) ($d = $s | $t)

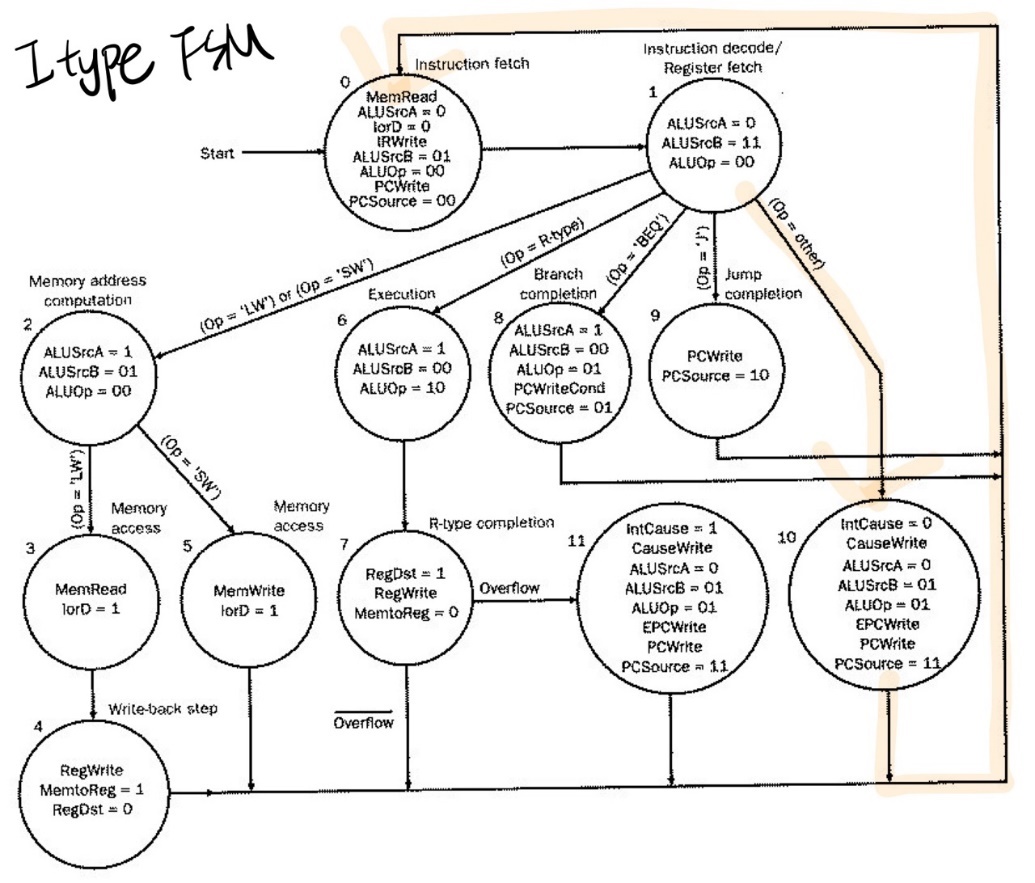
Write ALU result: Reg(Rd) ← ALU\_result

Next PC address: PC ← PC + 4

Operation: $d = $s | $t

**ADDIU(RegIm),**

[FSM]



일반적인 I-type instrction으로 볼 수 있습니다. State는 1->10->0 순으로 진행되는 것을 알 수 있습니다. 그리고 각 state마다 해당되는 bit 변화를 변화시키는 작업을 진행합니다.

(연산과정을 상세히 풀어보면 다음과 같습니다.)

Fetch instruction: Instruction ← MEM[PC] Fetch operands:

data1 ← Reg(Rs), data2 ← Extend(imm16) (shift-extended를 거친)

Execute operation: ALU\_result ← op(data1, data2) ($t = $s + SE(i))

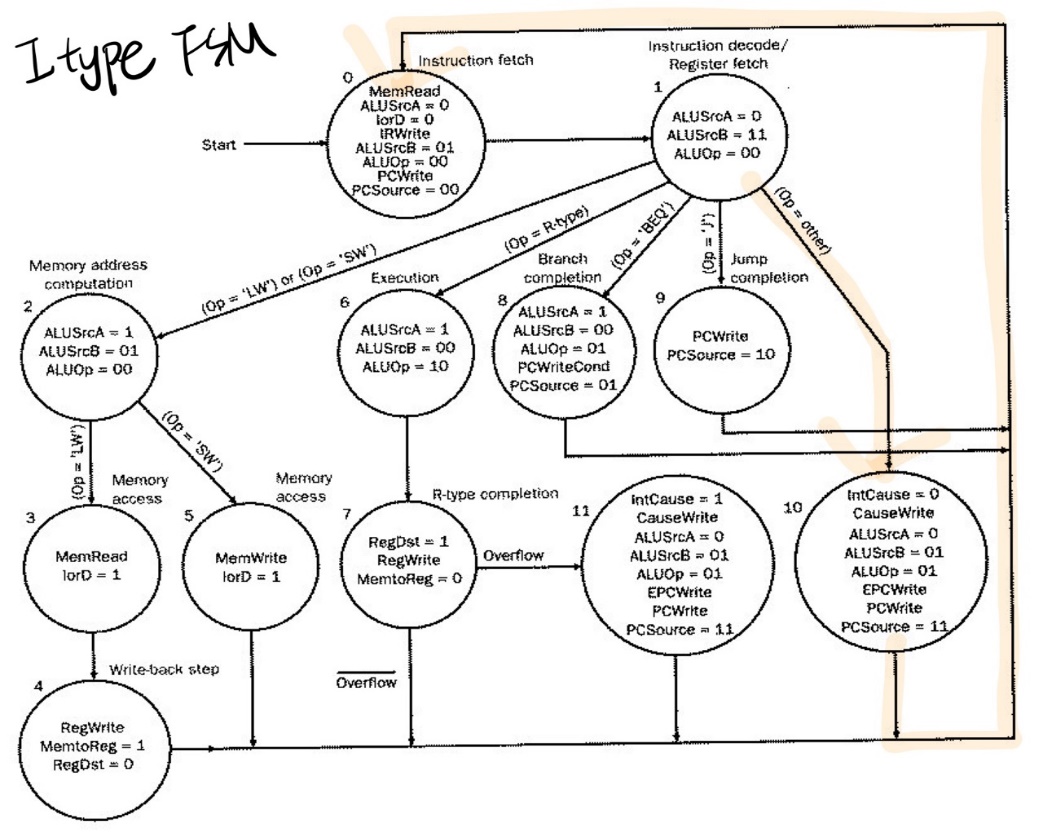
Write ALU result: Reg(Rt) ← ALU\_result

Next PC address: PC ← PC + 4

Operation: $t = $s + SE(i)

**XORI**

[FSM]



일반적인 I-type instrction으로 볼 수 있습니다. State는 1->10->0 순으로 진행되는 것을 알 수 있습니다. 그리고 각 state마다 해당되는 bit 변화를 변화시키는 작업을 진행합니다.

(연산과정을 상세히 풀어보면 다음과 같습니다.)

Fetch instruction: Instruction ← MEM[PC]

Fetch operands: data1 ← Reg(Rs), data2 ← Extend(imm16) (zero-extended를 거친)

Execute operation: ALU\_result ← op(data1, data2) ($t = $s ^ ZE(i))

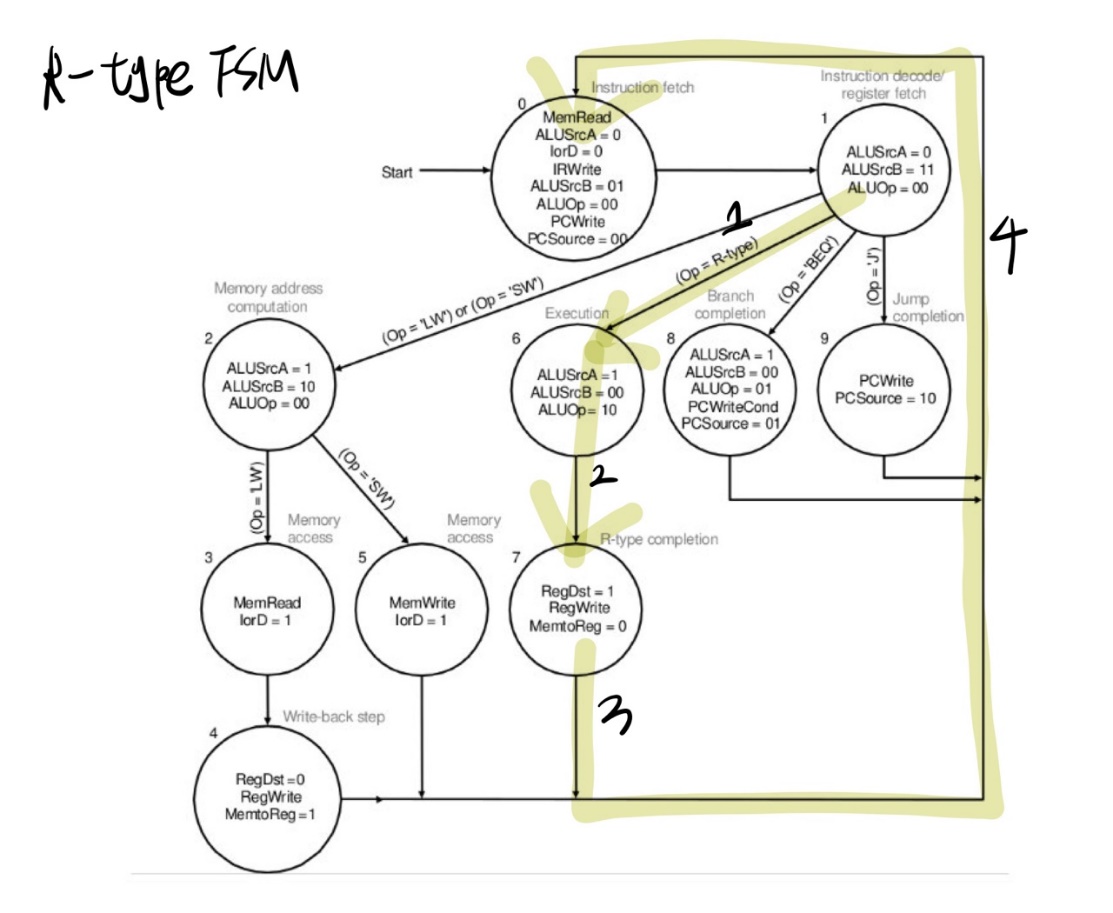
Write ALU result: Reg(Rt) ← ALU\_result

Next PC address: PC ← PC + 4

Operation: $t = $s ^ ZE(i)

**SLL(R-type instruction)**

[FSM]

****

R-type instruction으로 볼 수 있습니다. State는 1->6->7->0 순으로 진행되는 것을 알 수 있습니다. 그리고 각 state마다 해당되는 bit 변화를 변화시키는 작업을 진행합니다.

