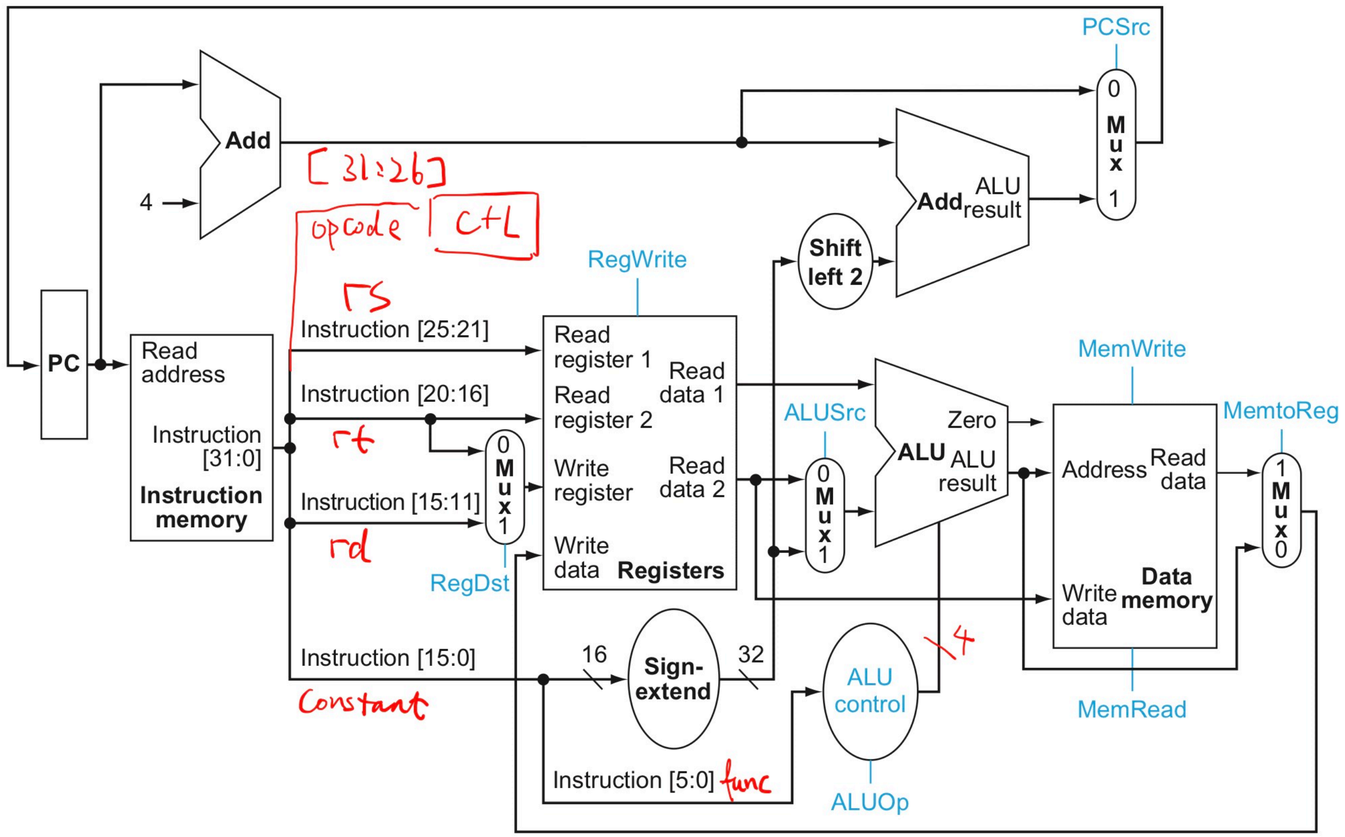
Project 3

The MIPS processor simulation

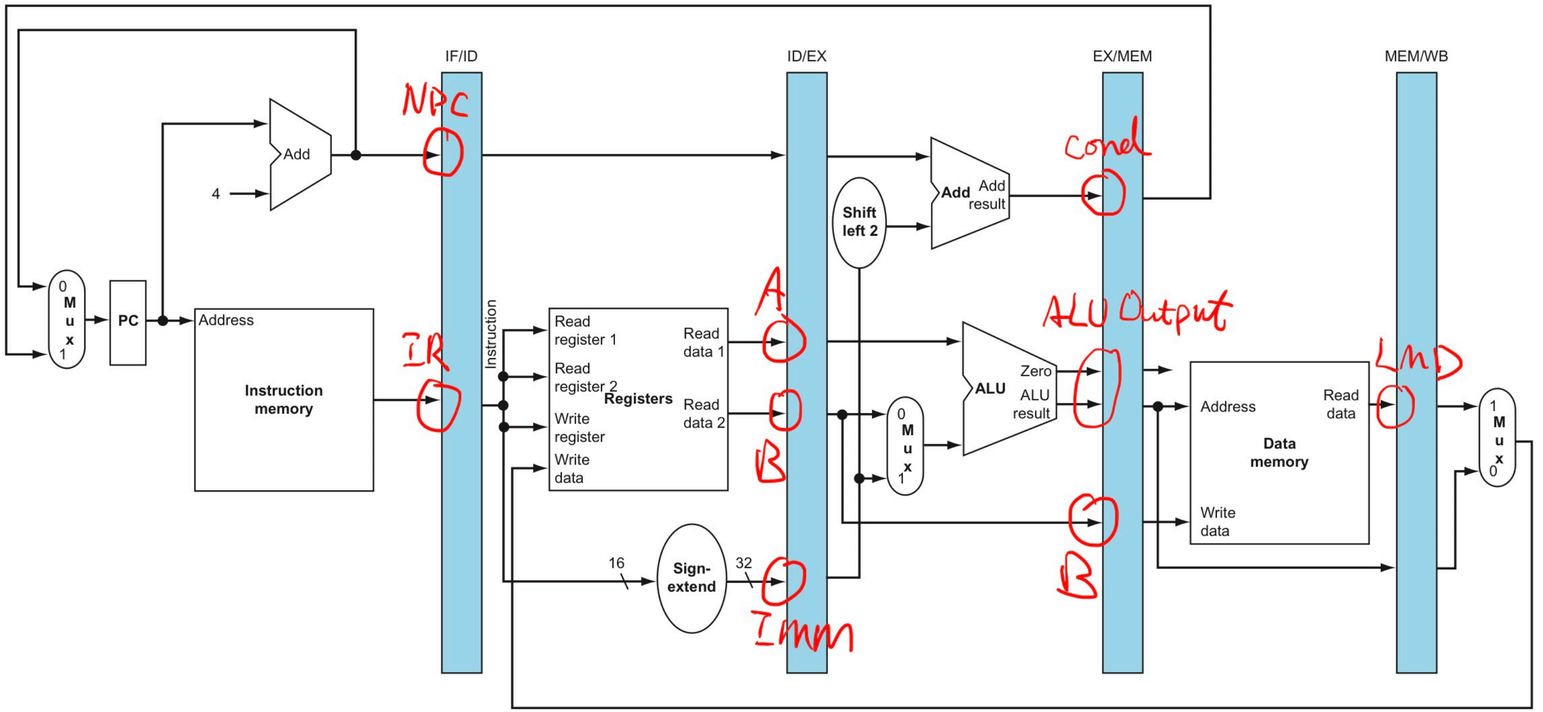
Offering

- 5 stage pipeline
- Branching
- Hazard prevention
- Special MUL instruction
- Assembler
- Disassembler

Component Reference



Pipeline Reference



IF stage

```
if (this->ctl_stall || this->pc.get() >= this->inst_length) {
    return;
}
auto npc = this->pc.get();
auto ins = this->inst_memory.read(npc);
this->stage_latches.ifId = IF_ID(npc, ins);
this->pc.next();
this->cycle_utilized();
this->IF_hit++;
```

