



Project 4 Overview

KECE456 Code and System Optimization (Fall 2023)

T.A Byung Ho Choi

School of Electrical and Computer Engineering

Korea University, Seoul

Contents

- **Tool Guide (ModelSim)**

- ModelSim-Intel FPGA Edition install & create project

- **Project Overview**

- LLVM (Low Level Virtual Machine)
- GCC (GNU C Compiler)
- RISC-V
 - IMAD (Integer Multiply-Add) Instruction Extension
 - Extended HW Design

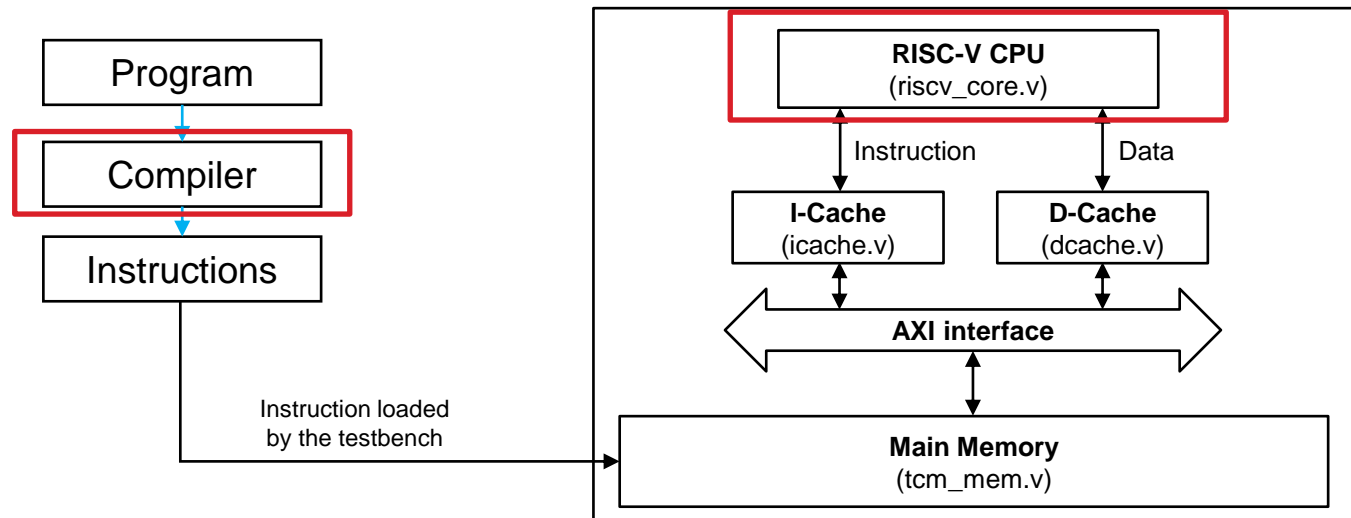
- **Appendix**

- How to Build LLVM & GCC
- CGDB basics
- RV32I ISA

Project Overview

• Abstraction

- IMAD instruction 확장
 - SW: LLVM & GCC compiler를 수정
 - HW: RISC-V core를 수정
- LLVM, GCC compiler를 이용하여, program (.c)을 RISC-V instruction으로 compile
- Testbench를 이용하여 instruction을 memory에 load한 뒤, reset signal을 trigger
- CPU가 initial PC(program counter)부터 instruction을 fetch하여 instruction 실행
- 성능 분석 및 평가



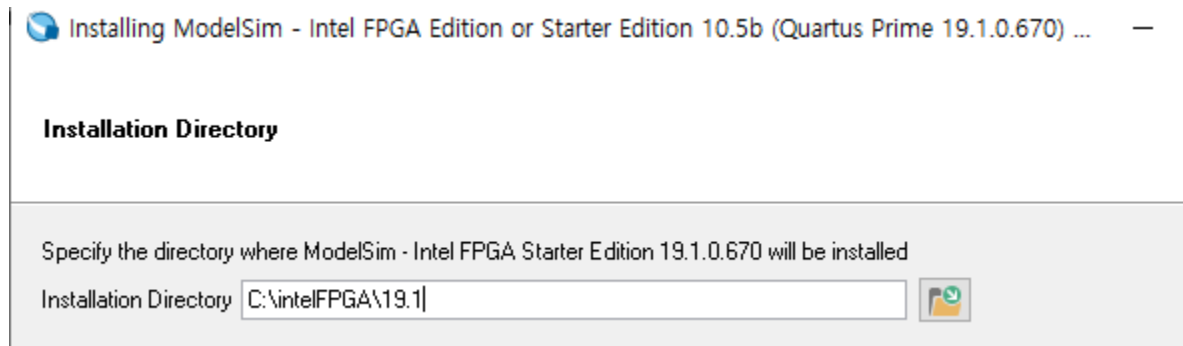
[Fig 11. RISC-V system overview]

Tool 설치

• ModelSim-Intel FPGA Edition 다운로드

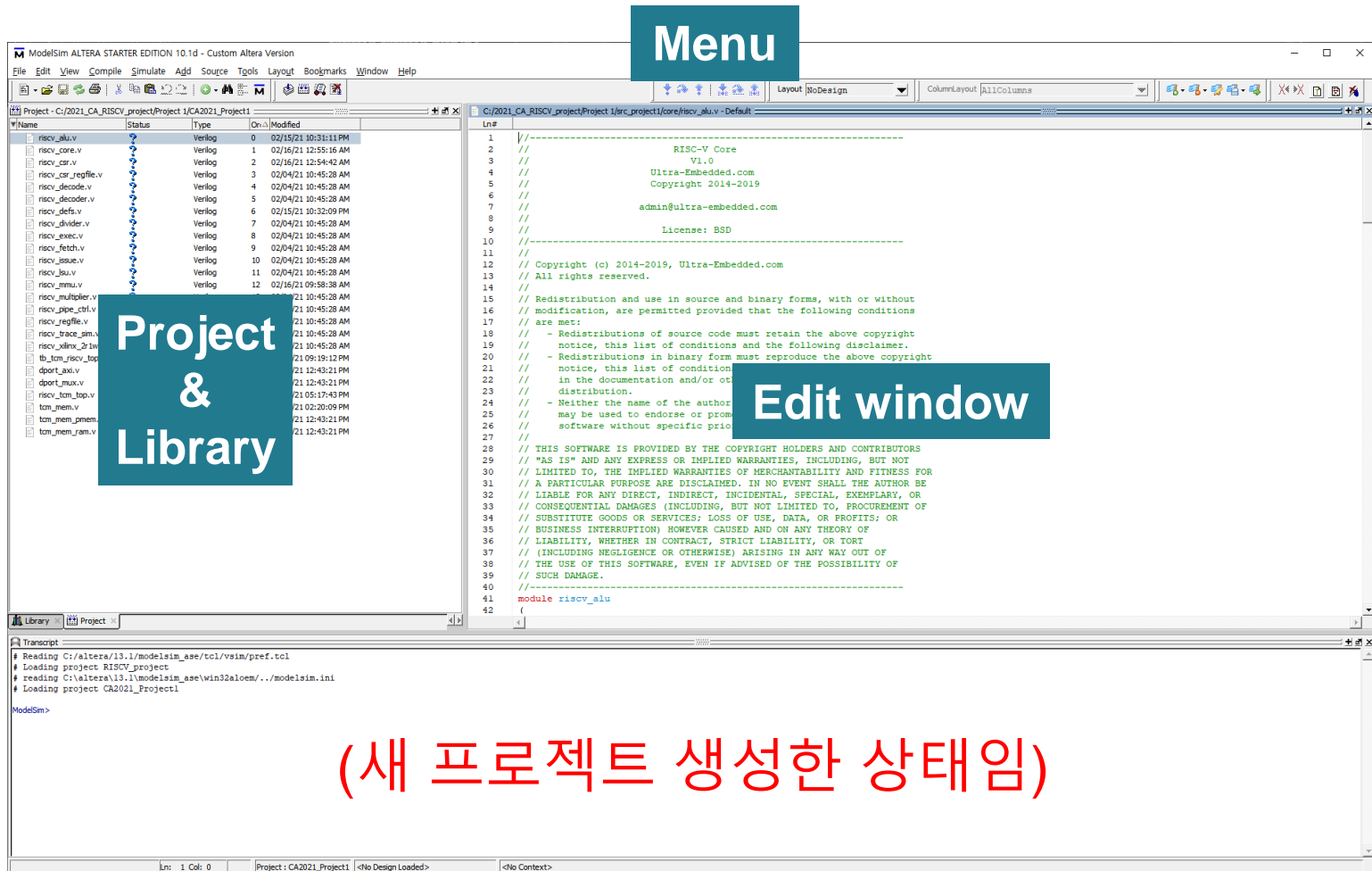
- 아래 링크에서 **ModelSim** 설치파일과 **Project source code**를 다운받는다.
 - https://drive.google.com/file/d/1-YmzkWuHG1-0vujz4eC9ISyJLHAKODiD/view?usp=drive_link
 - “Quartus-web-13.1.0.162-windows.zip” 파일을 압축해제하여 “setup.bat”을 실행하고, 아래와 같이 설치 경로를 설정한다.

