

Project 3: Problem 1

프로그램 실행 이후, 모든 register data의 변화는 그 파형을 이미지로 첨부하였으며, 이미지를 첨부한 순서는 실제 **register data 변화의 순서와 동일**합니다.

추가적으로, stack layout 및 call graph 이미지는 problem1의 reference 폴더에 첨부했습니다.

개요

- 본 프로젝트의 목적은 `problem1.c`의 순차적인 function call sequence에 따라 function call이 발생할 때, calling convention에 따라 변화하는 RISC-V register data 및 stack layout의 변화를 확인하고, 분석하는 것이다.
- 결과 분석 이전에 calling convention에 대해 간략하게 설명하겠다.
- 이는 function을 call할 때, 지켜야 하는 공통의 규약으로, 크게 다음의 6가지 step으로 나누어 생각할 수 있다.

Procedure Calling: Required Steps (Linkage Convention)

1. Place parameters in registers x10 to x17 (set function arguments)
2. Transfer control to procedure: jal ProcedureLabel
 1. Address of the following instruction (return address) put in x1 (ra)
 2. Jumps to the target address
3. Acquire storage for procedure (getting a stack)
4. Perform procedure's operations
5. Place result in register for caller
6. Return to place of call (address in x1): jalr x0, 0(x1)
 - Like jal, but jumps to 0 + address in x1
 - Use x0 as rd (x0 cannot be changed)
 - Can also be used for computed jumps

Registers
x0: the constant value 0
x1: return address
x2: stack pointer
x3: global pointer
x4: thread pointer
x5 – x7, x28 – x31: temporaries
x8: frame pointer
x9, x18 – x27: saved registers
x10 – x11: function arguments/results
x12 – x17: function arguments



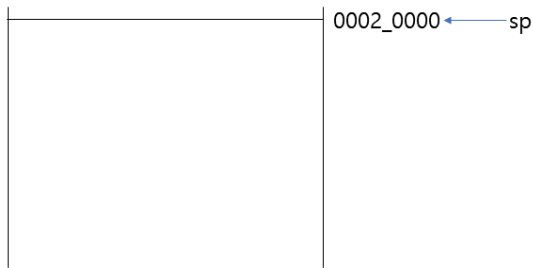
13



그림 1. Calling Convention

<fib-0x14>

<fib-0x14>



Register data

Reg	Data
ra	0000_0000
sp	0000_0000 -> 0002_0000
s0	0000_0000
a0	0000_0000
a4	0000_0000
a5	0000_0000
이외의 reg	0000_0000

*전부 hexadecimal 표기입니다.

그림 2. <fib-0x14> Stack Layout & Register Data

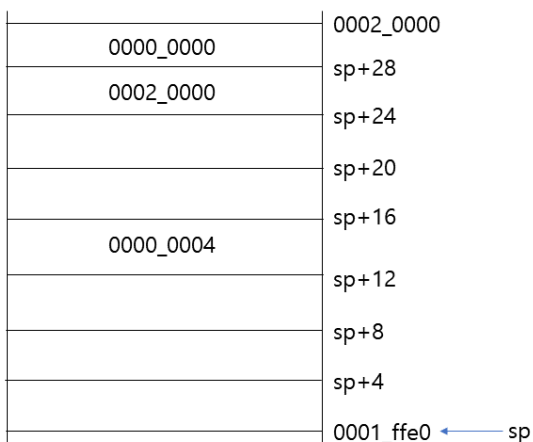
- 전체 프로그램 실행을 위한 stack pointer 값을 assign하고, main으로 jump한다.
 - sp 0000_0000 → 0002_0000

x2_sp_w	00000000	00000000	00020000					
---------	----------	----------	----------	--	--	--	--	--

- problem1.c 를 참고하면, int main (void) 로 정의되어 있으므로, 별도의 argument setting은 하지 않는다.
- ra 또한 그대로 0000_0000이기 때문에, x1 register(ra)의 value도 변하지 않는다.

<main>

<main>



Register data

Reg	Data
ra	0000_0000 -> 0000_00ac
sp	0002_0000 -> 0001_ffe0
s0	0000_0000 -> 0002_0000
a0	0000_0000 -> 0000_0003
a4	0000_0000
a5	0000_0000 -> 0000_00004 -> 0000_0003
이외의 reg	0000_0000

*전부 hexadecimal 표기입니다.