ID stage

```
if (this→ctl_stall) return false;
auto latch = &this→stage_latches.ifId;
auto ins = latch→getIr();
// return true when nop, this will force PC to increase in IF stage and reach inst_len sooner or later
if (ins.is_nop()) return true;
auto reg_a = ins.rs();
auto reg_b = 0;
auto dat_b = 0;
auto dat_imm = SignExt().extend(ins.imme());
if (!ins.is_imm()) {
    if (ins.funt() == SLL || ins.funt() == SRL) {
        reg_a = ins.rt();
        dat_b = ins.shamt();
    } else {
        reg_b = ins.rt();
    }
} else if (ins.op() == BEQ || ins.op() == SW) {
    // rt is originally reserved for result in immediate instructions
    // in branching will be used for comparison
    reg_b = ins.rt();
}
```

```
if (reg_b ≠ 0) {
    if (this→has_data_hazard(reg_b)) return false;
    dat_b = this→registerFile.read_reg(reg_b);
}
if (reg_a ≠ 0 && this→has_data_hazard(reg_a)) return false;
auto dat_a = this→registerFile.read_reg(reg_a);
if (ins.op() == BEQ) {
    this→ctl_stall = true;
}
this→stage_latches.idEx = ID_EX(dat_a, dat_b, dat_imm, latch→getNpc(), ins);
latch→set_nop();
this→cycle_utilized();
this→ID_hit++;
return true;
```

EX stage

```
auto latch = &this→stage_latches.idEx;
auto ins = latch→getIr();
if (ins.is_nop()) return;
auto ins_op = ins.op();
auto latch_a = latch→getA();
auto latch_b = latch→getB();
auto alu_a = latch_a;
auto alu_b = 0;
// determinate ALU B by instructions
if (ins_op == LUI) {
    alu_a = latch→getImm();
    alu_b = 16;
} else if (ins.is_imm() && ins.op() ≠ BEQ) {
    alu_b = latch→getImm();
} else {
    alu_b = latch_b;
}
```

```
auto alu_op = this→alu_ctl.decode(ins);
auto alu_out = this→alu.compute(alu_a, alu_b, alu_op);
// special case for branch, we don't multiply the imm by 4 because the
auto cond = latch→getNpc() + latch→getImm() + 1;
// instruction memory line is exactly 32 bit, 4 bytes
this→stage_latches.exMem = EX_MEM(cond, alu_out, latch→getB(), ins);
latch→set_nop();
this→cycle_utilized();
this→EX_hit++;
```

MEM stage

```
auto latch = &this->stage_latches.exMem;
auto ins = latch->getIr();
if (ins.is_nop()) return;
auto alu_res = latch->getAluOut();
auto alu_res_dat = alu_res_to_dat(alu_res);
auto mem_res = 0;
auto utlized = true;
switch (ins.op()) {
    case LW:
        // Load Word, should load data into the latch
        mem_res = this->data_memory.read(alu_res_dat);
        break;
    case SW:
        // Save Word, should write data into memory
        this->data_memory.write(alu_res_dat, latch->getB());
        break;
```

```
  case BEQ:
        // Branch EQual, when ALU result == 0 which indicates equal, should set
program counter to cond in latch
        if (alu_res == 0) this->pc.set(latch->getCond());
        // unset the control stall flag, so the IF stage will be enabled in next cycle
        this->ctl_stall = false;
        break;
    default:
        utlized = false;
        break;
}
if (utlized) {
    this->cycle_utilized();
    this->MEM_hit++;
}
this->stage_latches.memWb = MEM_WB(mem_res, alu_res, ins);
latch->set_nop();
```