*설치 경로에 한글이 있으면 안됨



[Fig 1. Installation path setting]

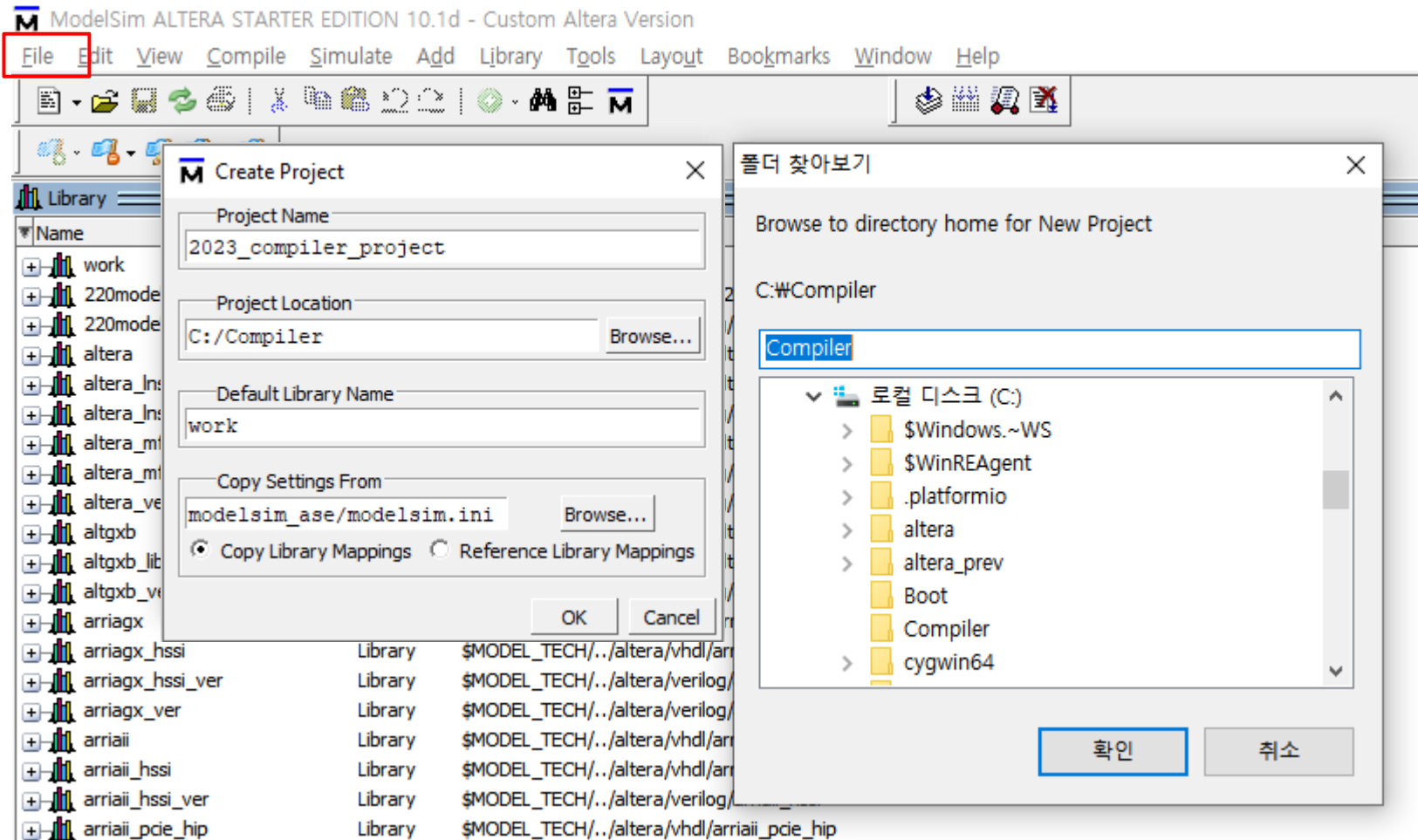
Tool Layout



[Fig 2. ModelSim Layout]

Project 생성: 새로운 프로젝트 생성

- ModelSim 소프트웨어 실행 후 Jumpstart 메뉴에서 Create a Project를 선택하거나, 메뉴 > File > New > Project... 선택하여 새 프로젝트를 만든다.

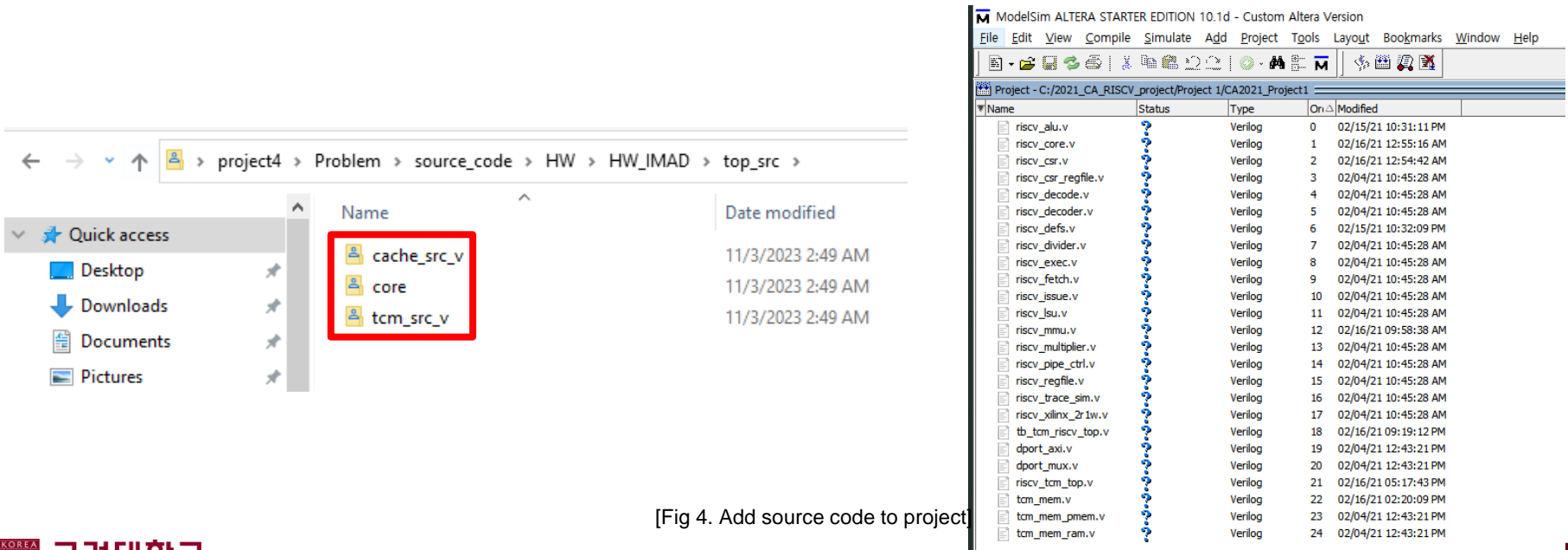


[Fig 3. Create a project]

Project 생성: 프로젝트에 기존 소스코드 추가

- Project 탭에서 우클릭하여 Add to the Project ->Add Existing File

- 제공된 모든 Verilog source code는 /source/HW/ 아래에 위치한다.
 - HW_IMAD: IMAD 연산기가 추가된 HW code
 - 여기에 IMAD instruction을 확장한다
 - HW_without_IMAD: IMAD 연산기가 추가되지 않은 baseline HW code
 - Testbench: 성능 측정을 위해 제공된 testbench
- HW_IMAD 또는 HW_without_IMAD /top_src 하위의 모든 file을 project에 추가한다.