그림 3. <main> Stack Layout & Register Data

- main 함수의 실행을 위한 stack을 push한다.

- 이 과정은 그림 1. Calling Convention의 세 번째 step에 해당하는 부분임을 확인할 수 있다.

84: fe010113 addi sp,sp,-32

- sp 0002_0000 → 0001_ffe0

x2_sp_w	0001ffe0	00020000	0001ffe0																	
---------	----------	----------	----------	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

- 함수 내에서 function call이 발생하게 되면, 각 register 값이 overwrite되기 때문에, 이를 고려하여 각 register data의 값을 stack에 save해준다.

- stack에 ra, s0 register 값을 save.

- 이는 그림 3의 main 함수 stack layout에서 각 memory에 저장되는 값을 통해 확인할 수 있다.

- 변화한 stack pointer의 값에 맞게 frame pointer의 값도 변경한다.

90: 02010413 addi s0,sp,32

- s0 0000_0000 → 0002_0000

x8_s0_w	00020000	00000000	00020000																	
---------	----------	----------	----------	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

- problem1.c에 정의된 main 함수의 procedure를 perfume한다. 이 과정은 그림 1. Calling Convention의 네 번째 step에 해당하는 부분임을 확인할 수 있다.

- int n = 4에 따라 a5 register에 0000_0004를 load & spill한다.

- stack에 save된 register data는 그림 3의 main 함수 stack layout을 통해 확인할 수 있다.

- a5 0000_0000 → 0000_0004

x15_a5_w	00000004	00000000	00000004																	
----------	----------	----------	----------	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

- 이후, x1 = fib(n-1)의 fib(n-1)에 따라 다음을 실행한다. (Argument setting, Jump to the target address)

a0: fff78793 addi a5,a5,-1

- a5 0000_0004 → 0000_0003

x15_a5_w	00000004	00000004	00000003																	
----------	----------	----------	----------	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

a4: 00078513 mv a0,a5

- a0 0000_0000 → 0000_0003

x10_a0_w	00000000	00000000	00000003																	
----------	----------	----------	----------	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

- 위의 과정은 그림 1. Calling Convention의 첫 번째 step에 해당하는 부분으로, function call에 필요한 argument register 값을 setting하는 과정임을 확인할 수 있다.

a8: f6dff0ef jal ra,14 <fib>

- ra 0000_0000 → 0000_00ac

x1_ra_w	00000000	00000000	000000ac																	
---------	----------	----------	----------	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

- 위의 과정은 그림 1. Calling Convention의 두 번째 step에 해당하는 부분으로, function call 전에 return address(ra)를 setting해주고, target address로 jump하여 function call을 수행하는 과정임을 확인할 수 있다.

<fib>, n = 3

<fib>, n(a0) = 3

Register data

0000_00ac	0001_ffe0
0001_ffe0	sp+28
0000_0000	sp+24
	sp+20
	sp+16
0000_0003	sp+12 [-20(s0)]
	sp+8
	sp+4
	0001_ffc0 ← sp

Reg	Data
ra	0000_00ac -> 0000_0050
sp	0001_ffe0 -> 0001_ffc0
s0	0002_0000 -> 0001_ffe0
a0	0000_0003 -> 0000_0002
a4	0000_0000 -> 0000_0003
a5	0000_0003 -> 0000_0001 -> 0000_0003 -> 0000_0002
이외의 reg	0000_0000

*전부 hexadecimal 표기입니다.

그림 4. <fib>, n = 3 Stack Layout & Register Data

- fib 함수의 실행을 위한 stack을 push한다.
 - 이 과정은 그림 1. Calling Convention의 세 번째 step에 해당하는 부분임을 확인할 수 있다.
 - 14: fe010113 addi sp,sp,-32 # 1ffe0 <main+0x1ff5c>
 - sp 0001_ffe0 → 0001_ffc0

x2_sp_w	0001ffe0	0001ffe0		0001ffc0	
---------	----------	----------	--	----------	--