Execution: ALU operation 진행합니다. 즉, or operation: $d = $t << a 에서 $t << a, 즉 or operation을 진행합니다.

R-type completion: register에 연산결과를 저장하는 과정을 거칩니다. $d = $t << a 에서 $d에 저장하는 과정이라고 할 수 있습니다.

(연산과정을 상세히 풀어보면 다음과 같습니다.)

Fetch instruction: Instruction ← MEM[PC]

Fetch operands: data1 ← Reg(Rs), data2 ← Reg(Rt)

Execute operation: ALU\_result ← func(data1, data2) ($d = $t << a)

Write ALU result: Reg(Rd) ← ALU\_result

Next PC address: PC ← PC + 4

Operation: $d = $t << a

**SRAV(R-type instruction)**

[FSM]

****

R-type instruction으로 볼 수 있습니다. State는 1->6->7->0 순으로 진행되는 것을 알 수 있습니다. 그리고 각 state마다 해당되는 bit 변화를 변화시키는 작업을 진행합니다.

Execution: ALU operation 진행합니다. 즉, or operation $d = $t >>> $s 에서 $t << a, 즉 or operation을 진행합니다.

R-type completion: register에 연산결과를 저장하는 과정을 거칩니다. $d = $t >>> $s 에서 $d에 저장하는 과정이라고 할 수 있습니다.

(연산과정을 상세히 풀어보면 다음과 같습니다.)

Fetch instruction: Instruction ← MEM[PC]

Fetch operands: data1 ← Reg(Rs),

data2 ← Reg(Rt)

Execute operation: ALU\_result ← func(data1, data2) ($d = $t >>> $s)

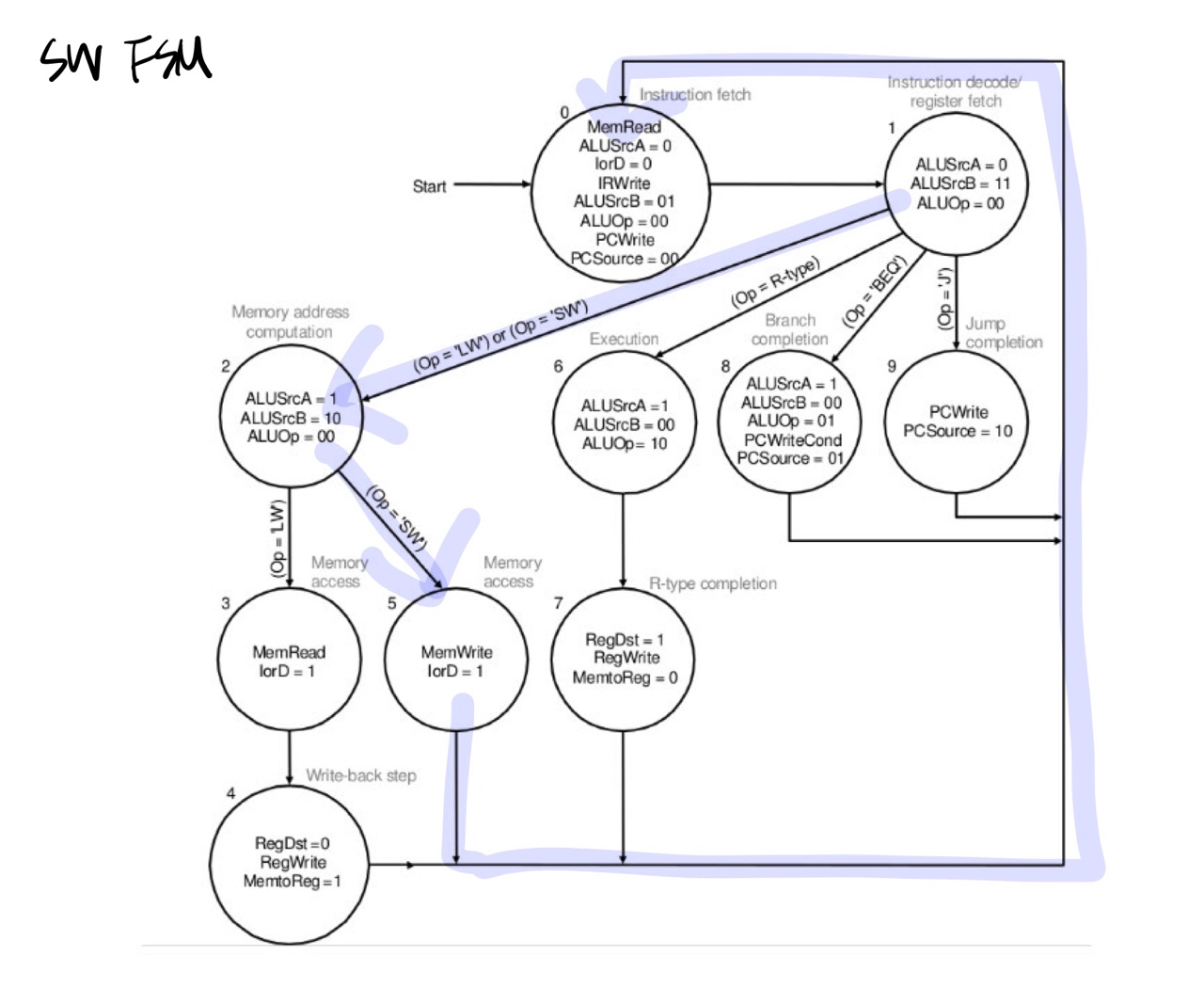
Write ALU result: Reg(Rd) ← ALU\_result

Next PC address: PC ← PC + 4

Operation: $d = $t >>> $s

**SH(store)**

[FSM]

****

store instruction으로 볼 수 있습니다. State는 1->2->5->0 순으로 진행되는 것을 알 수 있습니다. 그리고 각 state마다 해당되는 bit 변화를 변화시키는 작업을 진행합니다.

Memory address computation: address ← base + sign\_extend(imm16)에서 base +

sign\_extend(imm16)에 해당하는 작업을 진행합니다.

Memory access: MEM[address] ← data (MEM [$s + i]:2 = LH ($t))와 같이 위의 address computation에서의 operation결과를 저장합니다.

(연산과정을 상세히 풀어보면 다음과 같습니다.)

Fetch instruction: Instruction ← MEM[PC]

Fetch registers: base ← Reg(Rs), data ← Reg(Rt)

Calculate address: address ← base + sign\_extend(imm16)

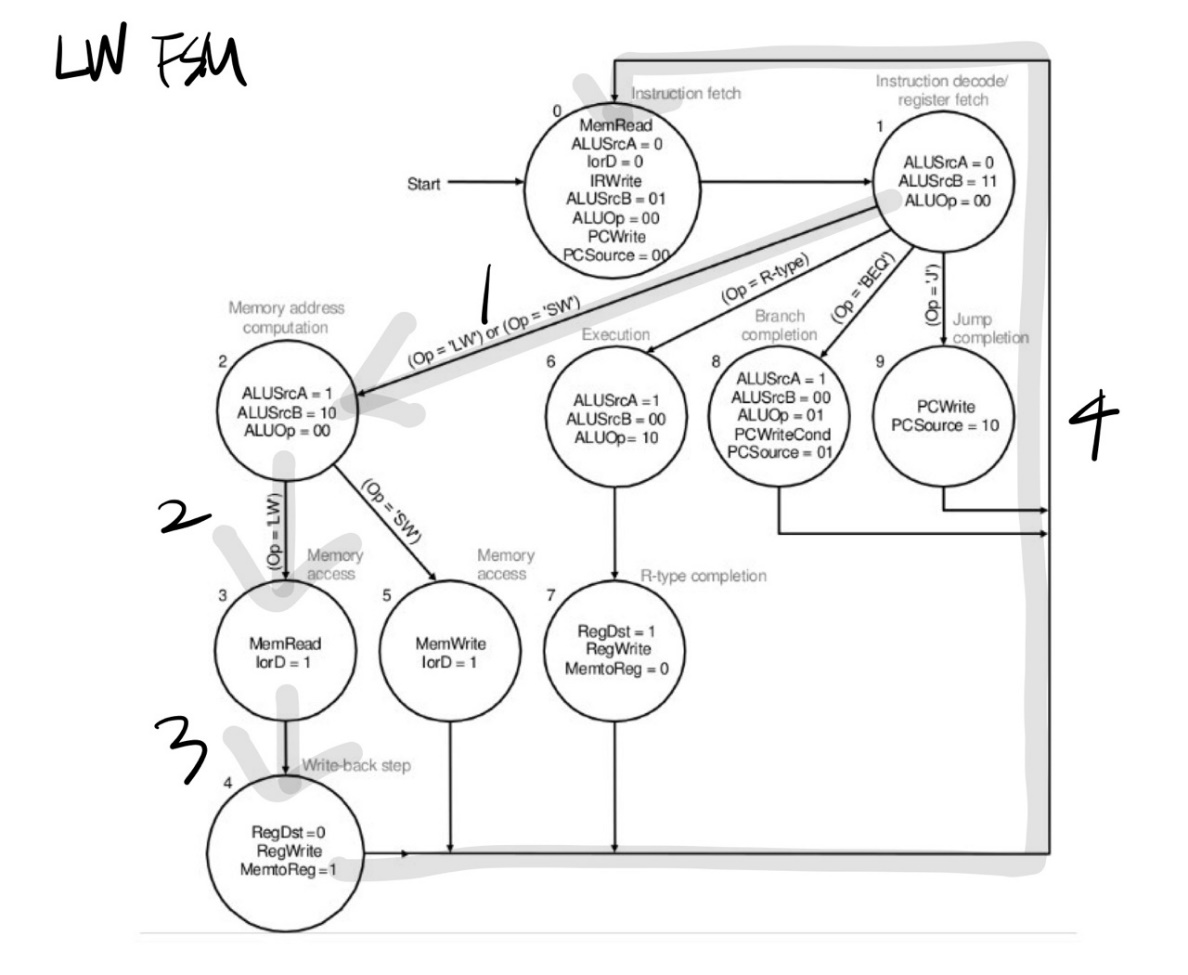
Write memory: MEM[address] ← data (MEM [$s + i]:2 = LH ($t))

Next PC address: PC ← PC + 4

Operation: MEM [$s + i]:2 = LH ($t)

**LH(load)**

[FSM]

****

load instruction으로 볼 수 있습니다. State는 1->2->3->4->0 순으로 진행되는 것을 알 수 있습니다. 그리고 각 state마다 해당되는 bit 변화를 변화시키는 작업을 진행합니다.