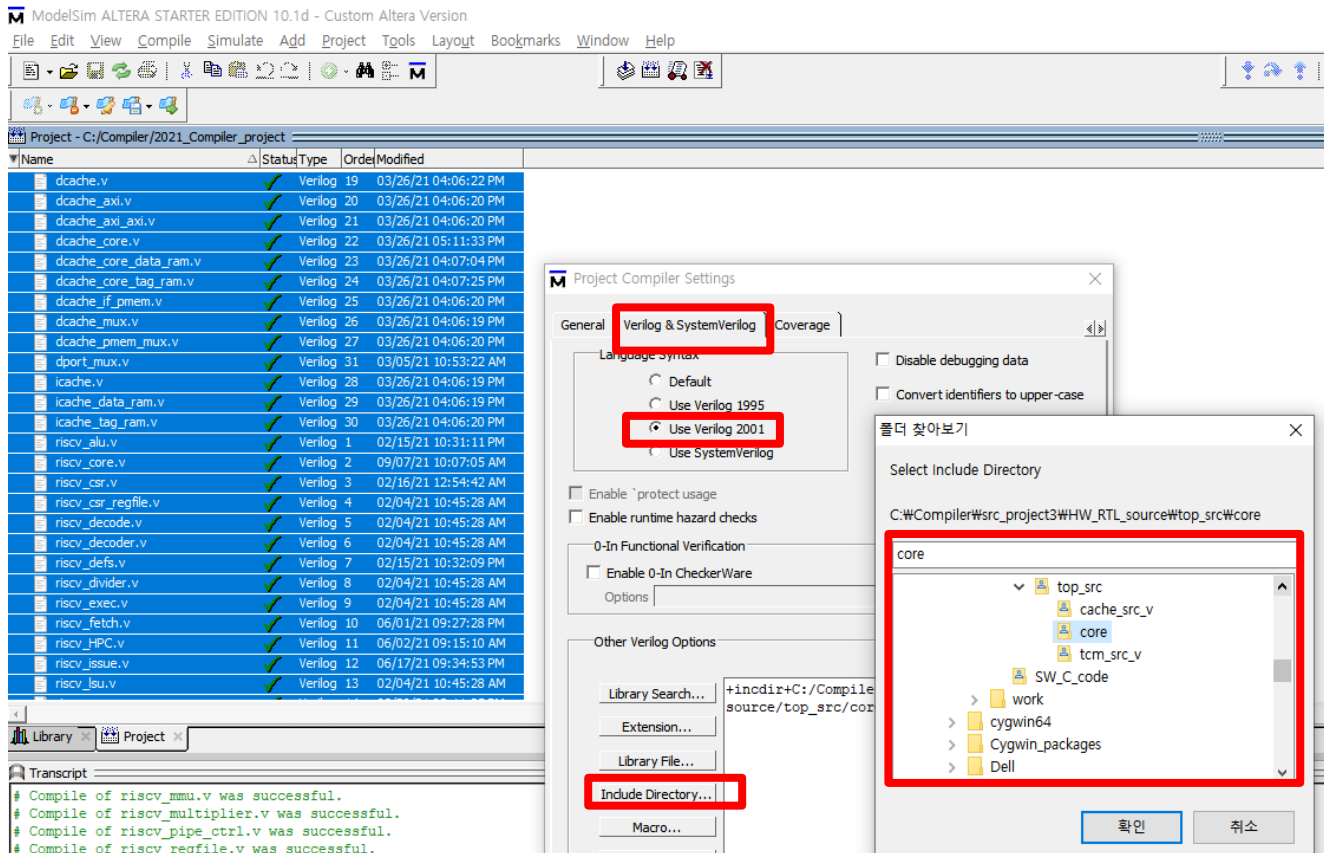


[Fig 4. Add source code to project]

Project 생성: 프로젝트에 Header File Include

- Project에 추가된 모든 Existing File을 선택한다.

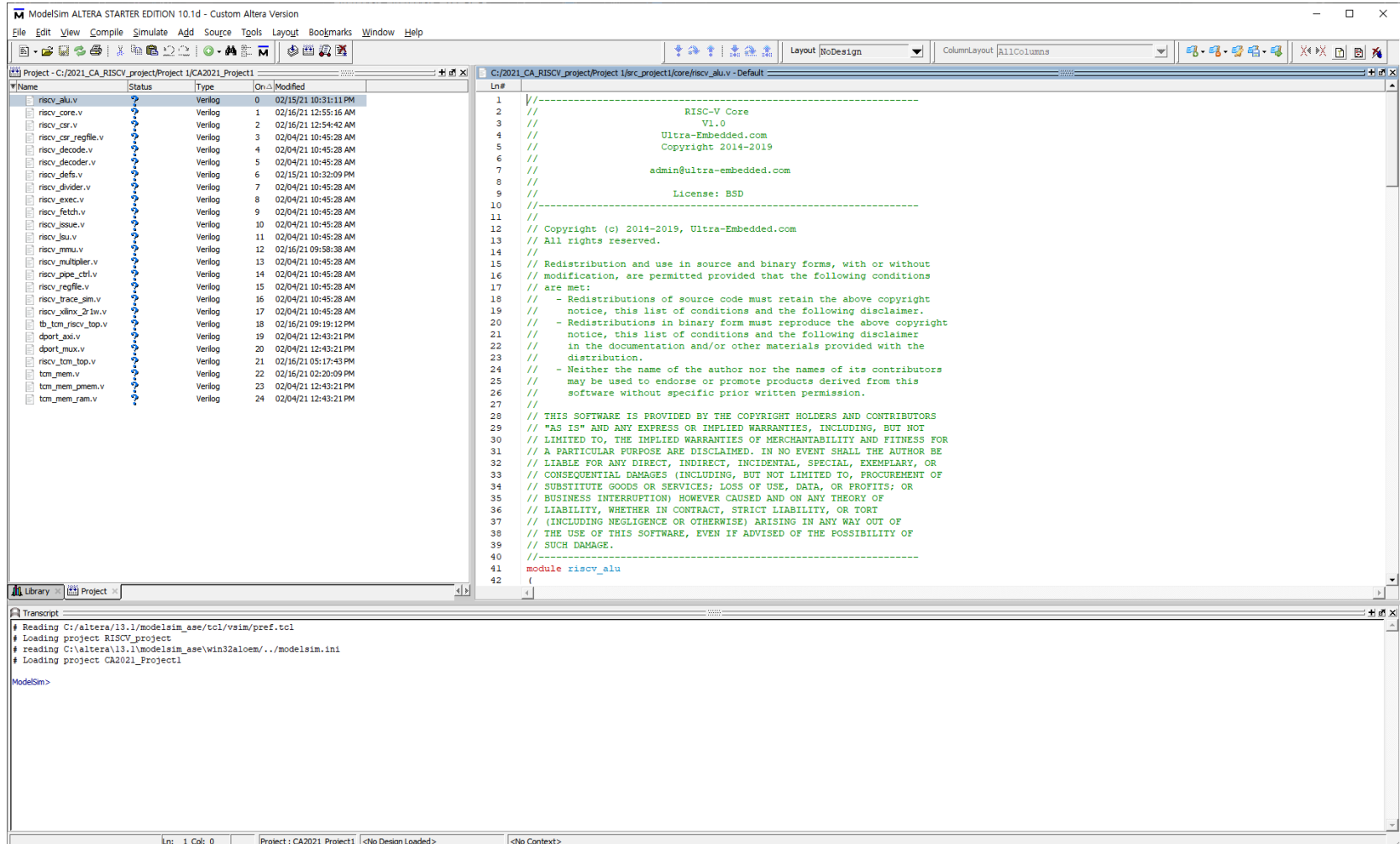
- 마우스 우클릭 → Properties → Verilog & SystemVerilog를 클릭하여 아래와 같이 선택 한다.
- /top_src/core 폴더를 선택한 후, 확인을 눌러 include한다.



[Fig 5. Compiler setting for Verilog]

Project 생성: 생성한 소스코드 파일 수정

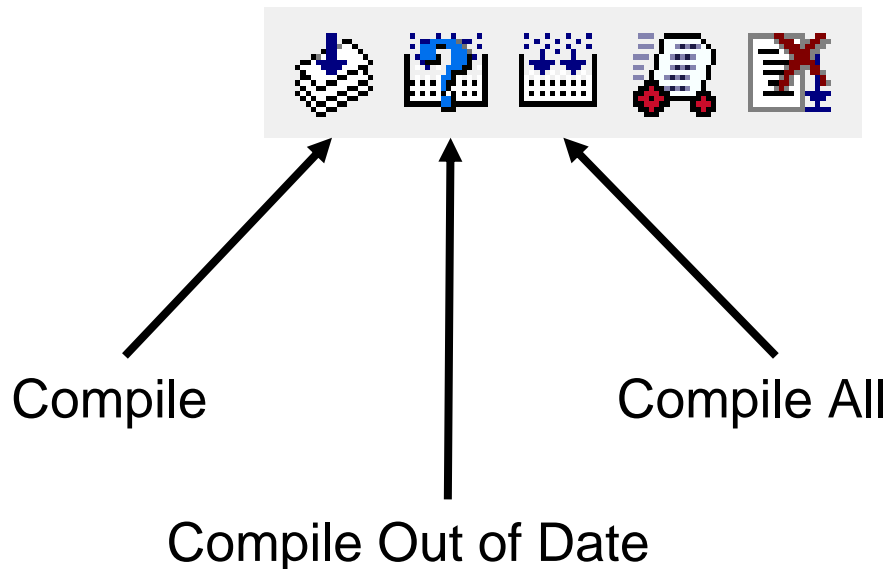
- Edit window에서 Verilog 문법에 맞게 소스코드를 작성한다.



[Fig 6. Source code editor]

Project 생성: 소스 코드 컴파일 및 에러 메시지

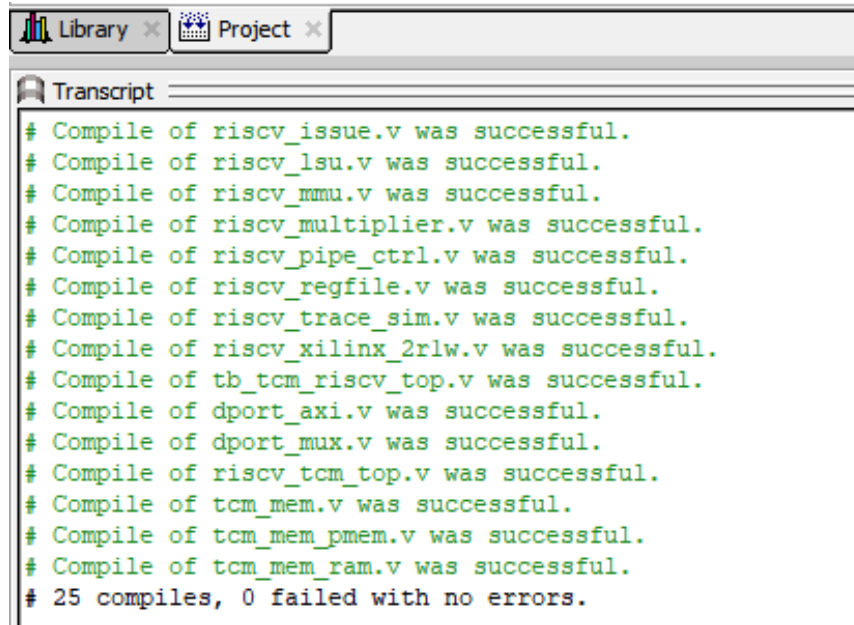
- 소스코드 컴파일을 통해 문법적 에러를 찾아낼 수 있고 시뮬레이션을 통해 논리적 에러를 찾아낼 수 있다.
- 메뉴 → **Compile** → **Compile All** 선택하면 프로젝트에 추가된 모든 소스코드를 컴파일하고, 추후에 수정된 코드만 컴파일 하고 싶을 때는 **Compile Selected** 등을 선택할 수 있다.