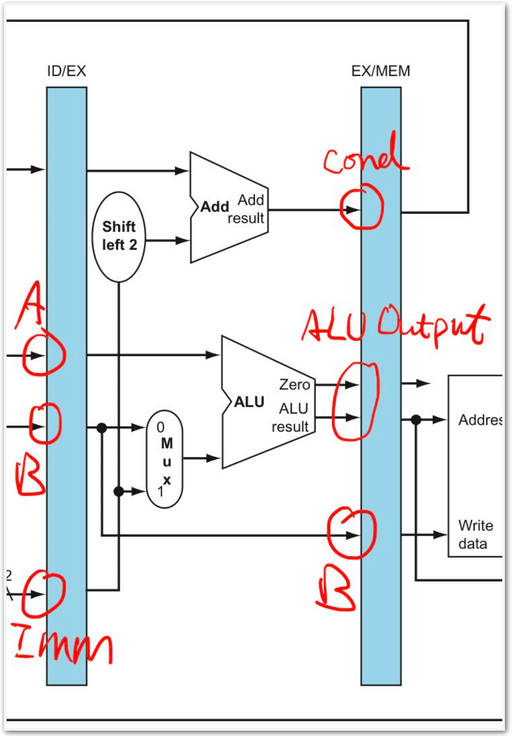
WB stage

```
auto latch = &this→stage_latches.memWb;
auto ins = latch→getIr();
if (ins.is_nop()) return {NOP_INST};
auto target = ins.dest_reg();
auto dat_write = 0;
switch (ins.op()) {
    case BEQ:
    case SW:
        // do nothing
        target = 0;
        break;
    case LW:
        dat_write = latch→getLmd();
        break;
```

```
  default:
        alu_res alu_res = latch->getAluOut();
        // extract the least significant 32 bits of the 64 bit ALU result
        reg_dat reg_1_res = alu_res_to_dat(alu_res);
        // assign to the register
        dat_write = reg_1_res;
        if (ins.op() == 0 && ins.funt() == MUL) {
            // handle the nasty MUL result from int64 to 2 int32 regs
            // extract the most significant 32 bits of the 64 bit ALU result
            reg_dat reg_2_res = (alu_res & 0xFFFFFFFF00000000) >> 32;
            this->registerFile.write_reg(target + 1, reg_2_res);
        }
        break;
}
if (target != 0) {
    this->registerFile.write_reg(target, dat_write);
    this->cycle_utilized();
    this->WB_hit++;
}
latch->set_nop();
return ins;
```

Branching

- Stall at ID stage
- ALU minus rs with rt
- Another adder computes address (cond)
- Update PC at MEM stage if ALU output is 0



Hazard Prevention

- Split data memory and instruction memory (structural hazard)
- Block following ID on BEQ in ID, restore at MEM (control hazard)
- Check destination register after ID stage (data hazard)

MUL Instruction

- A bad example on how data path should not been designed
- ALU have to be 64-bit output specially for this instruction
- Have to check 2 registers to prevent data hazard (2 target register)
- Usually multiplier use separate pipeline hardware alone with ALU
- Usually multiplier take more clock cycles (stages)
- Usually results goes into multiplier-quotient 2-word register and taken out by another set of instructions