Memory address computation: address ← base + sign\_extend(imm16) (SE (MEM [$s + i]:2)

Memory access: data ← MEM[address]

Write back: register로 Reg(Rt) ← data ($t = SE (MEM [$s + i]:2)) 과 같이 writeback하는 작업을 진행합니다.

(연산과정을 상세히 풀어보면 다음과 같습니다.)

Fetch instruction: Instruction ← MEM[PC]

Fetch base register: base ← Reg(Rs)

Calculate address: address ← base + sign\_extend(imm16) (SE (MEM [$s + i]:2)

Read memory: data ← MEM[address]

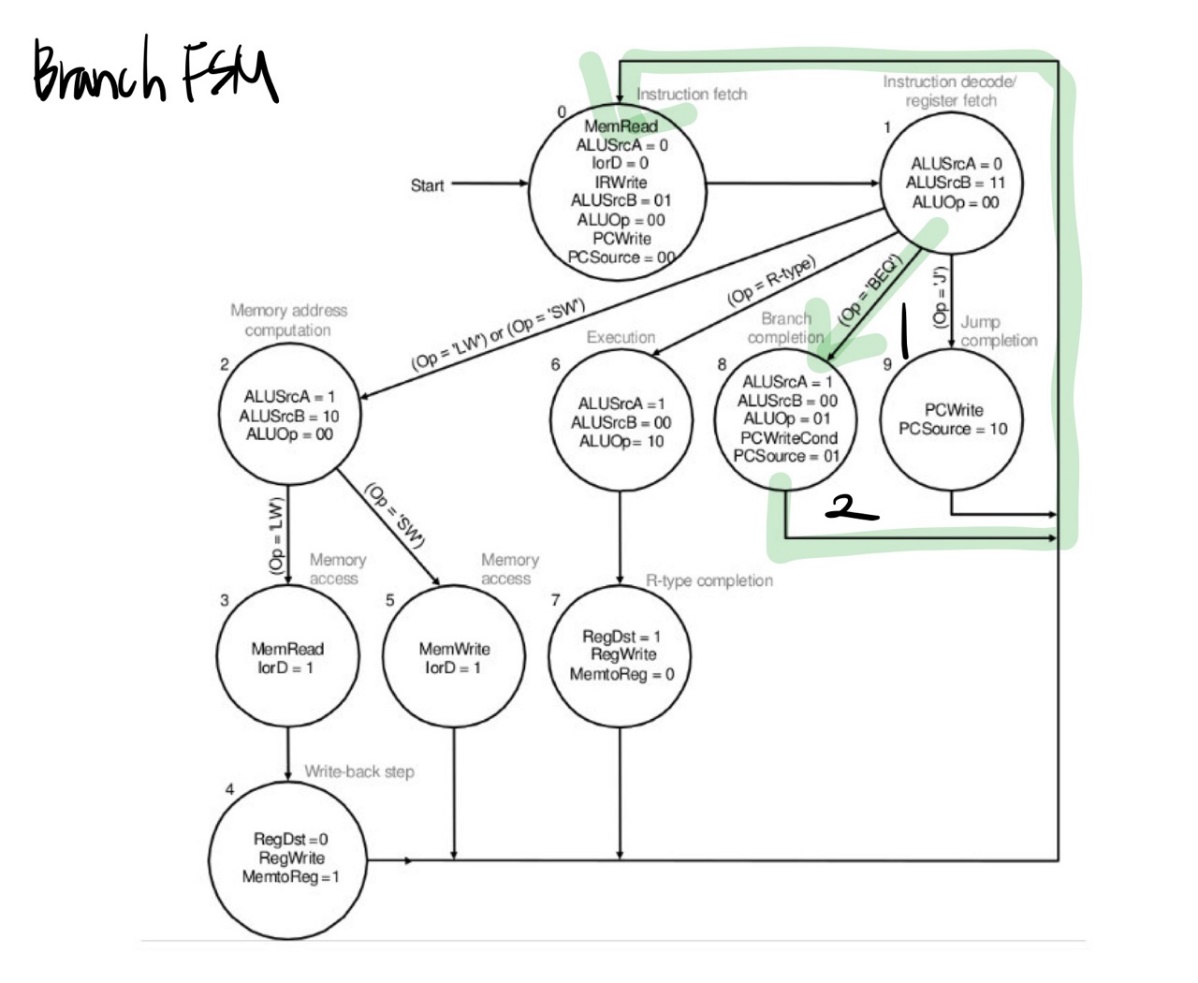
Write register Rt: Reg(Rt) ← data ($t = SE (MEM [$s + i]:2))

Next PC address: PC ← PC + 4

Operation: $t = SE (MEM [$s + i]:2)

**BLTZ(RegIm)**

[FSM]

****

Branch instruction으로 볼 수 있습니다. State는 1->6->0 순으로 진행되는 것을 알 수 있습니다. 그리고 각 state마다 해당되는 bit 변화를 변화시키는 작업을 진행합니다.

execution단계 없이 if ($s < 0) pc += i << 2 와 같은 Branch operation을 진행합니다.

(연산과정을 상세히 풀어보면 다음과 같습니다.)

Fetch instruction: Instruction ← MEM[PC]

Fetch operands: data1 ← Reg(Rs), data2 ← Reg(0)

Conditional check: if (rs < 0),

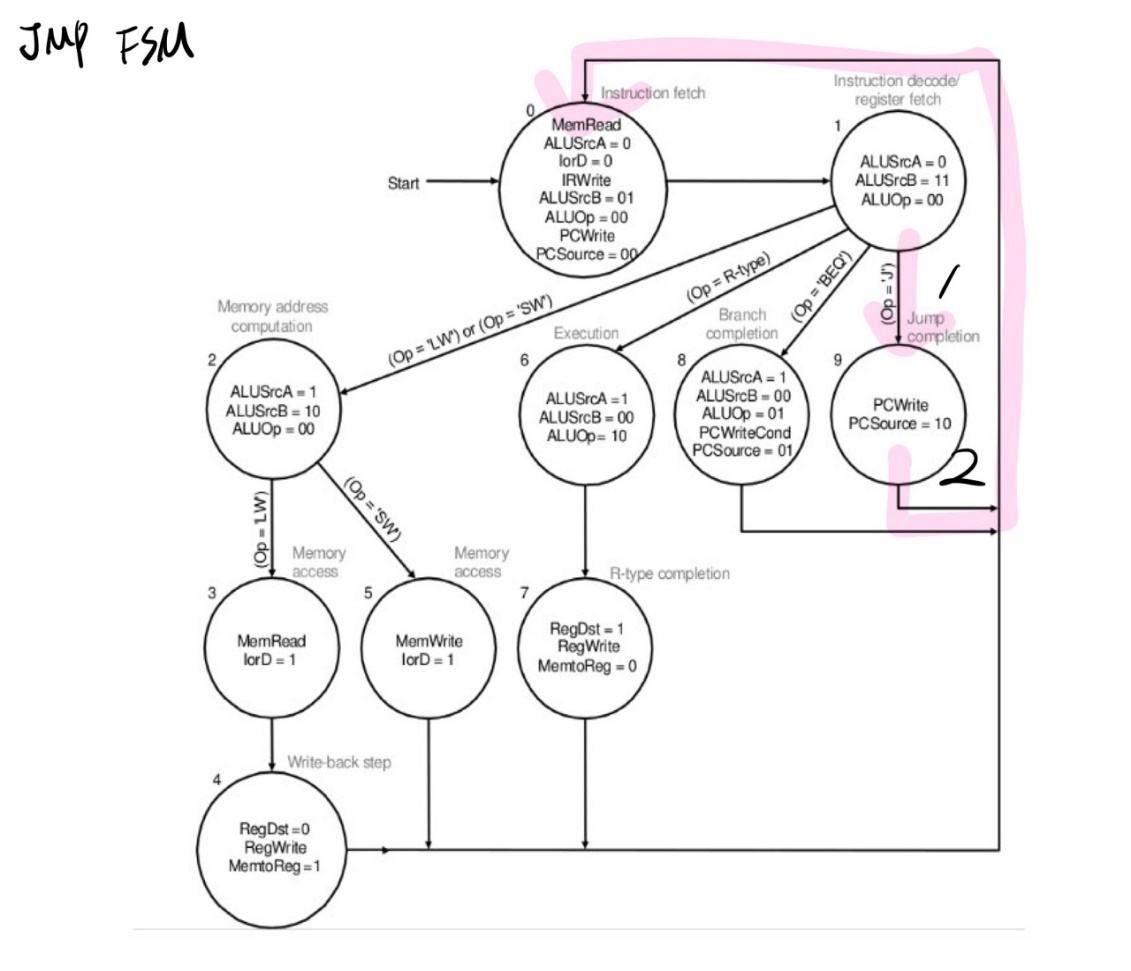
Branch: if (zero) PC ← PC + 4 + 4×sign\_ext(imm16)

else PC ← PC + 4

Operation: if ($s < 0) pc += i << 2

**JAL(jal)**

[FSM]

****

Jump instruction으로 볼 수 있습니다. State는 1->9->0 순으로 진행되는 것을 알 수 있습니다. 그리고 각 state마다 해당되는 bit 변화를 변화시키는 작업을 진행합니다.

execution단계 없이 $31 = pc; pc = pc4 | i26 << 2 와 같은 Jump operation을 진행합니다.