- 함수 내에서 function call이 발생하게 되면, 각 register 값이 overwrite되기 때문에, 이를 고려하여 각 register data의 값을 stack에 save해준다.
 - stack에 ra, s0, s1 register 값을 stack에 save.
 - 이는 그림 4의 fib 함수 stack layout에서 각 memory에 저장되는 값을 통해 확인할 수 있다.
- 변화한 stack pointer의 값에 맞게 frame pointer의 값도 변경한다.
 - 24: 02010413 addi s0,sp,32
 - s0 0002_0000 → 0001_ffe0

x8_s0_w	00020000	00020000	0001ffe0
---------	----------	----------	----------

- problem1.c에 정의된 fib 함수의 procedure를 perfume한다. 이 과정은 그림 1. Calling Convention의 네 번째 step에 해당하는 부분임을 확인할 수 있다.
 - caller로 부터 넘겨 받은 argument 값을 stack에 save한다. (추가적인 function call에 따른 register overwrite 고려)
 - 28: fea42623 sw a0,-20(s0)
 - 이는 그림 4의 fib 함수 stack layout에서 각 memory에 저장되는 값을 통해 확인할 수 있다.
 - 이후, if (n <= 1)에 따라 다음을 실행한다.
 - 2c: fec42703 lw a4,-20(s0)
 - a4 0000_0000 → 0000_0003

x14_a4_w	00000000	00000000	00000003
----------	----------	----------	----------

- 30: 00100793 li a5,1

- a5 0000_0003 → 0000_0001

x15_a5_w	00000003	00000003	00000001	
----------	----------	----------	----------	--

- 34: 00e7c663 blt a5,a4,40 <fib+0x2c>

- a5의 값은 1이고, a4의 값은 3이기 때문에, 40으로 jump한다. (if를 건너뛴다)

- 이후, fib(n-1)에 따라 다음을 실행한다.

- 40: fec42783 lw a5,-20(s0)

- a5 0000_0001 → 0000_0003

x15_a5_w	00000001	00000001	00000003	
----------	----------	----------	----------	--

- 44: fff78793 addi a5,a5,-1

- a5 0000_0003 → 0000_0002

x15_a5_w	00000003	00000003	00000002	
----------	----------	----------	----------	--

- 48: 00078513 mv a0,a5

- a0 0000_0003 → 0000_0002

x10_a0_w	00000003	00000003	00000002	
----------	----------	----------	----------	--

- 위의 과정은 그림 1. Calling Convention의 첫 번째 step에 해당하는 부분으로, function call에 필요한 argument register 값을 setting하는 과정임을 확인할 수 있다.

- 4c: fc9ff0ef jal ra,14 <fib>

- ra 0000_00ac → 0000_0050

x1_ra_w	000000ac	000000ac	00000050	
---------	----------	----------	----------	--

- 위의 과정은 그림 1. Calling Convention의 두 번째 step에 해당하는 부분으로, function call 전에 return address(ra)를 setting해주고, target address로 jump하여 function call을 수행하는 과정임을 확인할 수 있다.

<fib>, n = 2

<fib>, n(a0) = 2

Register data

0000_0050	0001_ffc0
0001_ffc0	sp+28
0000_0000	sp+24
	sp+20
	sp+16
0000_0002	sp+12 [-20(s0)]
	sp+8
	sp+4
	0001_ffa0 ← sp

Reg	Data
ra	0000_0050
sp	0001_ffc0 -> 0001_ffa0
s0	0001_ffe0 -> 0001_ffc0
a0	0000_0002 -> 0000_0001
a4	0000_0003 -> 0000_0002
a5	0000_0002 -> 0000_0001 -> 0000_0002 -> 0000_0001
이외의 reg	0000_0000

*전부 hexadecimal 표기입니다.