[Fig 7. Compile option]

Project 생성: 소스 코드 Compilation 및 Error Message

- 소스코드 상에 문제가 없다면 Compile했을 때 아래쪽 Transcript 창에 초록색으로 메시지가 뜬다.



```
# Compile of riscv_issue.v was successful.
# Compile of riscv_lsu.v was successful.
# Compile of riscv_mmu.v was successful.
# Compile of riscv_multiplier.v was successful.
# Compile of riscv_pipe_ctrl.v was successful.
# Compile of riscv_regfile.v was successful.
# Compile of riscv_trace_sim.v was successful.
# Compile of riscv_xilinx_2rlw.v was successful.
# Compile of tb_tcm_riscv_top.v was successful.
# Compile of dport_axi.v was successful.
# Compile of dport_mux.v was successful.
# Compile of riscv_tcm_top.v was successful.
# Compile of tcm_mem.v was successful.
# Compile of tcm_mem_pmem.v was successful.
# Compile of tcm_mem_ram.v was successful.
# 25 compiles, 0 failed with no errors.
```

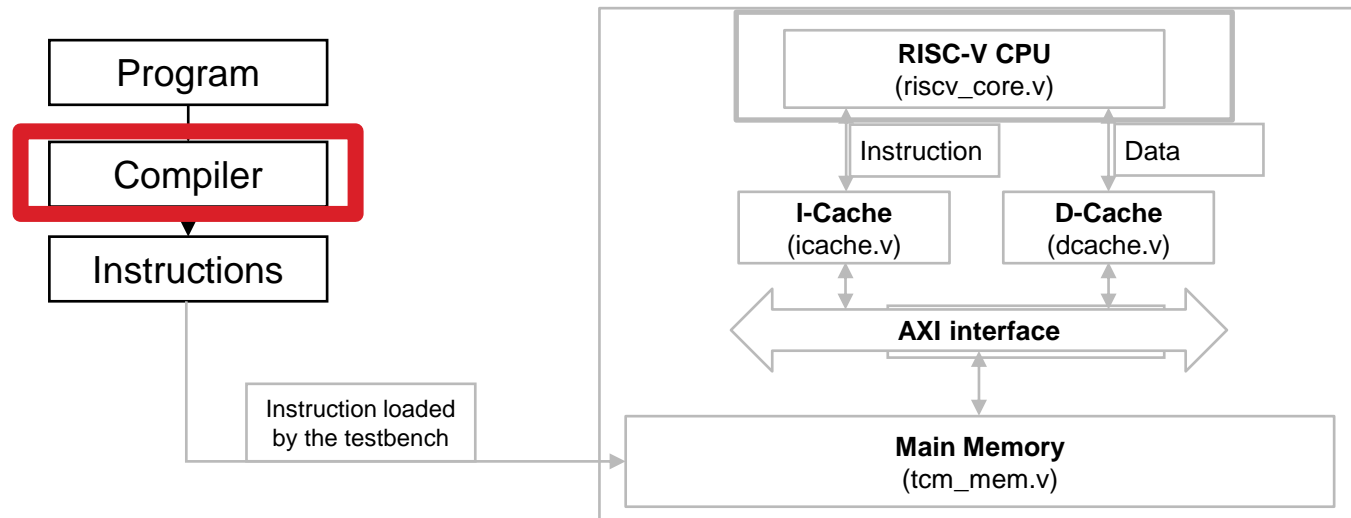
[Fig 8. Successfully compiled print]

- 소스코드 상에 에러가 있다면 해당 파일에 대해 빨간색으로 에러 메시지가 뜬다.

Project Overview

• Abstraction

- IMAD instruction 확장
 - SW: LLVM & GCC compiler를 수정
 - HW: RISC-V core를 수정
- LLVM, GCC compiler를 이용하여, program (.c)을 RISC-V instruction으로 compile
- Testbench를 이용하여 instruction을 memory에 load한 뒤, reset signal을 trigger
- CPU가 initial PC(program counter)부터 instruction을 fetch하여 instruction 실행
- 성능 분석 및 평가



[Fig 11. RISC-V system overview]

Compiler Platform Overview

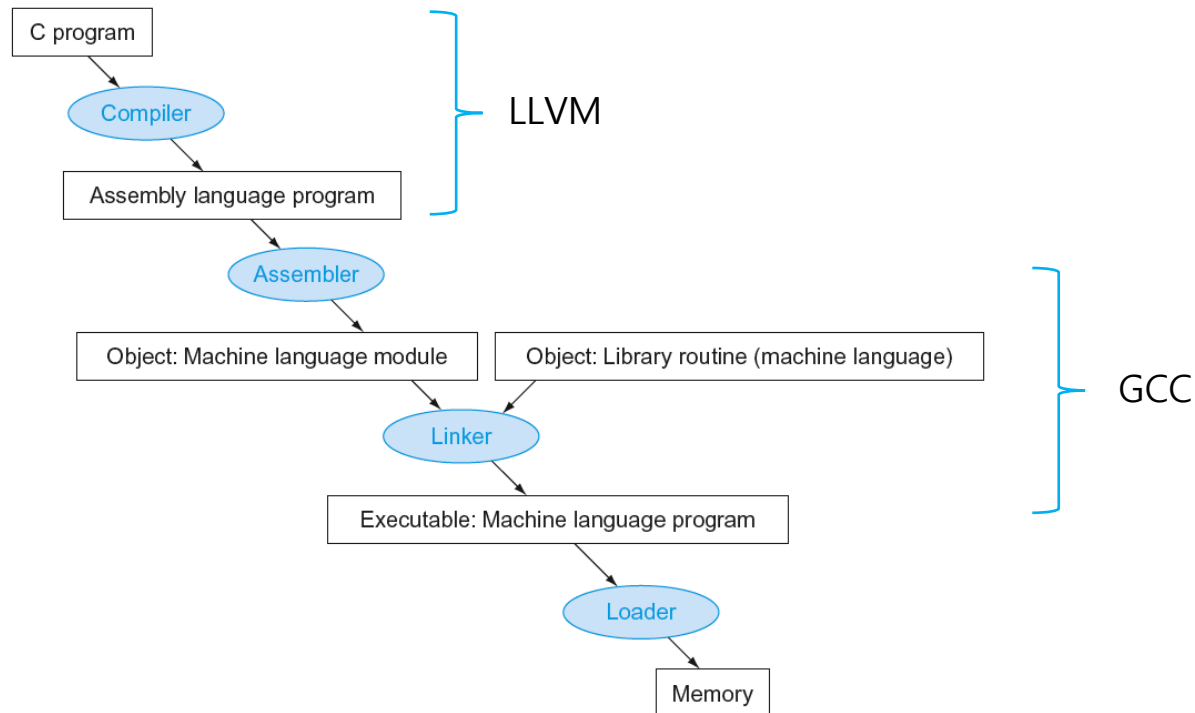
- Documents

- <https://llvm.org/docs/index.html>

- Github

- <https://github.com/llvm/llvm-project>

- <https://github.com/riscv-collab/riscv-gnu-toolchain.git>



[Fig 1. Compile sequence]

LLVM Overview

- **Front-end**

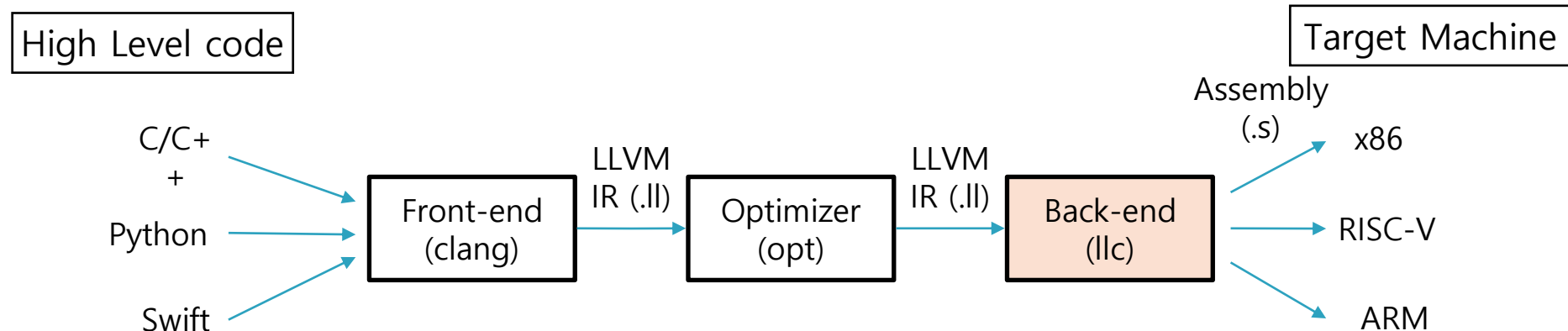
- 다양한 programming language로 작성된 program을 LLVM IR로 translation.