(연산과정을 상세히 풀어보면 다음과 같습니다.)

Fetch instruction: Instruction ← MEM[PC]

Target PC address: target ← PC[31:28] , Imm26 <<2

Jump: PC ← target

Write result: Reg($31) ← PC

Operation: $31 = pc; pc = pc4 | i26 << 2

**[설계 의도와 방법]**

<구현한 Multi Cycle CPU FSM Diagram>

위에서 그림과 함께 설명하였습니다.

< Micro-Instruction 의 Field 구분 및 Field 용도>

Micro instruction은 명령어 사이클의 각 주기에서 실행되는 마이크로 연산들에 대응되는 비트들로 이루어진 단어입니다. 그리고 이러한 micro instructions의 집합을 micro program이라고 합니다.

또한 이러한 micro instructions를 design하는 방법은 다음과 같습니다. 1. Start with list of control signals 2. Group signals together that make sense (vs. random): called “fields”

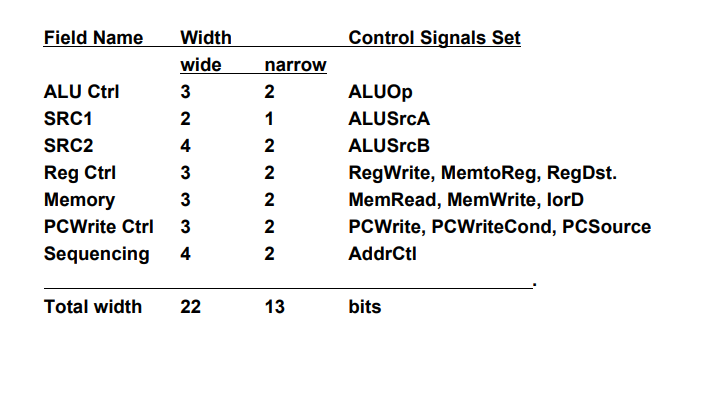
3. Places fields in some logical order (e.g., ALU operation & ALU operands first and microinstruction sequencing last)

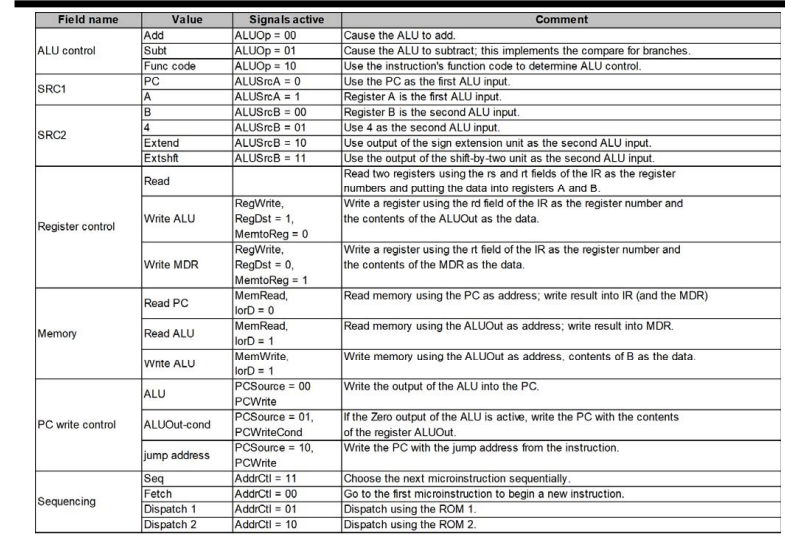
4. Create a symbolic legend for the microinstruction format, showing name of field values and how they set the control signals – Use computers to design computers

5. To minimize the width, encode operations that will never be used at the same time

또한 Micro-Instruction의 형식 즉 각 field구분과 각 field의 용도를 보면 다음과 같습니다.

Totalbits가 22일 때 즉 wide한 형식일 때의 field는 순서대로 ALU Ctrl(3) SRC1(2) SRC2(4) RegCtrl(3) Memory(3) PCWriteCtrl(3) Sequencing(4)으로 이루어지며 Totalbits가 13일 때 즉 wide한 형식일 때의 field는 순서대로 ALU Ctrl(2) SRC1(1) SRC2(2) RegCtrl(2) Memory(2) PCWriteCtrl(2) Sequencing(2)순서로 구성되는 것을 알 수 있습니다.





<Micro-Instruction 반복된 사용에 대한 분석>

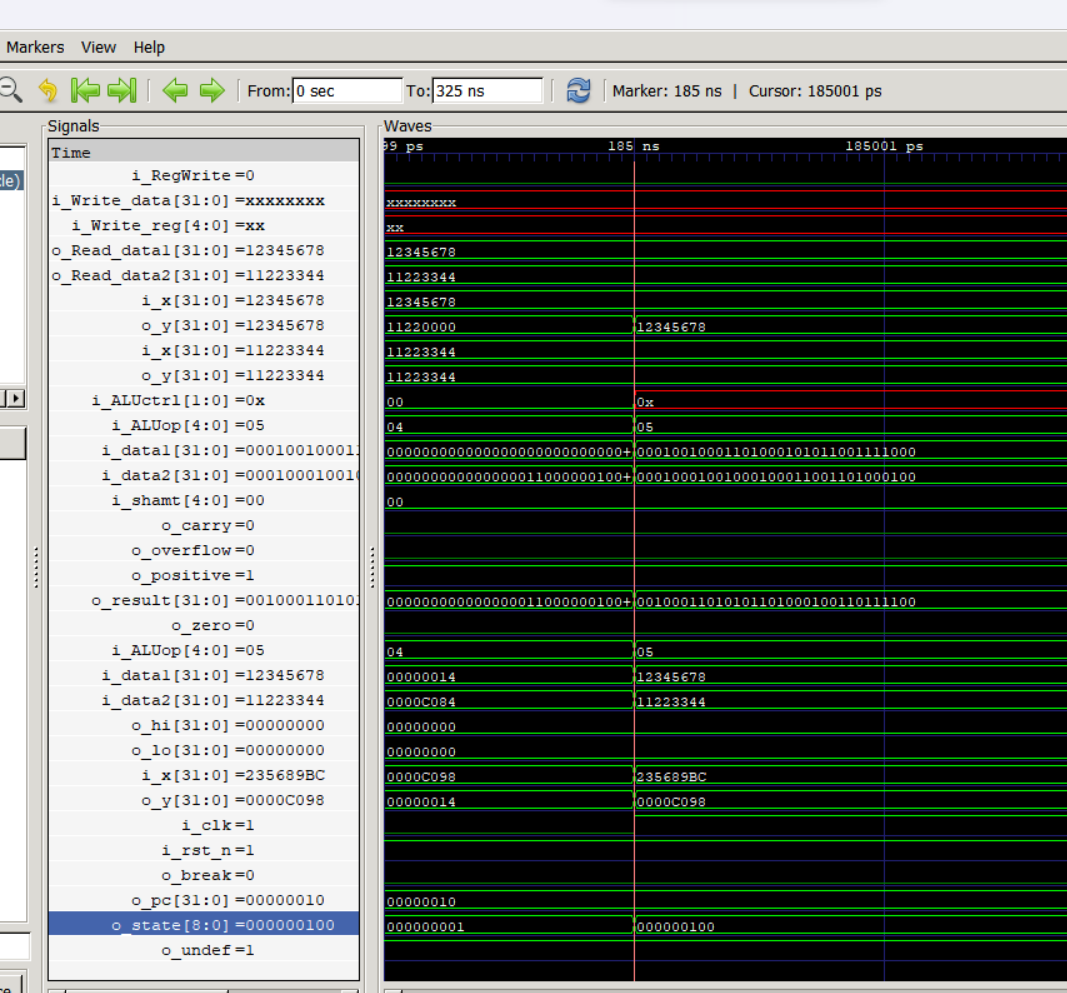
Micro-instructions set즉 Micro-program에서 장점과 같이 단점을 설명하면 다음과 같습니다. 장점으로는 설계가 용이하고, 유연성, 즉 조직, 시기, 기술의 변화에 쉽게 적응하며 설계 주기가 늦거나 현장에서 변경할 수 있다는 것과 매우 강력한 명령 집합을 구현할 수 있습니다(더 많은 control memory)