그림 5. <fib>, n = 2 Stack Layout & Register Data

- fib 함수의 실행을 위한 stack을 push한다.
 - 이 과정은 그림 1. Calling Convention의 세 번째 step에 해당하는 부분임을 확인할 수 있다.
 - 14: fe010113 addi sp,sp,-32 # 1ffe0 <main+0x1ff5c>
 - sp 0001_ffc0 → 0001_ffa0

x2_sp_w	0001ffc0	0001ffc0	0001ffa0
---------	----------	----------	----------

- 함수 내에서 function call이 발생하게 되면, 각 register 값이 overwrite되기 때문에, 이를 고려하여 각 register data의 값을 stack에 save해준다.
 - stack에 ra, s0, s1 register 값을 stack에 save.
 - 이는 그림 5의 fib 함수 stack layout에서 각 memory에 저장되는 값을 통해 확인할 수 있다.
- 변화한 stack pointer의 값에 맞게 frame pointer의 값도 변경한다.
 - 24: 02010413 addi s0,sp,32
 - s0 0001_ffe0 → 0001_ffc0

x8_s0_w	0001ffe0	0001ffe0	0001ffc0
---------	----------	----------	----------

- problem1.c에 정의된 fib 함수의 procedure를 perfume한다. 이 과정은 그림 1. Calling Convention의 네 번째 step에 해당하는 부분임을 확인할 수 있다.
 - caller로 부터 넘겨 받은 argument 값을 stack에 save한다. (추가적인 function call에 따른 register overwrite 고려)
 - 28: fea42623 sw a0,-20(s0)
 - 이는 그림 5의 fib 함수 stack layout에서 각 memory에 저장되는 값을 통해 확인할 수 있다.
 - 이후, if (n <= 1)에 따라 다음을 실행한다.
 - 2c: fec42703 lw a4,-20(s0)
 - a4 0000_0003 → 0000_0002

x14_a4_w	00000003	00000003	00000002
----------	----------	----------	----------

- 30: 00100793 li a5,1

- a5 0000_0002 → 0000_0001



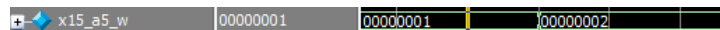
- 34: 00e7c663 blt a5,a4,40 <fib+0x2c>

- a5의 값은 1이고, a4의 값은 2이기 때문에, 40으로 jump한다. (if를 건너뛰다)

- 이후, fib(n-1)에 따라 다음을 실행한다.

- 40: fec42783 lw a5,-20(s0)

- a5 0000_0001 → 0000_0002



- 44: fff78793 addi a5,a5,-1

- a5 0000_0002 → 0000_0001



- 48: 00078513 mv a0,a5

- a0 0000_0002 → 0000_0001



- 위의 과정은 그림 1. Calling Convention의 첫 번째 step에 해당하는 부분으로, function call에 필요한 argument register 값을 setting하는 과정임을 확인할 수 있다.

- 4c: fc9ff0ef jal ra,14 <fib>

- 동일한 함수를 재귀적으로 call하는 것이기 때문에 ra의 값은 변하지 않고, 그대로 0000_0050이다.
- 위의 과정은 그림 1. Calling Convention의 두 번째 step에 해당하는 부분으로, function call 전에 return address(ra)를 setting해주고, target address로 jump하여 function call을 수행하는 과정임을 확인할 수 있다.

<fib>, n = 1

<fib>, n(a0) = 1

Register data

0000_0050	0001_ffa0
0001_ffa0	sp+28
0000_0000	sp+24
	sp+20
	sp+16
0000_0001	sp+12 [-20(s0)]
	sp+8
	sp+4
	0001_ff80 ← sp

Reg	Data
ra	0000_0050
sp	0001_ffa0 -> 0001_ff80 -> 00001_ffa0
s0	0001_ffc0 -> 0001_ffa0 -> 0001_ffc0
a0	0000_0001
a4	0000_0002 -> 0000_0001
a5	0000_0001
이외의 reg	0000_0000

*전부 hexadecimal 표기입니다.

그림 6. <fib>, n = 1 Stack Layout & Register Data

- fib 함수의 실행을 위한 stack을 push한다.
 - 이 과정은 그림 1. Calling Convention의 세 번째 step에 해당하는 부분임을 확인할 수 있다.
 - 14: fe010113 addi sp,sp,-32 # 1ffe0 <main+0x1ff5c>
 - sp 0001_ffa0 → 0001_ff80



x2_sp_w	0001ffa0	0001ffa0	0001ff80
---------	----------	----------	----------

- 함수 내에서 function call이 발생하게 되면, 각 register 값이 overwrite되기 때문에, 이를 고려하여 각 register data의 값을 stack에 save해준다.
 - stack에 ra, s0, s1 register 값을 stack에 save.
 - 이는 그림 6의 fib 함수 stack layout에서 각 memory에 저장되는 값을 통해 확인할 수 있다.
- 변화된 stack pointer의 값에 맞게 frame pointer의 값도 변경한다.
 - 24: 02010413 addi s0,sp,32
 - s0 0001_ffc0 → 0001_ffa0