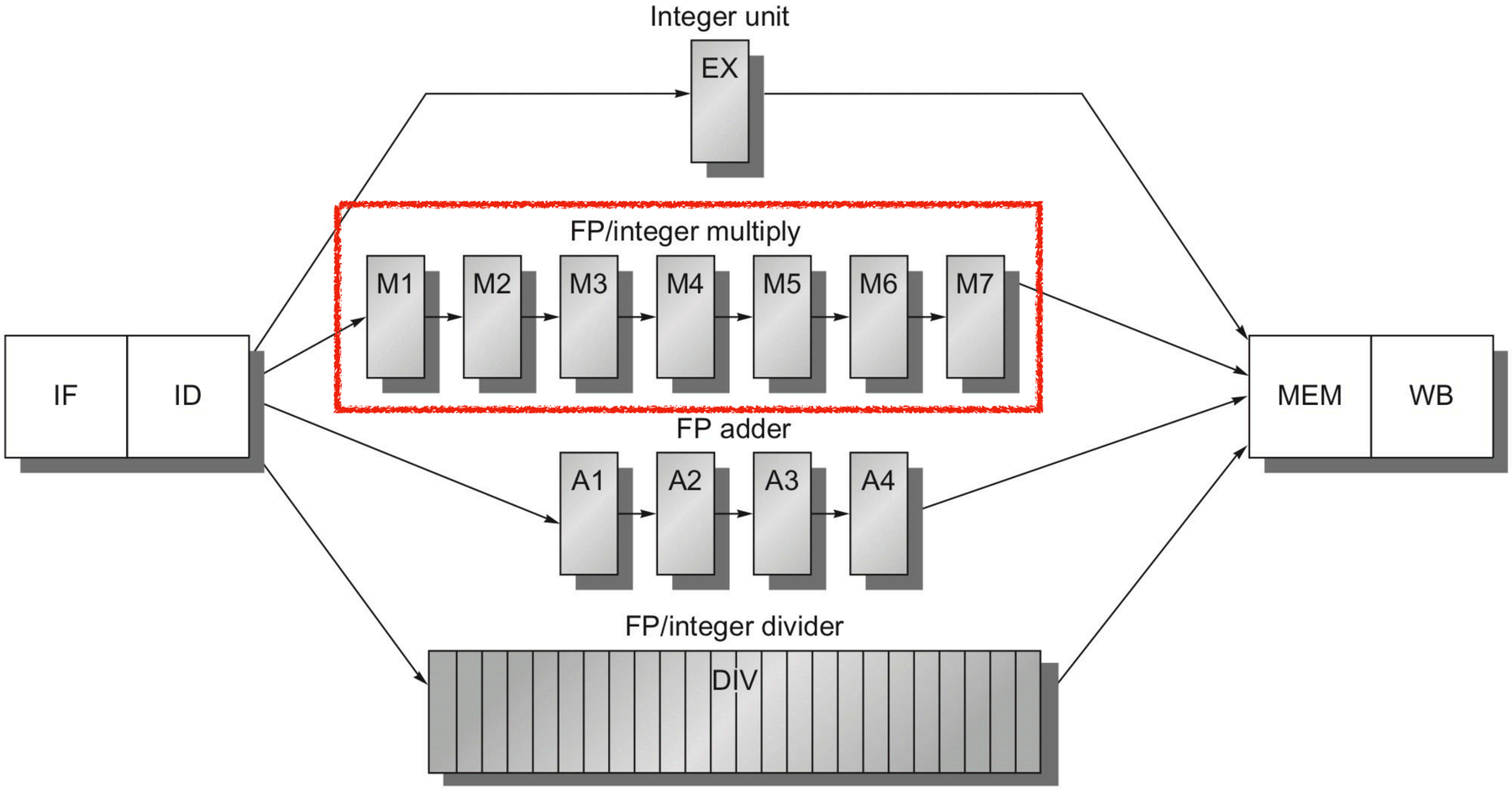


Figure C.30 A pipeline that supports multiple outstanding FP operations. The FP multiplier and adder are fully pipelined and have a depth of seven and four stages, respectively. The FP divider is not pipelined, but requires 24 clock cycles to complete. The latency in instructions between the issue of an FP operation and the use of the result of that operation without incurring a RAW stall is determined by the number of cycles spent in the execution stages. For example, the fourth instruction after an FP add can use the result of the FP add. For integer ALU operations, the depth of the execution pipeline is always one and the next instruction can use the results.