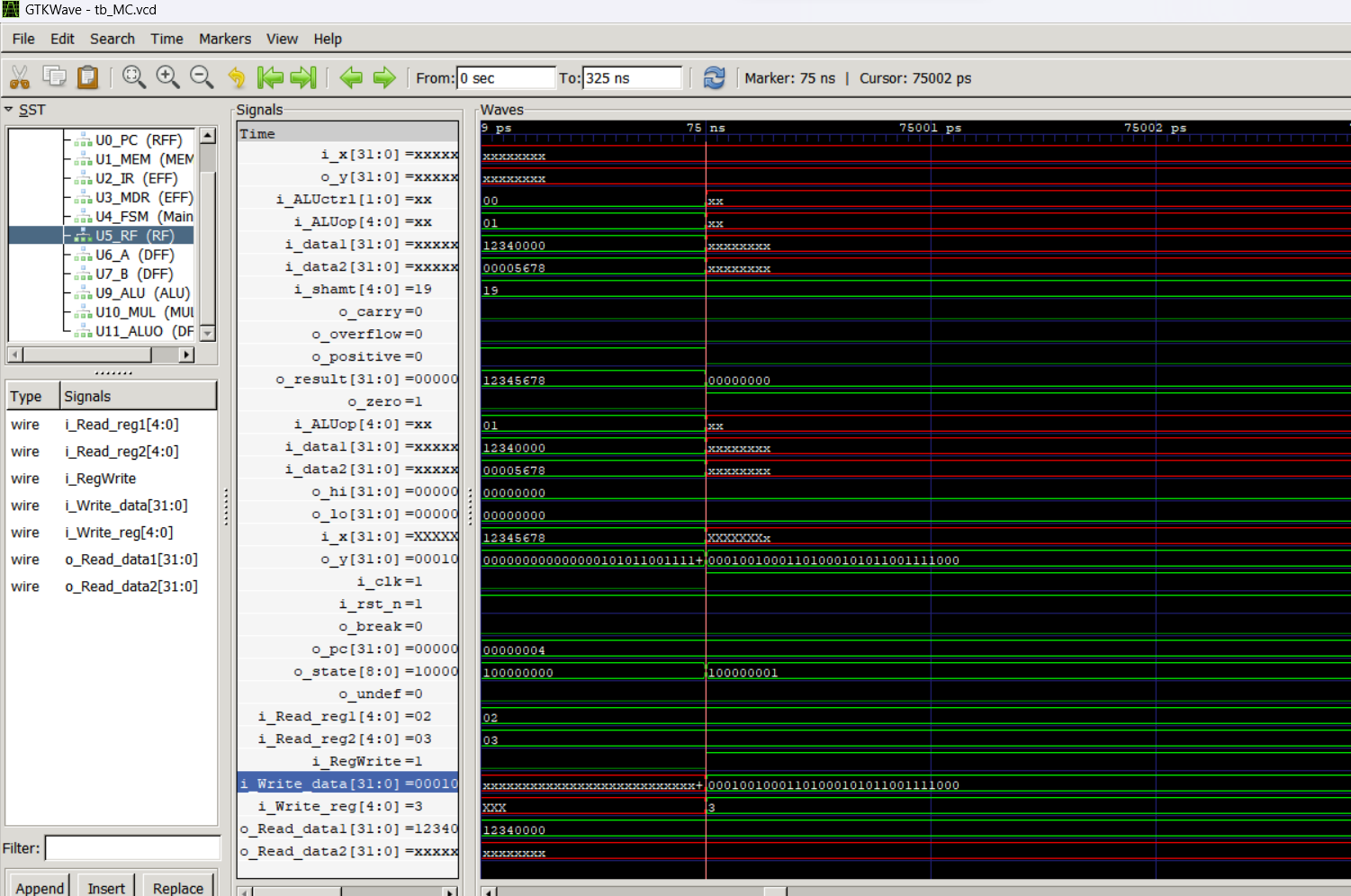
그 외 장점으로는 일반성, 여러 명령 집합을 동일한 컴퓨터에 구현할 수 있다는 것과 명령 집합을 응용 프로그램에 맞게 조정할 수 있다는 것이 있으며 호환성이 좋다는 점이 있습니다. 하지만 이러한 micro instructions의 반복된 사용은 동일한 명령 집합에 대해 많은 organization을 있게 하여서 그에 따른 구현 비용이 높고 느리다는 단점이 있습니다.

<시뮬레이션 결과와 예상 결과 비교 분석>

Addu(0x04,0x05)

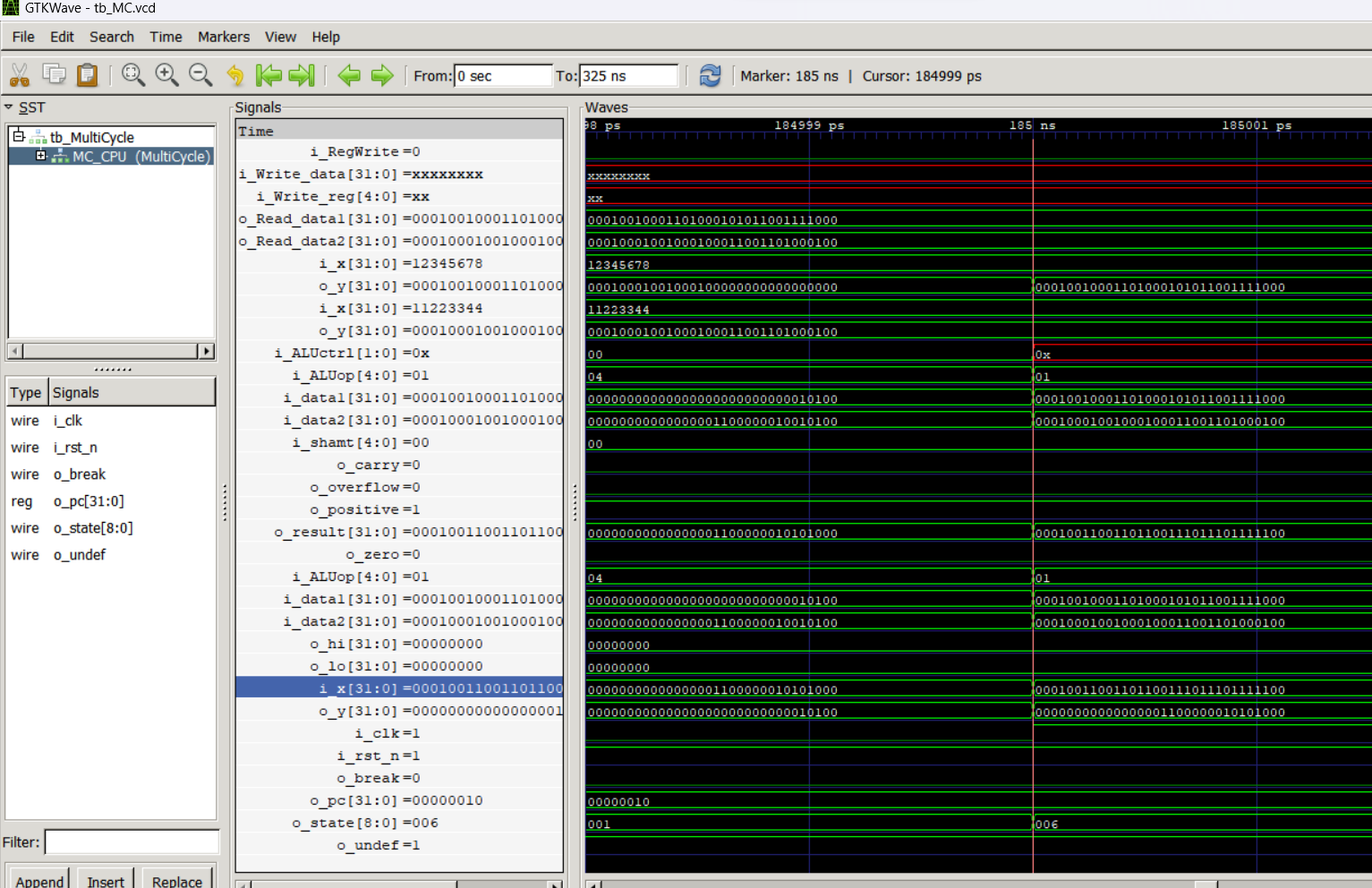
o\_state값을 보면 알 수 있듯이 해당하는 insturction에 맞는 state일때의 simulation결과가 i\_data1, i\_data2의 addu결과가 o\_result로 나온 것을 확인할 수 있었습니다. 위의 사진은 0x04번의 결과이고 그 아래 사진은 0x05의 결과인데 이때 쓸 rd의 주소로 3번을 설정하였고 그 결과가 3번에 저장된 것을 알 수 있습니다..