x8_s0_w	0001ffc0	0001ffc0	0001ffa0
---------	----------	----------	----------

- problem1.c에 정의된 fib 함수의 procedure를 perfume한다. 이 과정은 그림 1. Calling Convention의 네 번째 step에 해당하는 부분임을 확인할 수 있다.
 - caller로 부터 넘겨 받은 argument 값을 stack에 save한다. (추가적인 function call에 따른 register overwrite 고려)
 - 28: fea42623 sw a0,-20(s0)
 - 이는 그림 6의 fib 함수 stack layout에서 각 memory에 저장되는 값을 통해 확인할 수 있다.
 - 이후, if (n <= 1)에 따라 다음을 실행한다.
 - 2c: fec42703 lw a4,-20(s0)
 - a4 0000_0002 → 0000_0001

x14_a4_w	00000002	00000002	00000001
----------	----------	----------	----------

- 30: 00100793 li a5,1
 - 이전에도 a5의 값은 0000_0001이기 때문에, 명령 실행 이후의 a5 register의 값 또한 그대로 0000_0001이다.
 - 34: 00e7c663 blt a5,a4,40 <fib+0x2c>
 - a5의 값은 1이고, a4의 값은 1이기 때문에, 40으로 jump하지 않고, next instruction을 실행한다. (if 건너 뛰지 않음)
 - 이후, return n 에 따라 다음을 실행한다.
 - 38: fec42783 lw a5,-20(s0)
 - 이전에도 a5의 값은 0000_0001이기 때문에, 명령 실행 이후의 a5 register의 값 또한 그대로 0000_0001이다.
 - 3c: 0300006f j 6c <fib+0x58>
 - 6c로 jump한다.
 - 6c: 00078513 mv a0,a5
 - 이전에도 a0의 값은 0000_0001이기 때문에, 명령 실행 이후의 a0 register의 값 또한 그대로 0000_0001이다.
 - 위의 과정은 그림 1. Calling Convention의 다섯 번째 step에 해당하는 부분으로, caller에게 return 값을 전달 해주기 위해 result register를 setting하는 과정임을 확인할 수 있다.
 - stack을 pop하기 전에, stack에 저장해주었던 overwrite되기 전의 register 값을 해당되는 register로 load한다. 이는 다음과 같다.
 - 70: 01c12083 lw ra,28(sp)
 - 74: 01812403 lw s0,24(sp)
 - s0 0001_ffa0 → 0001_ffc0
- 
- 78: 01412483 lw s1,20(sp)
- 마지막으로, result를 반환하며 load한 return address(ra)로 jump하고, stack을 pop한다. 이 과정은 calling convention의 여섯 번째 step에 해당하는 부분임을 확인할 수 있으며, 다음과 같다.
 - 7c: 02010113 addi sp,sp,32
 - stack pointer의 값을 변화시켜, stack을 pop한다.
 - sp 0001_ff80 → 0001_ffa0
- 
- 80: 00008067 ret
 - ra로 jump한다.
- testbench의 다음 코드에 의해 0000_8067 을 실행한 뒤, 프로그램 실행이 종료된다.

```
if(debug_inst == 32'h0000_8067)begin
    wait_clocks(5);
    $finish();
end
```

- 이때 실행된 함수의 call graph를 통해 살펴보면, 다음과 같다.