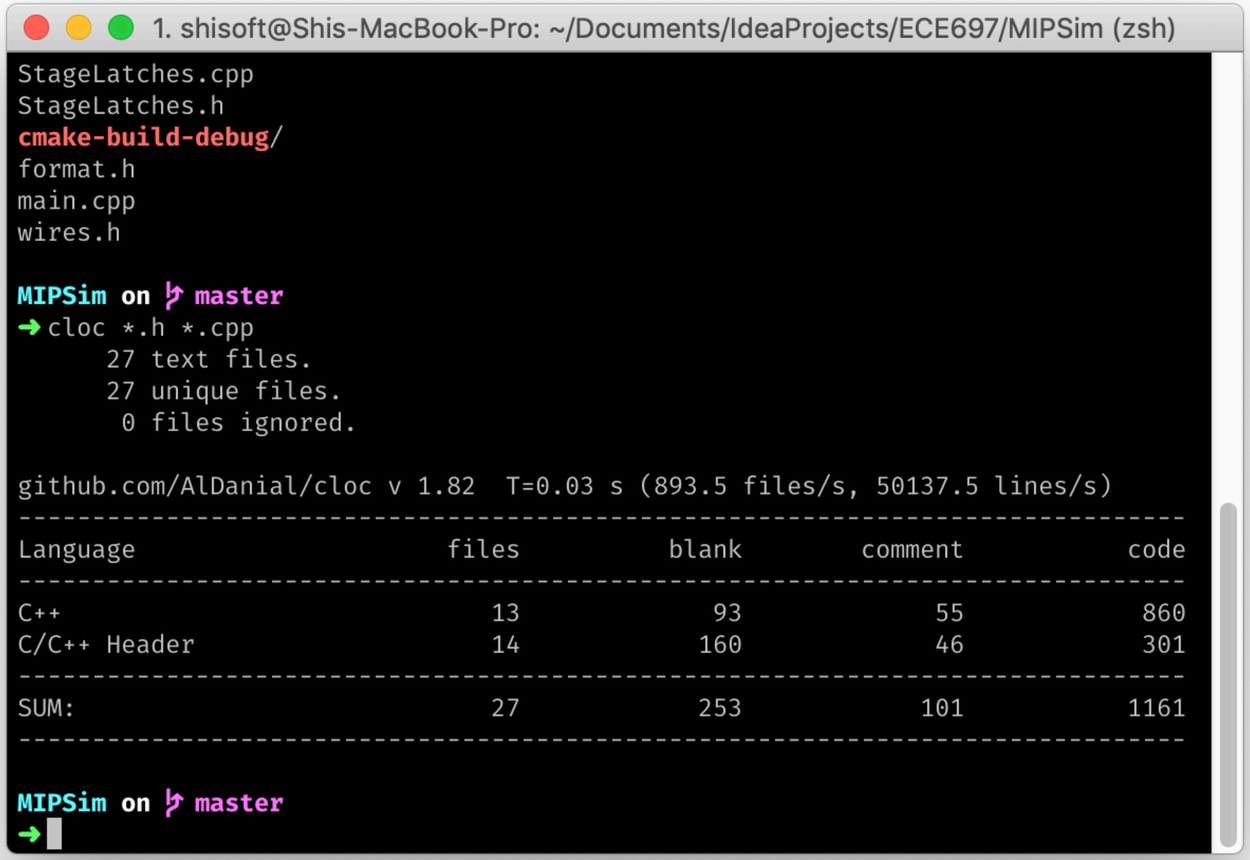
Assembler

- About 200 lines of C++ code
- DFA
- 3 types of tokens (instruction name, register, immediate)
- Enumerate all supporting instructions
- Enables taking ASM code instead of binary instructions