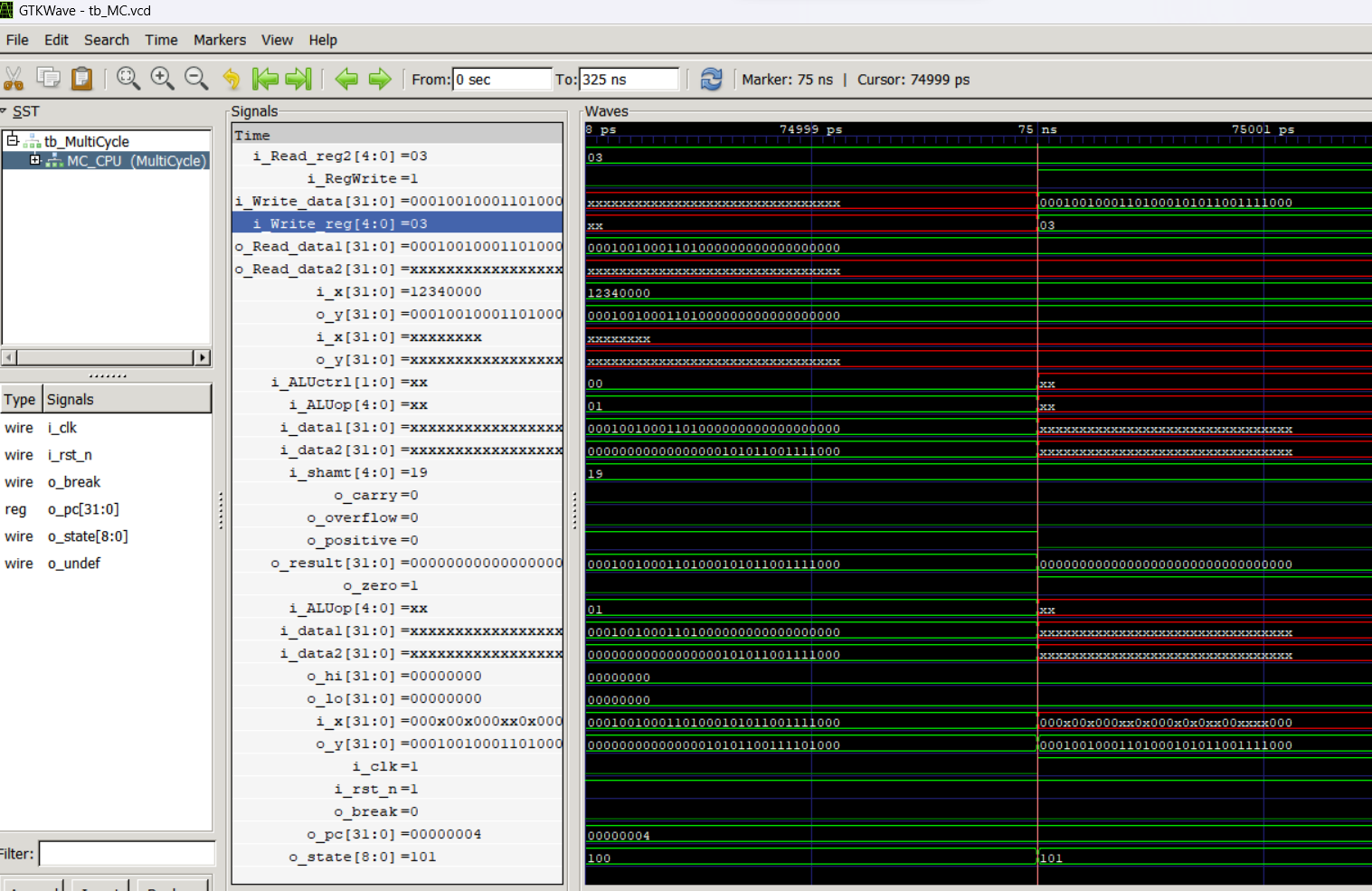




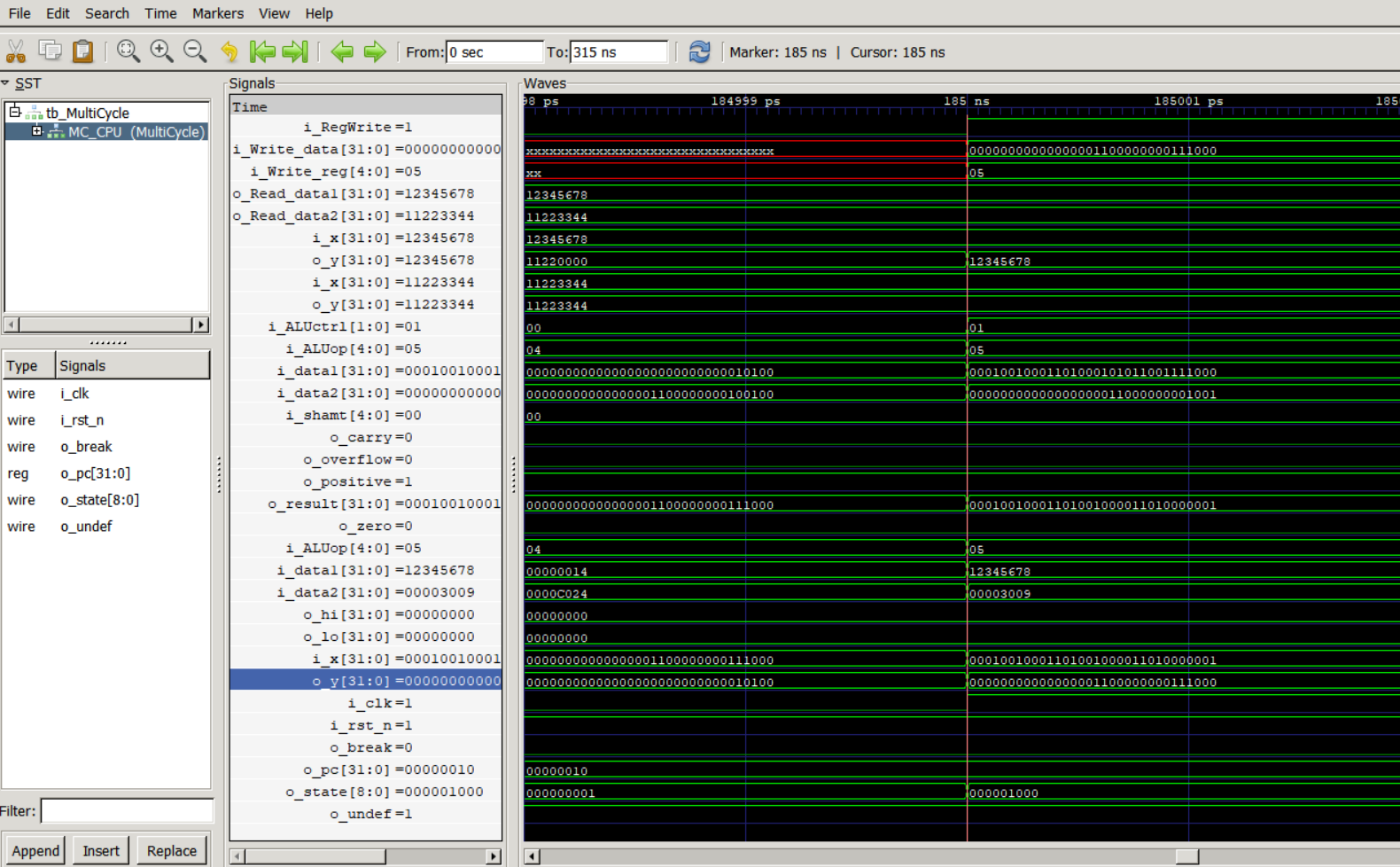
OR(0x06,0x07)

State가 0x06일 때 Or연산이 진행된 것을 확인 할 수 있었습니다. i\_data1, i\_data2를 입력으로 넣었을 때 결과 o\_result의 결과가 비트연산이 진행된 것을 확인 할 수 있습니다. 그리고 그 다음 그림은 rd에 해당하는 3 register에 연산결과가 저장된 것을 알 수 있습니다.



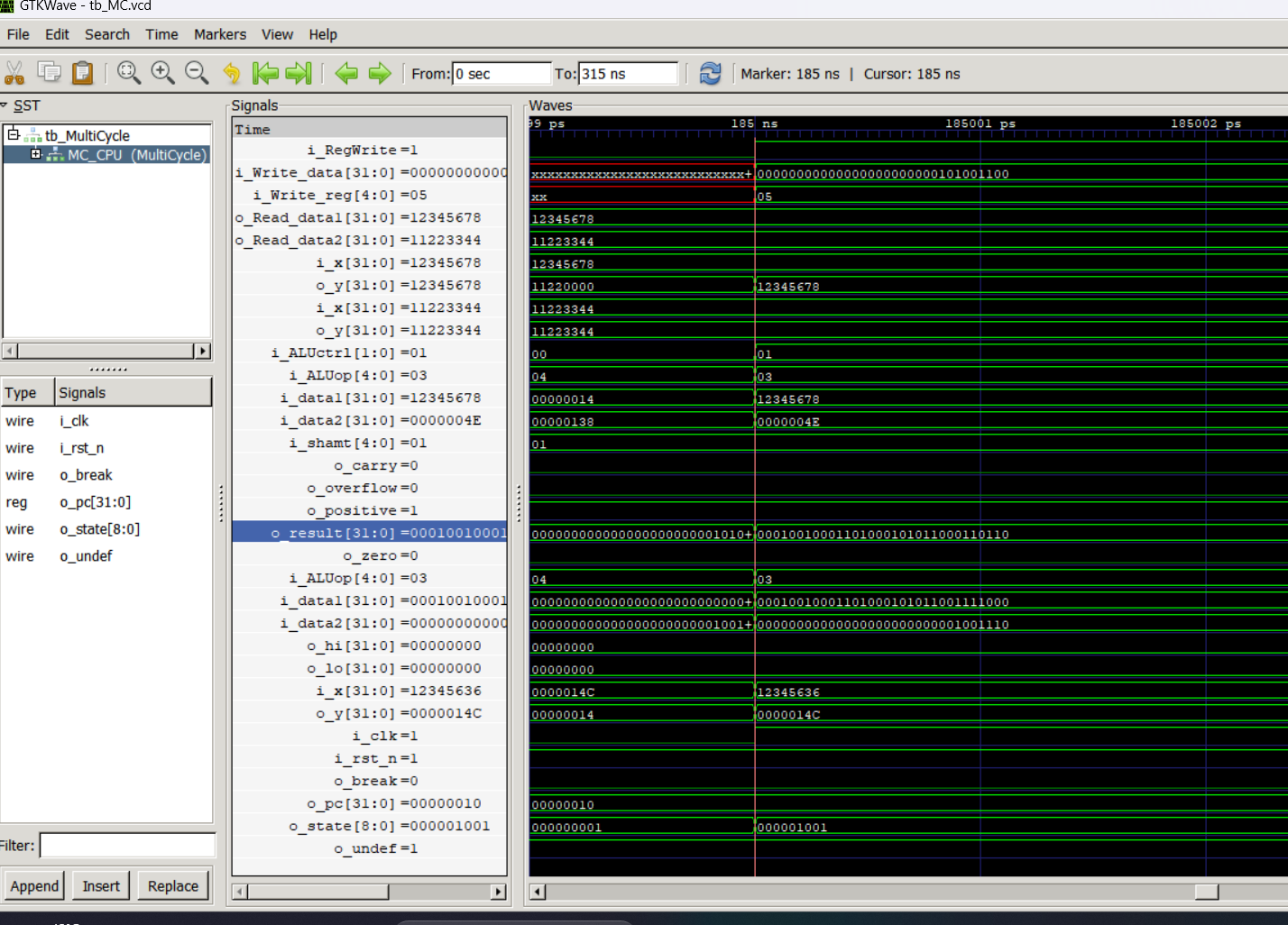


Addiu(0x08) i\_data1에 들어간 값과 I-type 에서 $t = $s +SE(i)의 연산결과와 그에 해당하는 register, 05번에 연산결과가 저장하는 과정을 거치는 것을 알 수 있습니다.