- **Optimizer**

- LLVM IR을 이용하여 다양한 optimization adoption.

- **Back-end**

- Target machine을 위한 assembly instruction file (.s) generation.



[Fig 2. LLVM compiler overview]

LLVM Back-end

- Document

- <https://llvm.org/docs/CodeGenerator.html#instruction-selection>

- Initial DAG Builder

- LLVM IR code를 DAG IR로 translation.

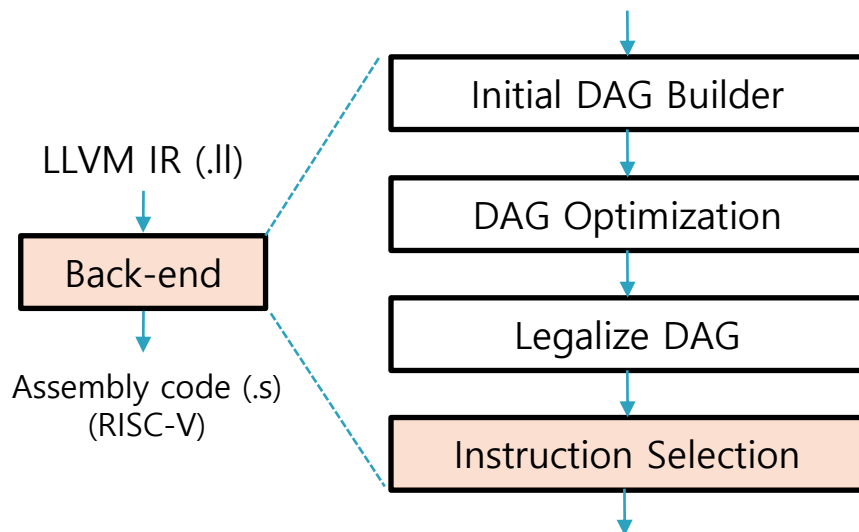
- DAG Optimization

- DAG redundancies를 제거하고, instruction selection을 위한 efficient DAG generation.

- Legalize DAG

- Target back-end architecture에서 support 하는 operator로 구성되어 있는지 checking.

- Instruction Selection



[Fig 3. LLVM back-end flow]

```
main() => compileModule() => PM.run(*M); //~/llvm-project/llvm/tools/llc/llc.cpp
=> ~~~~
=> llvm::SelectionDAGISel::runOnMachineFunction
   //~/llvm/lib/CodeGen/SelectionDAG/SelectionDAGISel.cpp:509
=> llvm::SelectionDAGISel::SelectAllBasicBlocks
   //~/llvm/lib/CodeGen/SelectionDAG/SelectionDAGISel.cpp:1372
=> llvm::SelectionDAGISel::CodeGenAndEmitDAG
   //~/llvm-project/llvm/lib/CodeGen
   /SelectionDAG/SelectionDAGISel.cpp:975
=> llvm::SelectionDAGISel::DoInstructionSelection
   //~/llvm/lib/CodeGen/SelectionDAG/SelectionDAGISel.cpp:1099
===== machine indep vs.machine dep interface =====
=> Select() //~/llvm/lib/Target/RISCV/RISCVISelDAGToDAG.cpp:1420
=> llvm::RISCVToDAGISel::SelectCode
   //~/build/lib/Target/RISCV/RISCVGenDAGISel.inc:660766
=> llvm::SelectionDAGISel::SelectCodeCommon
   //~/llvm/lib/CodeGen/SelectionDAG/SelectionDAGISel.cpp:2840
```

[Fig 4. LLVM instruction selection kernel call graph]

LLVM RISC-V ISA Extension

• Target Description (.td) File

- LLVM은 target machine의 instruction을 target description file에 정의한다.
- LLVM에 RISC-V instruction을 추가하기 위해서 다음의 두가지 file에 대한 수정이 필요하다.
 - ISDOpcodes.h
 - RISCVInstrInfo.td
- 아래의 예시를 참고하여 IMAD (Integer Multiply Add) instruction을 추가한다.

Example for ISDOpcodes.h

```
// ~/llvm/include/llvm/CodeGen
/ISDOpcodes.h

namespace llvm {
namespace ISD {
enum NodeType {
    /// Simple integer binary arithmetic
    /// operators.
    ADD,
    SUB,
    MUL,
    SDIV,
    UDIV,
    SREM,
    UREM,
    ~~~~
}}}
```

Instruction의
operator type을 정의
하는 namespace

[Fig 4. Operator type namespace]

Example for RISCVInstrInfo.td: ADD and FMADD

```
// ~/llvm/lib/Target/RISCV/RISCVInstrInfo.td
let hasSideEffects = 0, mayLoad = 0, mayStore = 0 in
class ALU_rr<bits<7> funct7, bits<3> funct3, string opcodestr>
: RVInstr<funct7, funct3, OPC_OP, (outs GPR:$rd), (ins GPR:$rs1, GPR:$rs2),
opcodestr, "$rd, $rs1, $rs2">;

def ADD : ALU_rr<0b0000000, 0b000, "add">, Sched<[WriteIALU, ReadIALU, ReadIALU]>;

// ~/llvm/lib/Target/RISCV/RISCVInstrInfoF.td
// fmaddd: rs1 * rs2 + rs3
def : Pat<(fma FPR32:$rs1, FPR32:$rs2, FPR32:$rs3), (FMADD_S $rs1, $rs2, $rs3, 0b111)>;
```

Instruction field
description

Destination/source
operand

Floating point multiply-add
instruction pattern

Pattern matching
selector

[Fig 5. RISC-V instruction definition]

LLVM IMAD Instruction Generation

• Test with LLVM IR File

- 제공된 "imad.ll"은 [Fig 6]과 같다.
- 이를 default LLVM back-end compiler 로 compile하면, [Fig 7]과 같이 " mul " instruction과 " add " instruction이 생성된다.
- 반면, IMAD instruction을 추가한 LLVM back-end compiler 로 compile하면, [Fig 8]과 같이 3-source operand와 1-destination operand의 "imad" instruction이 생성된다.
- 이때, 확장된 LLVM compiler를 실행하는 command는 다음과 같다.
 - \$ /home/compiler/work/llvm-project/build/bin/llc -mattr=+m imad.ll

```
define i32 @mul(i32 %a, i32 %b) nounwind {  
    %1 = mul i32 %a, %b  
    %2 = add i32 %1, %b  
    ret i32 %2  
}
```

[Fig 6. "imad.ll"]

```
.type mul,@function  
mul:                                     # @mul  
# %bb.0:  
mul a0, a0, a1  
add a0, a0, a1  
ret  
.Lfunc_end0:  
.size mul, .Lfunc_end0-mul  
# -- End function  
.section ".note.GNU-stack","",@progbits
```

[Fig 7. "imad.s" with default "llc"]

```
.type mul,@function  
mul:                                     # @mul  
# %bb.0:  
imad a0, a0, a1, a1  
ret  
.Lfunc_end0:  
.size mul, .Lfunc_end0-mul  
# -- End function  
.section ".note.GNU-stack","",@progbits
```

[Fig 8. "imad.s" with IMAD instruction extended "llc"]

GCC Assembler RISC-V ISA Extension (1/2)

• Document

- https://www.cse.iitb.ac.in/grc/intdocs/gcc-implementation-details.html#toc_GCC-Source-Organization

• IMAD Instruction Addition

- GCC Assembler도 LLVM과 유사하게 instruction의 format을 아래의 file에 정의한다.
 - `riscv-opc.c`
 - `riscv-opc.h`
 - `riscv-dis.c`
- 아래의 예시를 참고하여 IMAD instruction을 확장한다.

```
// ...\\riscv-gnu-toolchain\\riscv-  
binutils\\include\\opcode\\riscv-opc.h  
// ADD instruction mask & match encoding bit  
#define MATCH_ADD 0x33  
#define MASK_ADD 0xfe00707f
```

[Fig 9. RISC-V instruction encoding bit]

```
// ...\\riscv-gnu-toolchain\\riscv-binutils\\opcodes\\riscv-  
opc.c  
// ADD instruction format definition  
const struct riscv_opcode riscv_opcodes[] =  
{  
  { "add", 0, INSN_CLASS_I, "d,s,t", MATCH_ADD, MASK_ADD,  
    match_opcode, 0 },  
  ~~~~  
}
```

[Fig 10. RISC-V instruction format definition]

```
// ...\\riscv-gnu-toolchain\\riscv-binutils\\opcodes\\riscv-dis.c  
/* Print insn arguments for 32/64-bit code. */  
static void print_insn_args (const char *d, insn_t l, bfd_vma  
pc, disassemble_info *info)  
{  
  ~~~  
  struct riscv_private_data *pd = info->private_data;  
  int rs1 = (l >> OP_SH_RS1) & OP_MASK_RS1;  
  int rd = (l >> OP_SH_RD) & OP_MASK_RD;  
  
  for (; *d != '\\0'; d++) {  
    switch (*d) {  
      case 't':  
        print (info->stream, "%s",  
          riscv_gpr_names[EXTRACT_OPERAND (RS2, 1)]);  
        break;  
    }  
  }  
}
```