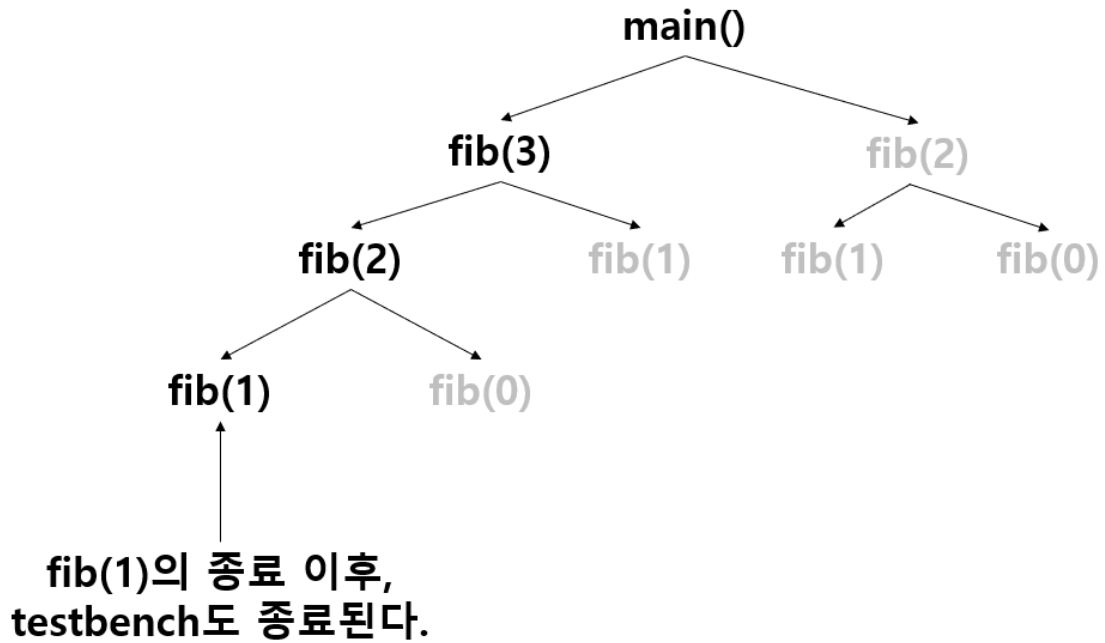


그림 7. Function Call graph of Problem 1

Problem 2 (`riscv_HPC.V` 및 `problem2_2.png` 첨부)

- Input으로 받은 opcode를 기준으로 다음과 같이 type별로 구분하여, counting하였다.

RV32I Base Instruction Set										
imm[31:12]				rd	0110111	LUI	U-type			
imm[31:12]				rd	0010111	AUIPC	J-type			
imm[20:10:1 11 19:12]				rd	1101111	JAL				
imm[11:0]			rs1	000	rd	1100111	JALR			
imm[12:10:5]			rs2	rs1	000	imm[4:1:11]	1100011	BEQ		
imm[12:10:5]			rs2	rs1	001	imm[4:1:11]	1100011	BNE		
imm[12:10:5]			rs2	rs1	100	imm[4:1:11]	1100011	BLT		
imm[12:10:5]			rs2	rs1	101	imm[4:1:11]	1100011	BGE		
imm[12:10:5]			rs2	rs1	110	imm[4:1:11]	1100011	BLTU		
imm[12:10:5]			rs2	rs1	111	imm[4:1:11]	1100011	BGEU		
imm[11:0]			rs1	000	rd	0000011	LB			
imm[11:0]			rs1	001	rd	0000011	LH			
imm[11:0]			rs1	010	rd	0000011	LW			
imm[11:0]			rs1	100	rd	0000011	LBU			
imm[11:0]			rs1	101	rd	0000011	LHU			
imm[11:5]			rs2	rs1	000	imm[4:0]	0100011	SB		
imm[11:5]			rs2	rs1	001	imm[4:0]	0100011	SH		
imm[11:5]			rs2	rs1	010	imm[4:0]	0100011	SW		
imm[11:0]			rs1	000	rd	0010011	ADDI			
imm[11:0]			rs1	010	rd	0010011	SLTI			
imm[11:0]			rs1	011	rd	0010011	SLTIU			
imm[11:0]			rs1	100	rd	0010011	XORI			
imm[11:0]			rs1	110	rd	0010011	ORI			
imm[11:0]			rs1	111	rd	0010011	ANDI			
0000000			shamt	rs1	001	rd	0010011	SLLI		
0000000			shamt	rs1	101	rd	0010011	SRLI		
0100000			shamt	rs1	101	rd	0010011	SRAI		
0000000			rs2	rs1	000	rd	0110011	ADD		
0100000			rs2	rs1	000	rd	0110011	SUB		
0000000			rs2	rs1	001	rd	0110011	SLL		
0000000			rs2	rs1	010	rd	0110011	SLT		
0000000			rs2	rs1	011	rd	0110011	SLTU		
0000000			rs2	rs1	100	rd	0110011	XOR		
0000000			rs2	rs1	101	rd	0110011	SRL		
0100000			rs2	rs1	101	rd	0110011	SRA		
0000000			rs2	rs1	110	rd	0110011	OR		
0000000			rs2	rs1	111	rd	0110011	AND		
fm		pred	succ	rs1	000	rd	0001111	FENCE		
000000000000				00000	000	00000	1110011	ECALL		
000000000001				00000	000	00000	1110011	EBREAK		