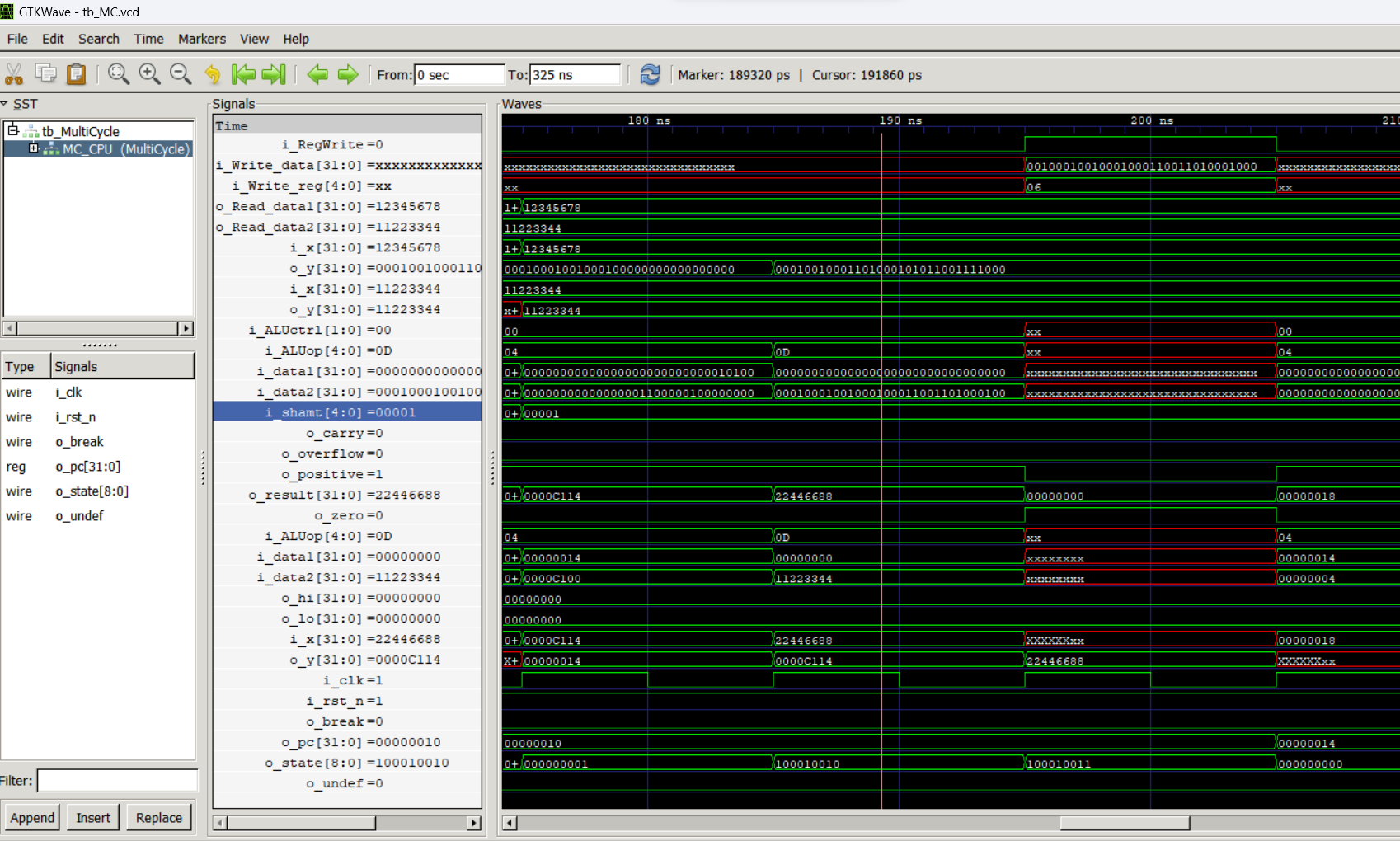
XORI(0x09)

State가 binary로 표현된 0x09번일 때 i\_data1과 i\_data2의 xori결과가 5번 register $t에 저장된 것을 알 수 있습니다.



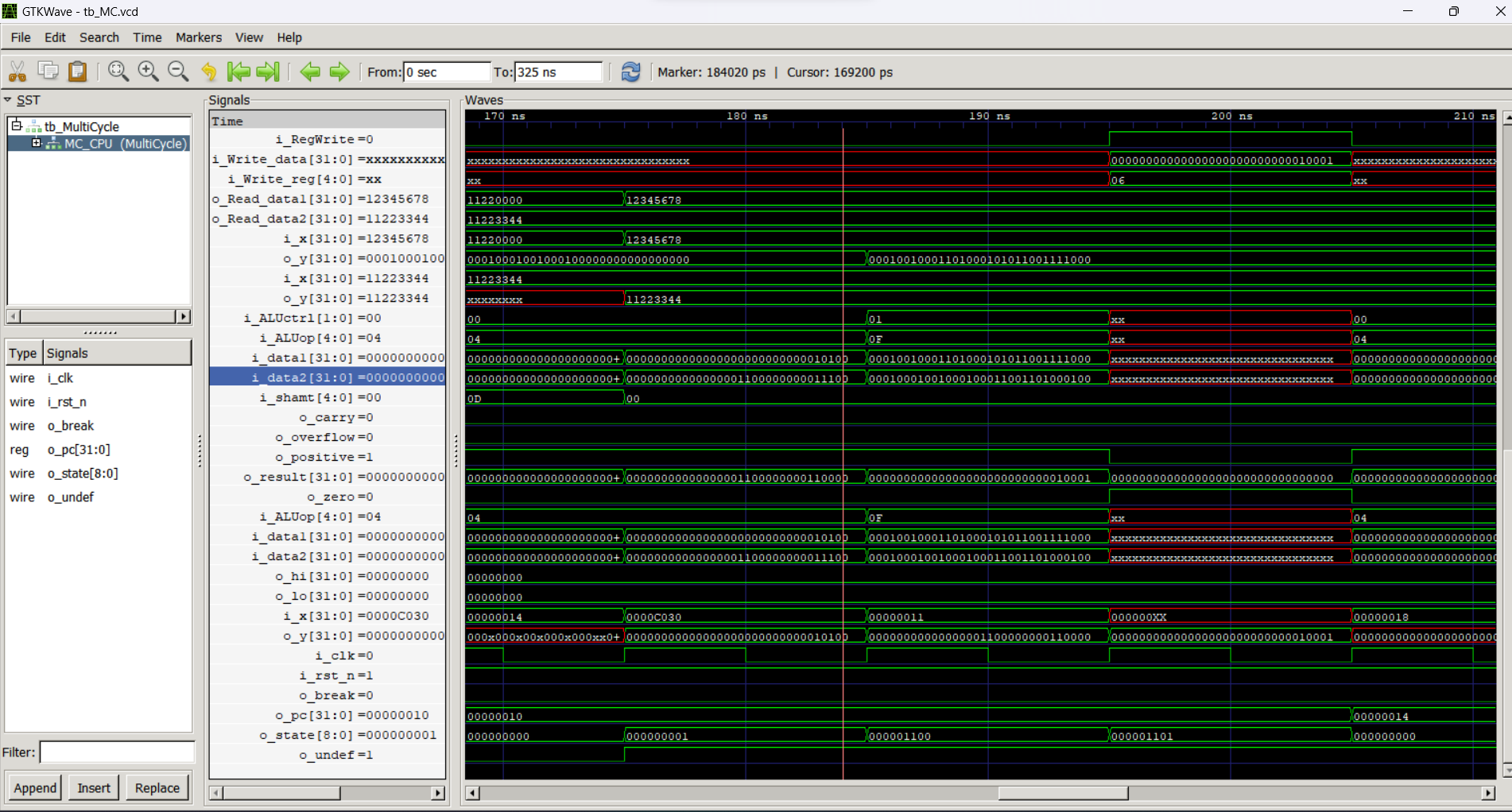
SLL

표시된 현재 state에서 ‘$t<<a’ 의 연산결과가 다음 state에서 rd $d 06에 write된 것을 알 수 있습니다.



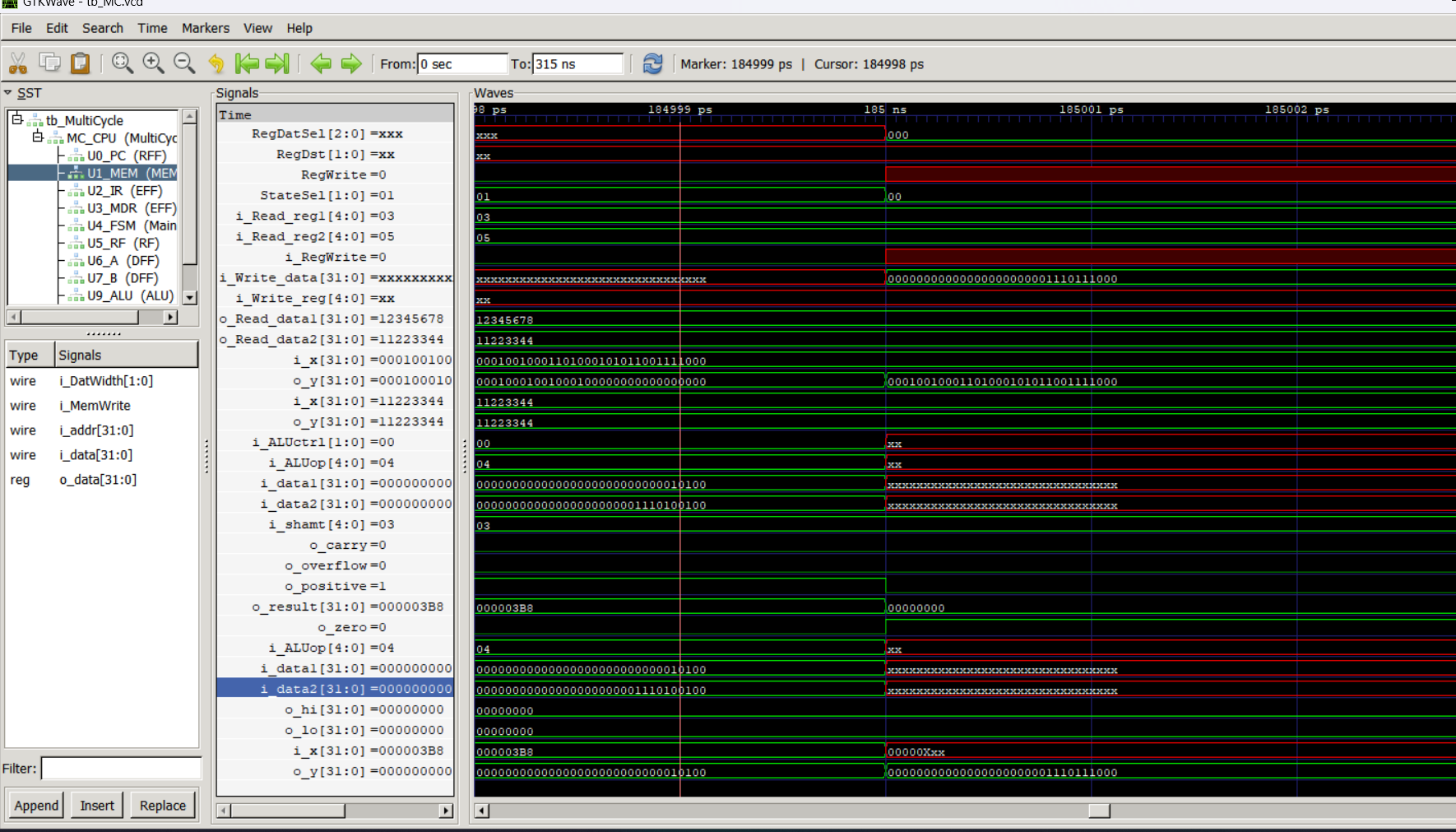
SRAV

표시된 현재 state에서 ‘$t>>>a’ 의 연산결과가 다음 state에서 rd $d 06에 write된 것을 알 수 있습니다.