[Fig 11. GCC disassembly print function]

GCC Assembler RISC-V ISA Extension (2/2)

• IMAD Instruction Compile & Dump

- 제공된 imad.ll을 IMAD instruction이 추가된 llc로 compile하면, [Fig 13]과 같은 assembly code (imad.s)가 생성된다.
- IMAD instruction이 추가된 GCC assembler로 위의 assembly code (imad.s)를 compile 후, object dump 하면 [Fig 15]와 같은 dump file이 생성된다.

```
.type mul,@function
mul:                                # @mul
# %bb.0:
mul a0, a0, a1
add a0, a0, a1
ret
.Lfunc_end0:
.size mul, .Lfunc_end0-mul
# -- End function
.section ".note.GNU-stack","",@progbits
```

[Fig 12. "imad.s" with default "llc"]

```
.type mul,@function
mul:                                # @mul
# %bb.0:
imad a0, a0, a1, a1
ret
.Lfunc_end0:
.size mul, .Lfunc_end0-mul
# -- End function
.section ".note.GNU-stack","",@progbits
```

[Fig 13. "imad.s" with IMAD instruction extended "llc"]

Virtual address
and offset

RISC-V instruction

operator & operand

```
00000000 <mul>:
0:      02b50533
4:      00b50533
8:      00008067

mul      a0,a0,a1
add      a0,a0,a1
ret
```

[Fig 14. "imad.dump" with default "as"]

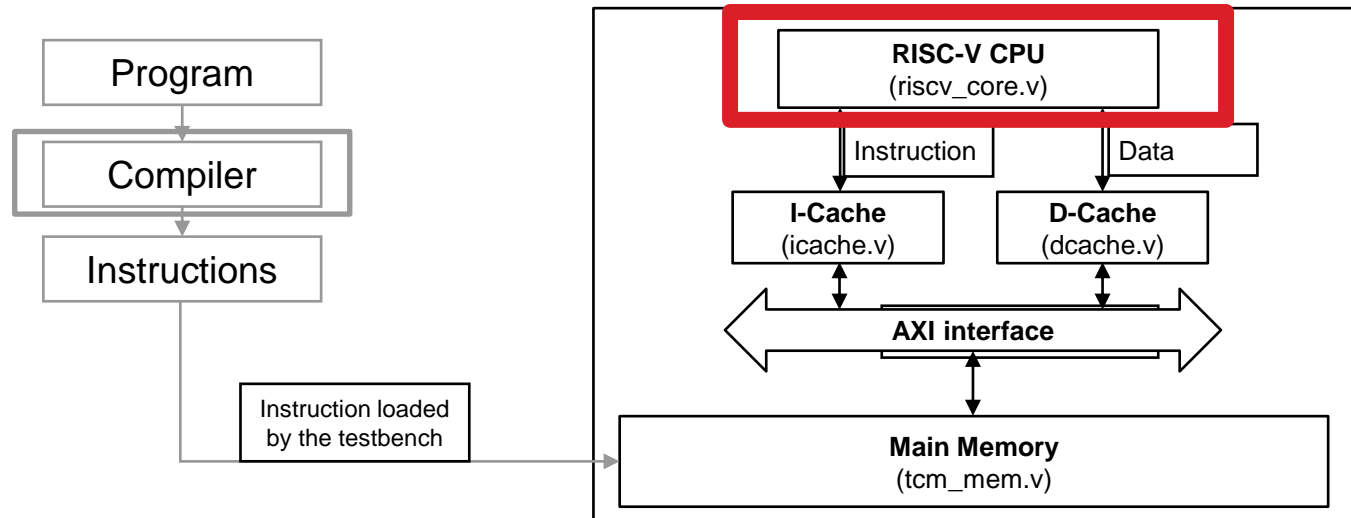
```
00000000 <mul>:
0:      5cb50533      imad      a0,a0,a1,a1
4:      00008067      ret
```

[Fig 15. "imad.dump" with IMAD instruction extended "as"]

Project Overview

• Abstraction

- IMAD instruction 확장
 - SW: LLVM & GCC compiler를 수정
 - HW: RISC-V core를 수정
- LLVM, GCC compiler를 이용하여, program (.c)을 RISC-V instruction으로 compile
- Testbench를 이용하여 instruction을 memory에 load한 뒤, reset signal을 trigger
- CPU가 initial PC(program counter)부터 instruction을 fetch하여 instruction 실행
- 성능 분석 및 평가



[Fig 11. RISC-V system overview]

RISC-V Open-Source Project

- Verilog로 작성된 32-bit RISC-V core와 RV32IM instruction set simulator를 사용한다

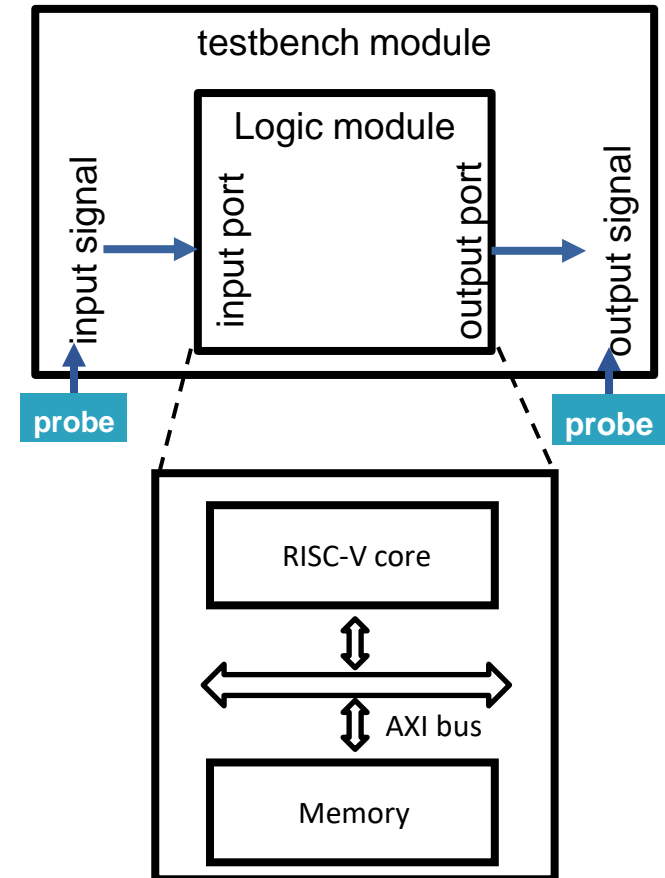
- Github: <http://github.com/ultraembedded/riscv>

- **Uploaded source project folder tree**

- /src
 - /core
 - All Verilog code for RISC-V core
 - /tb
 - All Verilog code for testbench and input instruction sequence
 - /tcm_top
 - All Verilog code for TCM (Main memory) and top module

- **AXI interface documents**

- https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf

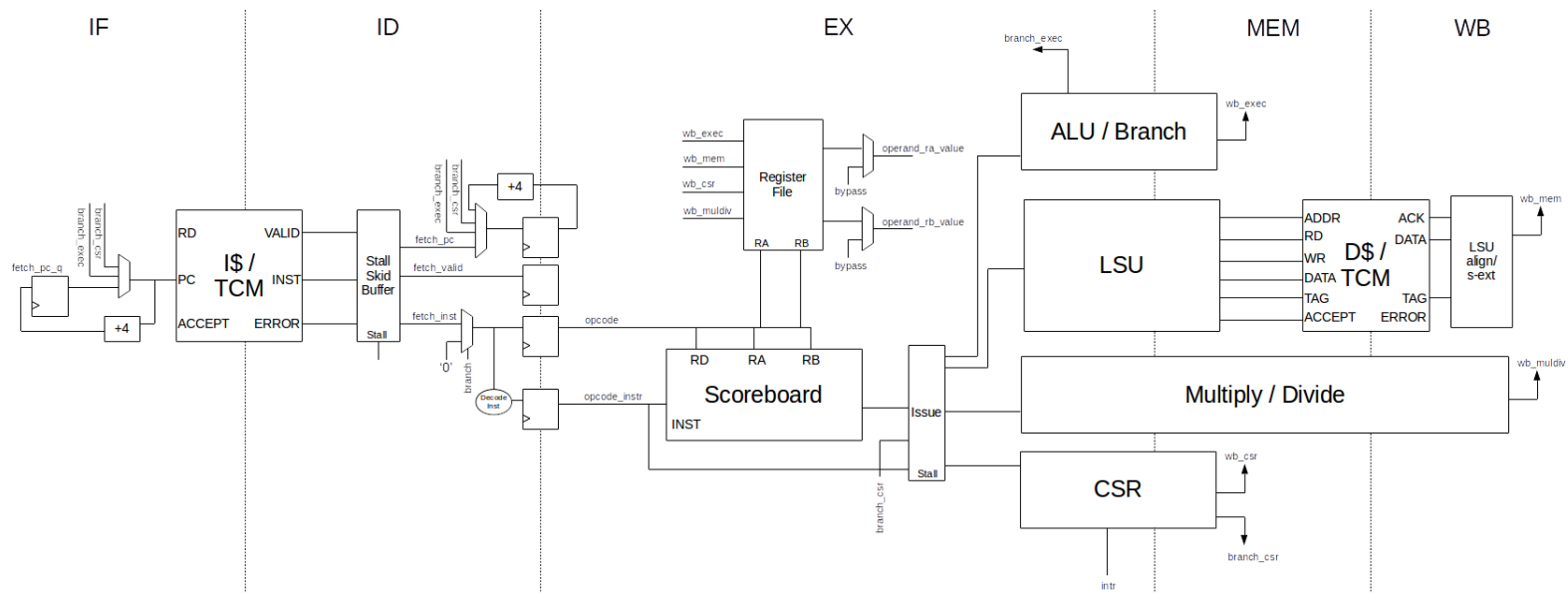


[Fig 10. Testbench diagram]