31	25	24	20	19	15	14	12	11	7	6	0
imm[11:5]		imm[4:0]		rs1	funct3		rd		opcode		
7		5		5	3		5		7		
0000000		shamt[4:0]		src	SLLI		dest		OP-IMM		
0000000		shamt[4:0]		src	SRLI		dest		OP-IMM		
0100000		shamt[4:0]		src	SRAI		dest		OP-IMM		

I-type specialization

31	27	26	25	24	20	19	15	14	12	11	7	6	0
funct7		imm[11:4]		rs2	rs1	funct3		rd		opcode			
imm[11:5]		rs2		rs1	funct3		imm[4:0]		opcode				
imm[12:10:5]		rs2		rs1	funct3		imm[4:1:11]		opcode				
				imm[31:12]				rd		opcode			
				imm[20:10:1 11 19:12]				rd		opcode			

R-type

KOREA UNIVERSITY		KOREA COMM. UNIV.	
------------------	--	-------------------	--

그림 8. RV32I Base Instruction Set

- `riscv_HPC.v`의 변경된 코드는 다음과 같다.

```

/*
Project 3 riscv_HPC.v 변경된 코드 (Line 72 ~ 98)
*/

always @ (posedge clk or posedge rst_i)
begin
    if (rst_i)
    begin
        HPC_req_Rtype <= 32'h0000_0000;
        HPC_req_Itype <= 32'h0000_0000;
        HPC_req_Stype <= 32'h0000_0000;
        HPC_req_Btype <= 32'h0000_0000;
        HPC_req_Utype <= 32'h0000_0000;
        HPC_req_Jtype <= 32'h0000_0000;
    end
    else if (req_inst_valid)
    begin
        if (req_inst_opcode[6:0] == 7'b0110011)
            HPC_req_Rtype <= HPC_req_Rtype + 32'd1;
        else if (req_inst_opcode[6:0] == 7'b1100111 || req_inst_opcode[6:0] == 7'b0000011 || req_inst_opcode[6:0] == 7'b0010011)
            HPC_req_Itype <= HPC_req_Itype + 32'd1;
        else if (req_inst_opcode[6:0] == 7'b0100011)
            HPC_req_Stype <= HPC_req_Stype + 32'd1;
        else if (req_inst_opcode[6:0] == 7'b1100011)
            HPC_req_Btype <= HPC_req_Btype + 32'd1;
        else if (req_inst_opcode[6:0] == 7'b0110111 || req_inst_opcode[6:0] == 7'b0010111)
            HPC_req_Utype <= HPC_req_Utype + 32'd1;
        else if (req_inst_opcode[6:0] == 7'b1101111)
            HPC_req_Jtype <= HPC_req_Jtype + 32'd1;
    end
end
end

```

- HPC 기능 검증 (`tb_HPC_verification.v`)

```
VSIM 3> run
# File Read Done!
# Request enqueue start!
# Reset disable... Simulation Start !!!
# Request enqueue end!
# Core reset
# Check type: R-type
# Pass
# Check type: I-type
# Pass
# Check type: S-type
# Pass
# Check type: B-type
# Pass
# Check type: U-type
# Pass
# Check type: J-type
# Pass
```

- `tb_problem2_2.v`를 시뮬레이션한 결과(problem2_2.png)는 다음과 같다.

	HPC_req_Rtype	HPC_req_Utype	HPC_req_Stype	HPC_req_Btype	HPC_req_Itype
0	0	0	0	0	0
34	0	0	0	0	0
15	0	0	0	0	0
3	0	0	0	0	0
1	0	0	0	0	0
5	0	0	0	0	0