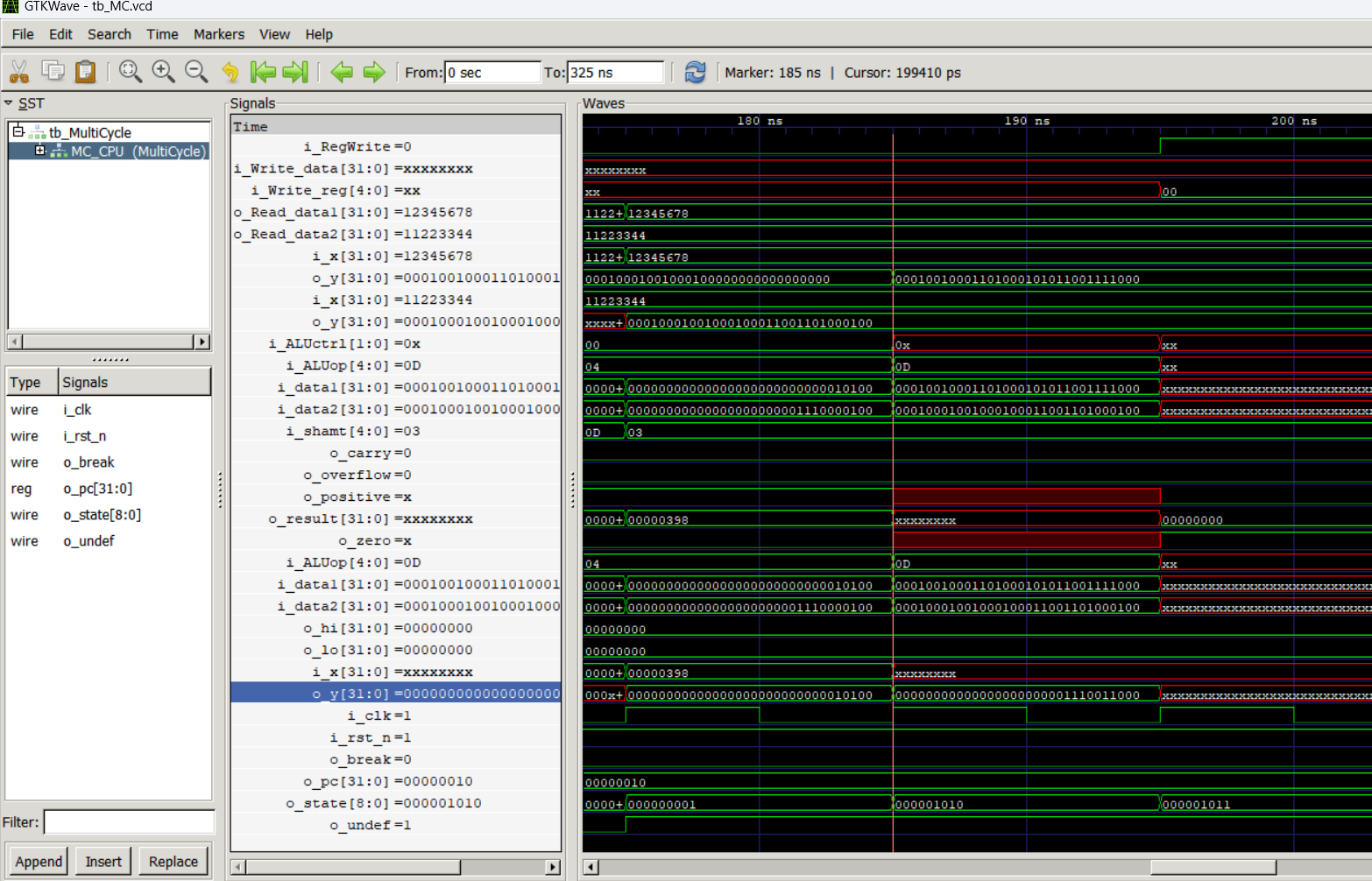
SH

MEM[$s + i]:2 = LH($t) $t에 해당하는 half word를 load해서 memory에 저장하는 역할을 합니다. 주어진 주소에 해당하는 메모리에 저장하는 과정을 거칩니다.



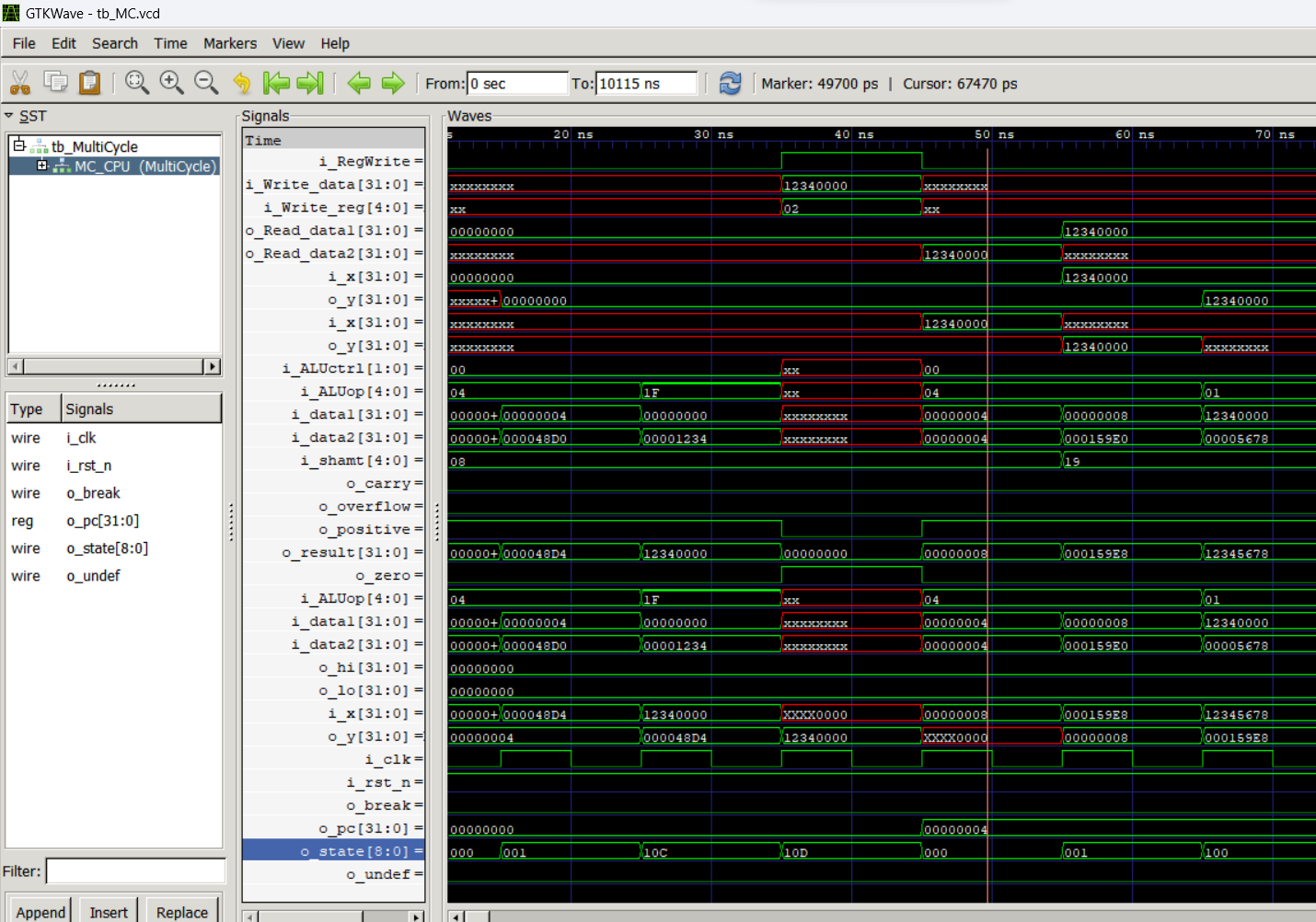
LH

$t = SE(MEM[$s+i]:2) MEM[$s+i]:2의 값을 sign extend한 뒤 $t에 저장(대입)하는 과정을 거칩니다. Load의 과정은 공통적으로 거치는 if->decode이후 execution->memory access->memory read completion의 과정을 거칩니다. 아래그림에서는 0x10과 0x11 즉 execution과 memory access단계가 표현된 것을 확인 할 수 있습니다.



BLTZ

If($s<0) pc+= i<<2 로 $s의 data가 음수여야 해당 state에 대한 simulation결과를 확인할 수 있습니다.

JAL

$31 = pc; pc = pc4 | i26<<2 와 같은 instruction은 0x14 즉, binary로 표현된 값에 해당하는 state일일 때 해당작업을 진행하는 것을 알 수 있습니다.