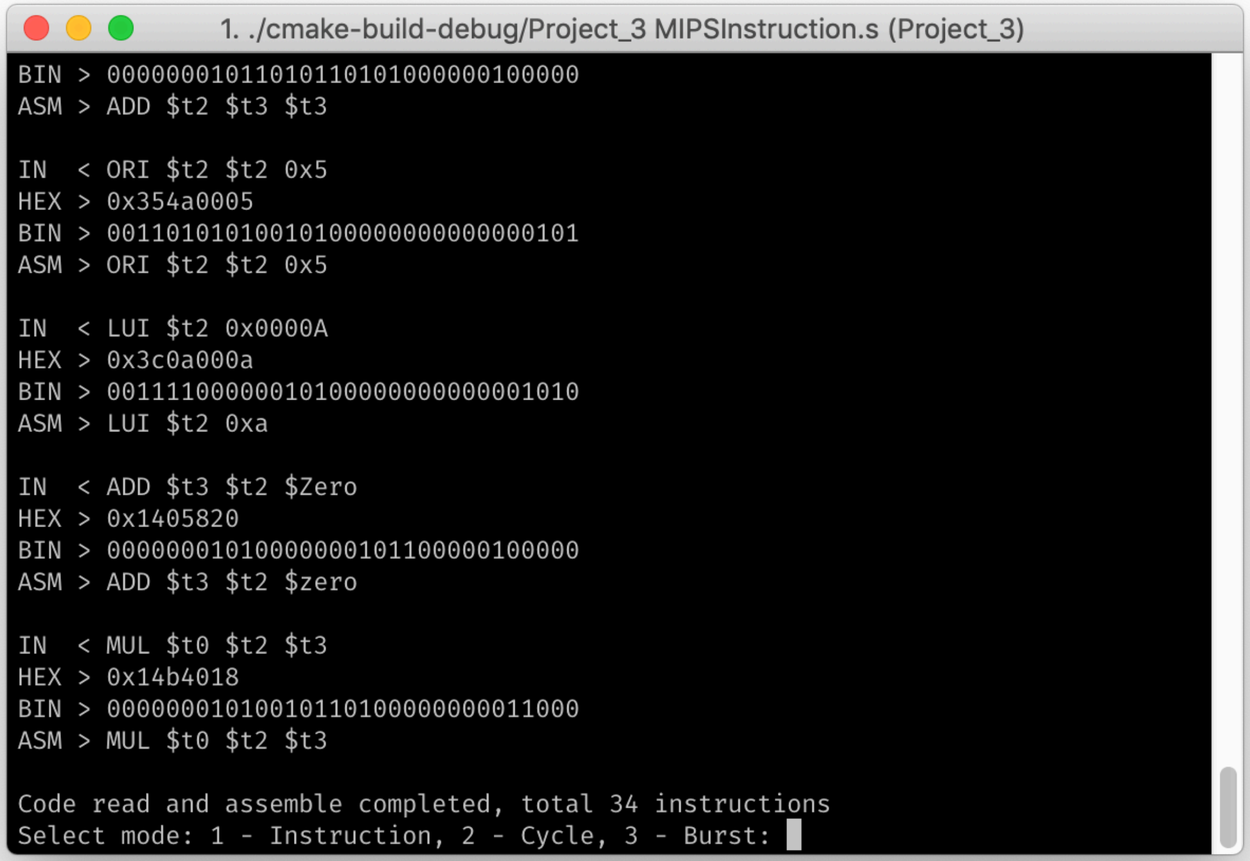
Disassembler

We know how to decode instructions, disassembler is straight forward

Workload


1. shisoft@Shis-MacBook-Pro: ~/Documents/IdeaProjects/ECE697/MIPSim (zsh)
StageLatches.cpp
StageLatches.h
cmake-build-debug/
format.h
main.cpp
wires.h
MIPSim on master
➜ cloc *.h *.cpp
27 text files.
27 unique files.
0 files ignored.
github.com/AlDanial/cloc v 1.82 T=0.03 s (893.5 files/s, 50137.5 lines/s)
Language files blank comment code
C++ 13 93 55 860
C/C++ Header 14 160 46 301
SUM: 27 253 101 1161
MIPSim on master
➜

Screenshot


1. ./cmake-build-debug/Project_3 MIPSInstruction.s (Project_3)
BIN > 00000001011010110101000000100000
ASM > ADD $t2 $t3 $t3
IN < ORI $t2 $t2 0x5
HEX > 0x354a0005
BIN > 00110101010010100000000000000101
ASM > ORI $t2 $t2 0x5
IN < LUI $t2 0x0000A
HEX > 0x3c0a000a
BIN > 00111100000010100000000000001010
ASM > LUI $t2 0xa
IN < ADD $t3 $t2 $Zero
HEX > 0x1405820
BIN > 00000001010000000101100000100000
ASM > ADD $t3 $t2 $zero
IN < MUL $t0 $t2 $t3
HEX > 0x14b4018
BIN > 00000001010010110100000000011000
ASM > MUL $t0 $t2 $t3
Code read and assemble completed, total 34 instructions
Select mode: 1 - Instruction, 2 - Cycle, 3 - Burst:

Demo

See how the simulator works in action

Q & A

Source code available

https://github.com/shisoft/MIPSim

Thank you

Hao Shi
haoshi@umass.edu