RISC-V Core Architecture

• Main Features

- 5-stage in-order
- 32-bit RISC-V ISA CPU core
- Support RISC-V integer (I), multiplication and division (M), and CSR instructions (Z) extensions (RV32IMZicsr)



[Fig 12. RISC-V core architecture]

RISC-V Instruction

• Instruction type, meaning & classification

- R – Arithmetic data processing
 - ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND
- I – Immediate value data processing, load, long jump
 - ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SLRI, SRAI
 - LB, LH, LW, LBU, LHU
 - JALR
- S – Store memory access
 - SB, SH, SW
- B – Conditional jump to target PC
 - BEQ, BNE, BLT, BGE, BLTU, BGEU
- U – Upper intermediate related instruction
 - LUI, AUIPC
- J – Unconditional jump to target PC
 - JAL
- M extension
 - MUL/DIV
- 각 Instruction의 bit field가 register, immediate value 및 opcode를 의미한다.
 - Funct7과 func3은 further operation을 위한 field를 의미한다.
- 기타 내용은 RISC-V specification document를 참고할 것.

31	27	26	25	24	20	19	15	14	12	11	7	6	0		
funct7				rs2		rs1	funct3		rd	opcode				R-type	
imm[11:0]						rs1	funct3		rd	opcode				I-type	
imm[11:5]				rs2		rs1	funct3		imm[4:0]	opcode				S-type	
imm[12:10:5]				rs2		rs1	funct3		imm[4:1 11]	opcode				B-type	
imm[31:12]									rd		opcode				U-type
imm[20 10:1 11 19:12]									rd		opcode				J-type

[Fig 13. RV32I Instruction format]

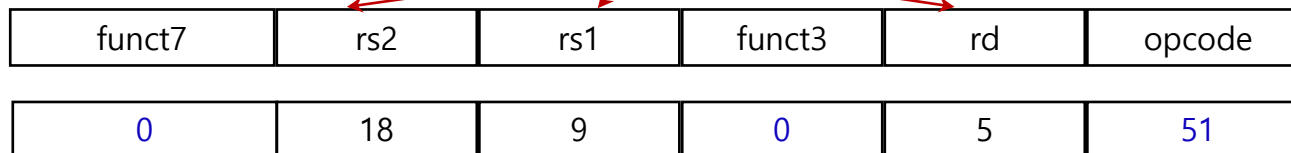
RISC-V Instruction Format

• Arithmetic Instruction Example

– Add instruction

**s1과 s2는 source operand를 t0는 destination operand를 의미함

add t0, s1, s2 # t0 = s1 + s2



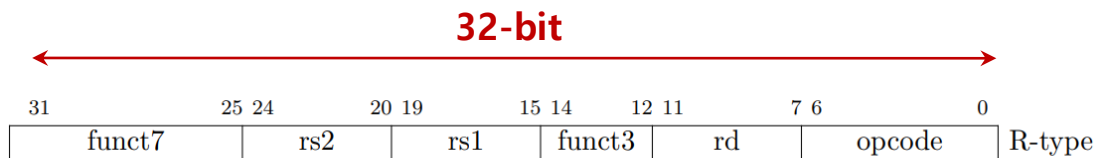
binary 0000000 10010 01001 000 00101 0110011

00000001 00100100 10000010 10110011

hexadecimal 0x0124 82B3

- Opcode (7 bit): Instruction의 operation을 specify
- funct7 (7 bit), funct3 (3 bit): Further opcode
- rs1 (5 bit): Register of the 1st source operand
- rs2 (5 bit): Register of the 2nd source operand
- rd (5 bit): Destination register

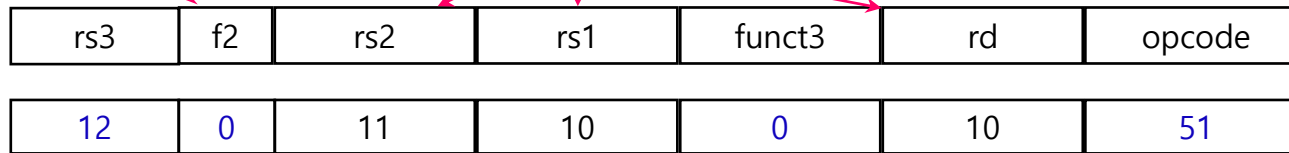
Name	Register Number
zero	x0
ra	x1
sp	x2
gp	x3
tp	x4
t0 – t2	x5–x7
s0 / fp	x8
s1	x9
a0 – a7	x10 – x17
s2 – s11	x18 – x27
t3 – t6	x28 – x31



RISC-V IMAD (Integer Multiply-Add) Instruction Extension

• IMAD Instruction Format

imad $a0, a0, a1, a2 \# a0 = a0 * a1 + a2$



binary 01100 00 01011 01010 000 01010 0110011
 01100000 10110101 00000101 00110011

hexadecimal 0x60b5 0533

- Opcode (7 bit): Instruction의 operation을 specify
- f2 (2 bit), funct3 (3 bit): Further opcode
- rs1 (5 bit): Register of the 1st source operand
- rs2 (5 bit): Register of the 2nd source operand
- rs3 (5 bit): Register of the 3rd source operand
- rd (5 bit): Destination register

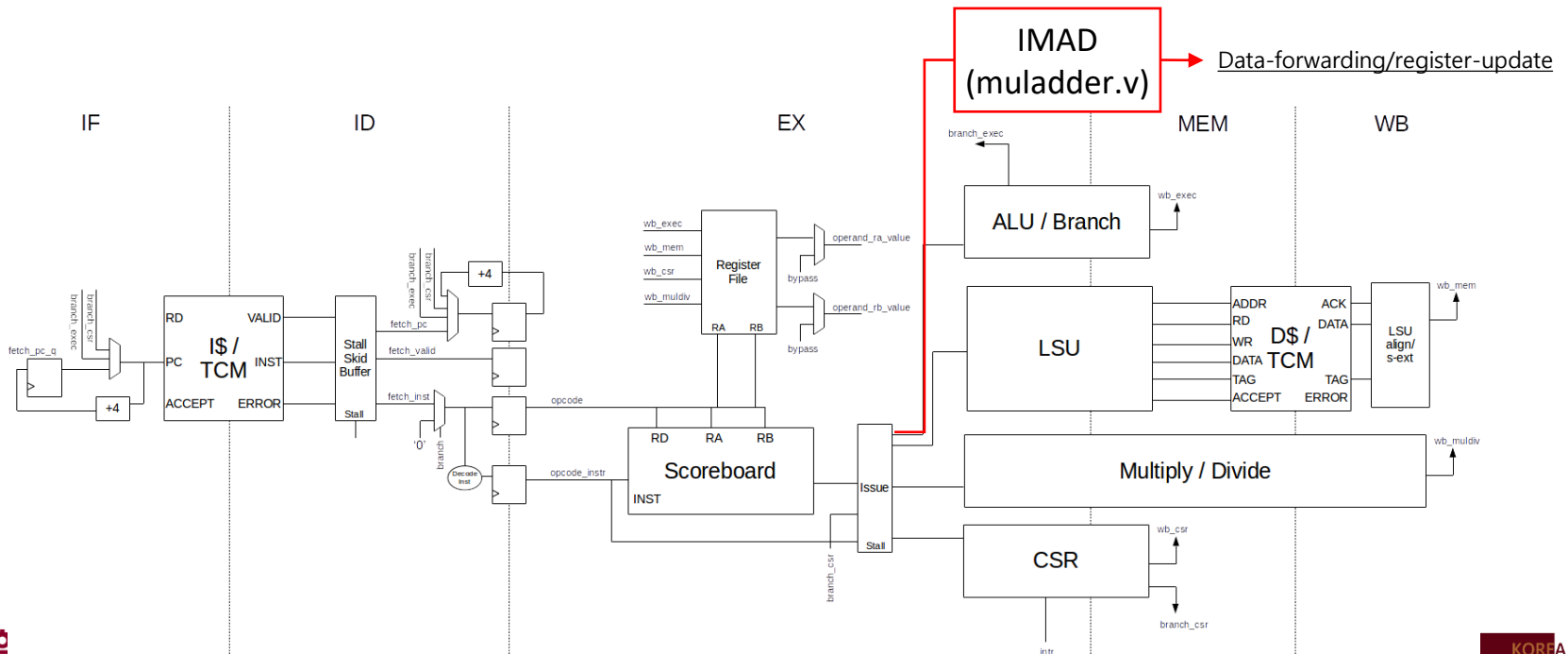
Name	Register Number
zero	x0
ra	x1
sp	x2
gp	x3
tp	x4
t0 – t2	x5-x7
s0 / fp	x8
s1	x9
a0 – a7	x10 – x17
s2 – s11	x18 – x27
t3 – t6	x28 – x31

RISC-V Core ISA Extension

• IMAD Unit

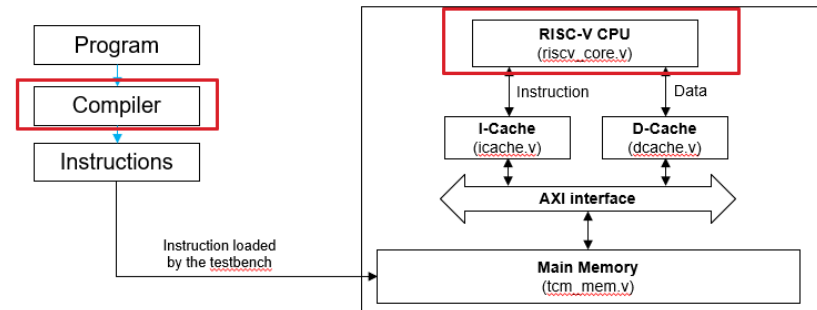
- 제공된 IMAD ("muladder.v") module은 behavior model이며 3-source operand에 대한 non-pipelined multiplication & addition 연산을 3 cycle 동안 수행한다.
 - IMAD module이 하나의 IMAD instruction 연산을 수행하는 동안 다른 IMAD instruction을 pipelining하여 수행할 수 없다.
- 오른쪽에 list up 되어있는 file을 수정하여 기존의 RISC-V core에 IMAD instruction에 대한 datapath와 control-path를 추가한다.

- riscv_decode.v
- riscv_decoder.v
- riscv_regfile.v
- riscv_def.v
- riscv_core.v
- riscv_issue.v
- riscv_pipe_ctrl.v



Simulation process

- Compiler와 RISC-V core에 imad instruction을 확장한다
 - Compiler를 수정한 후, GCC와 LLVM을 각각 build한다
 - Build command는 appendix 및 document 참고
 - RISC-V core를 수정한 후, 회로를 compile한다
- Build한 compiler를 통해, conv.c를 machine code로 compile한다
 - project4 압축파일에 첨부된 Makefile을 이용한다
- Compile된 결과물(out.hex)을, 제공된 Testbench를 통해, 수정된 RISC-V core로 실행하고, 결과를 분석한다



[Fig 11. RISC-V system overview]

Appendix. LLVM & GCC Build (Debug Mode)

- VMWare의 Terminal에 다음의 command를 입력하여 GCC와 LLVM을 build 한다.
- **GCC**
 - `cd ~/work/riscv-gnu-toolchain`
 - `./configure --prefix=/home/compiler/work/riscv-gnu-toolchain/riscv --with-arch=rv32imac --with-abi=ilp32`
 - `make`
- **LLVM**
 - `cd ~/work/llvm-project/build`
 - `~/Downloads/cmake-3.21.2/bin/cmake -G "Unix Makefiles" -DLLVM_TARGETS_TO_BUILD="RISCV" -DCMAKE_BUILD_TYPE="Debug" -DBUILD_SHARED_LIBS=True -DLLVM_USE_SPLIT_DWARF=True -DLLVM_OPTIMIZED_TABLEGEN=True -DLLVM_BUILD_TESTS=False -DDEFAULT_SYSROOT="/home/compiler/work/riscv-gnu-toolchain/riscv/bin/riscv32-unknown-elf" -DGCC_INSTALL_PREFIX="/home/compiler/work/riscv-gnu-toolchain/riscv/bin" -DLLVM_DEFAULT_TARGET_TRIPLE="riscv32-unknown-elf" -DLLVM_EXPERIMENTAL_TARGETS_TO_BUILD="RISCV" -DLLVM_ENABLE_PROJECTS=clang ../llvm`
 - `make`

Appendix. CGDB Basics

• CGDB 설치

- Linux terminal을 open한 뒤, 다음의 command를 입력하여 "cgdb" package를 설치한다.
 - `sudo apt-get install cgdb`
 - 다중 사용자 서버에서 `sudo apt-get` 사용시 주의
- Debugging하고자 하는 경로에서 다음의 argument를 입력한다.
 - `cgdb -ex -r --args /home/compiler/work/llvm-project/build/bin/llc -mattr=+m ~~~.ll`
- Command
 - Breakpoint – `"b **kernel_name**"`
 - Delete breakpoint – `"clear"` or `"d"`
 - Jump to caller – `"up"`
 - Jump to callee – `"down"`

Appendix. LLVM Instruction Scheduling Information

• Target Description Execution Cycle Example

- Target description file에 compile의 instruction scheduling 단계에서 사용되는 instruction의 execution cycle 정보를 정의할 수 있다.
 - 이는 core마다 각 instruction에 대한 execution cycle이 다른 경우, instruction scheduling optimization을 위한 것이다.
 - [Fig 16]은 RISC-V IDIV (Integer Division) instruction에 대한 default scheduling information이며, 별도의 execution cycle 정보가 정의되어 있지 않다.
 - 반면, [Fig 17]은 Rocket-chip (RISC-V core)의 IDIV instruction scheduling information으로 33 cycle이 정의되어 있다.
- IMAD instruction에 대한 execution cycle은 별도로 정의하지 않는다.

```
// M Extension ISA
// ~/llvm/lib/Target/RISCV/RISCVInstrInfoM.td

// Integer division instruction
def DIV : ALU_rr<0b0000001, 0b100, "div">,
        Sched<WriteIDiv, ReadIDiv, ReadIDiv>;
```

```
// Integer division
// ~/llvm/lib/Target/RISCV/RISCVSchedule.td

// Define scheduler resources associated with def operands.
// 32-bit or 64-bit divide and remainder
def WriteIDiv : SchedWrite;
```

[Fig 16. DIV instruction scheduling information]

```
// ~/llvm/lib/Target/RISCV/RISCVSchedRocket.td

// Integer division
// Worst case latency is used.
def : WriteRes<WriteIDiv>, [RocketUnitIDiv]> {
  let Latency = 33;
  let ResourceCycles = [33];
}
```

[Fig 17. Rocket-chip architecture DIV instruction execution cycle]

Appendix. RV32I Base Instruction Set

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI	
imm[31:12]				rd	0010111	AUIPC	
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

U-type

J-type

B-type

S-type

I-type

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

R-type

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7				rs2		rs1		funct3		rd		opcode		R-type
imm[11:0]						rs1		funct3		rd		opcode		I-type
imm[11:5]				rs2		rs1		funct3		imm[4:0]		opcode		S-type
imm[12 10:5]				rs2		rs1		funct3		imm[4:1 11]		opcode		B-type
imm[31:12]										rd		opcode		U-type
imm[20 10:1 11 19:12]										rd		opcode		J-type

Appendix. Trouble shooting

- gcc 또는 llvm을 build하는 과정에서 make: nothing to be done for 'all' 에러가 발생하는 경우
 - make clean 후에 다시 make
- gcc 또는 llvm을 build하는 과정에서 기타 에러가 발생하는 경우
 - make distclean 후에 다시 configure하고 make
- CMakeCache.txt 관련 에러 발생 시
 - 에러 메시지를 참고하여, 관련 파일을 삭제 후 다시 make

Appendix. Modification checklist

• Compiler

– LLVM

- /llvm-project/llvm/include/llvm/CodeGen/ISDOpcodes.h
- /llvm-project/llvm/lib/Target/RISCV/RISCVInstrInfo.td

– GCC

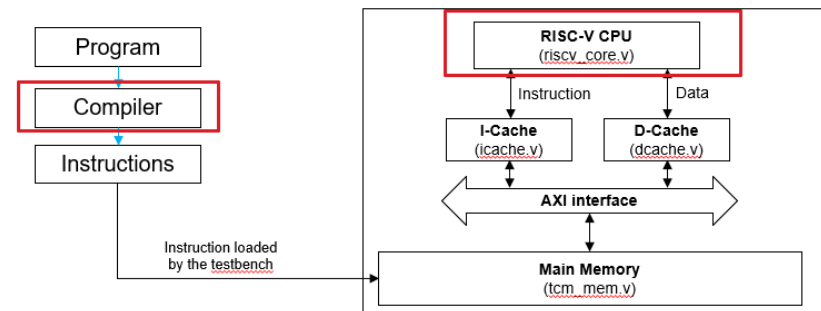
- /riscv-gnu-toolchain/riscv-binutils/include/opcode/riscv-opc.h
- /riscv-gnu-toolchain/riscv-binutils/opcodes/riscv-dis.c
- /riscv-gnu-toolchain/riscv-binutils/opcodes/riscv-opc.c

• RISC-V CPU

– /top_src/core

- riscv_decode.v, riscv_decoder.v, riscv_regfile.v, riscv_def.v, riscv_core.v, riscv_issue.v, riscv_pipe_ctrl.v

수정한 곳에 반드시 학번으로 주석을 남겨주세요!
Ex) //2022000000



[Fig 11. RISC-V system overview]