ECE4504
Project Phase 2: Pipeline Mechanisms

Emil Bengtsson

October 2018

Implementation

The goal of this phase of the project was to implement different kinds of branch prediction in the baseline EduMIPS64 simulator.

The first thing I implemented was functionality to flush the pipeline after a branch instruction and thus not starting any new instructions before the branch address and condition had been determined. I implemented this by not letting any instructions enter the IF stage of the pipeline while there is a branch instruction in the ID stage of the pipeline. After the branch instruction has passed through the ID stage, both the address and whether the branch is taken is known and new instructions can be issued.

The second thing to implement was functionality to predict that the branches are never taken. This is the default functionality of the EduMIPS64 simulator and I didn't modify it.

The third thing to implement was functionality to predict that the branches are always taken. Since both the branch address and whether the branch will be taken are both calculated in the ID stage, predicting that the branch will be taken is the same as flushing the pipeline and it was implemented the same way as flushing the pipeline was.

The fourth thing to implement was two dynamic branch prediction techniques; a 2-bit local predictor and a (2,1) correlating predictor. I implemented both predictors using a java HashMap with the instruction indices of the branches as keys and the predictors as values being mapped to. When using the 2-bit predictor, each key in the HashMap maps to one predictor. When using the (2,1) correlating predictor, each key in the HashMap maps to four predictors. The HashMap is updated after each branch instruction and it is used for each branch instruction to determine whether to predict branch taken or branch not taken.

When implementing the dynamic prediction techniques I capped the number of memory bits so that both predictors use the same amount of bits. This means the HashMap can only hold half as many entries of the (2,1) correlating predictor as it can of the 2-bit predictor.

Experimental Setup

For evaluation we were asked to find benchmarks on our own. I didn't find much on the internet so I wrote my own. All three benchmarks I have written contain branch instructions and should show differences between the different prediction strategies.

The first benchmark of the three I wrote is called “1to100” and it’s supposed to mimic a for loop. It branches 100 times and increments a counter before each jump, then it writes the value of the counter to memory and ends the program.

The second benchmark I wrote is called “1to50” and it is the same as 1to100, except it only increments the counter 50 times, and there is another branch instruction that is taken quite often but it doesn’t do anything.

The third benchmark program I wrote is called “if_dependencies” inspired by some code I saw in the course book that contained three if statements. The third of these if statements depends on the other two. The benchmark contains “iterates” 50 times. It contains two registers that get incremented in every “iteration”, one register is also equal to modulo 2 of itself and the other is equal to modulo 3 of itself. When these two registers are not equal in an “iteration”, a counter gets incremented. At the end of the program the counter is stored in memory.

In 1to100 the branch is almost always taken, so it’s not a very exciting program and will probably not show any big differences between any of the prediction strategies. 1to50 should show some differences between the static and dynamic prediction strategies, and if_dependencies should show some differences between the two dynamic prediction strategies.

I will evaluate the implementations by running the benchmark programs with each one of the prediction strategies. I will then compare the statistics of each run.

I will refer to the three benchmark programs I use by their names: 1to100, 1to50, and if_dependencies.

Results

Results of the benchmarks running with no branch prediction, just flushing the pipeline:

Benchmark	1to100	1to50	if_dependencies
Cycles	610	860	1118
Instructions	305	455	581
CPI	2.0	1.890	1.924
RAW Stalls	201	202	333
WAW Stalls	0	0	0
WAR Stalls	0	0	0
Structural Stalls (Divider not available)	0	0	0
Structural Stalls (Memory not available)	0	0	0
Branch Stalls	100	199	200

Results of the benchmarks running with branches predicted as not taken:

Benchmark	1to100	1to50	if_dependencies
Cycles	609	810	1045
Instructions	305	455	581
CPI	1.996	1.780	1.798
RAW Stalls	201	203	333
WAW Stalls	0	0	0
WAR Stalls	0	0	0
Structural Stalls (Divider not available)	0	0	0
Structural Stalls (Memory not available)	0	0	0
Branch Taken Stalls	99	148	127
Branch Misprediction Stalls	0	0	0

Results of the benchmarks running with branches predicted as taken:

Benchmark	1to100	1to50	if_dependencies
Cycles	610	860	1118
Instructions	305	455	581
CPI	2.0	1.890	1.924
RAW Stalls	201	202	333
WAW Stalls	0	0	0
WAR Stalls	0	0	0
Structural Stalls (Divider not available)	0	0	0
Structural Stalls (Memory not available)	0	0	0
Branch Taken Stalls	99	148	127
Branch Misprediction Stalls	1	51	73

Results of the benchmarks running with a 2-bit local branch predictor:

Benchmark	1to100	1to50	if_dependencies
Cycles	610	859	1102
Instructions	305	455	581
CPI	2.0	1.887	1.896
RAW Stalls	201	202	333
WAW Stalls	0	0	0
WAR Stalls	0	0	0
Structural Stalls (Divider not available)	0	0	0
Structural Stalls (Memory not available)	0	0	0
Branch Taken Stalls	99	148	127
Branch Misprediction Stalls	1	50	57

Results of the benchmarks running with a (2,1) correlating branch predictor:

Benchmark	1to100	1to50	if_dependencies
Cycles	610	859	1094
Instructions	305	455	581
CPI	2.0	1.887	1.882
RAW Stalls	201	202	333
WAW Stalls	0	0	0
WAR Stalls	0	0	0
Structural Stalls (Divider not available)	0	0	0
Structural Stalls (Memory not available)	0	0	0
Branch Taken Stalls	99	148	127
Branch Misprediction Stalls	1	50	49

Comment on the results

The results from running the benchmarks on the different configurations of the program show that all the predictors are about the same for 1to100 and 1to50 benchmarks, the dynamic predictors are marginally better than always predicting branch taken. The if_dependencies benchmark shows some more interesting results though. Both the dynamic prediction techniques are quite clearly better than always predicting branch taken, and the (2,1) correlating predictor is quite a bit better than the 2-bit local predictor. This is because it also considers recent branch instructions which better captures the behaviour of if statements that depend on each other.

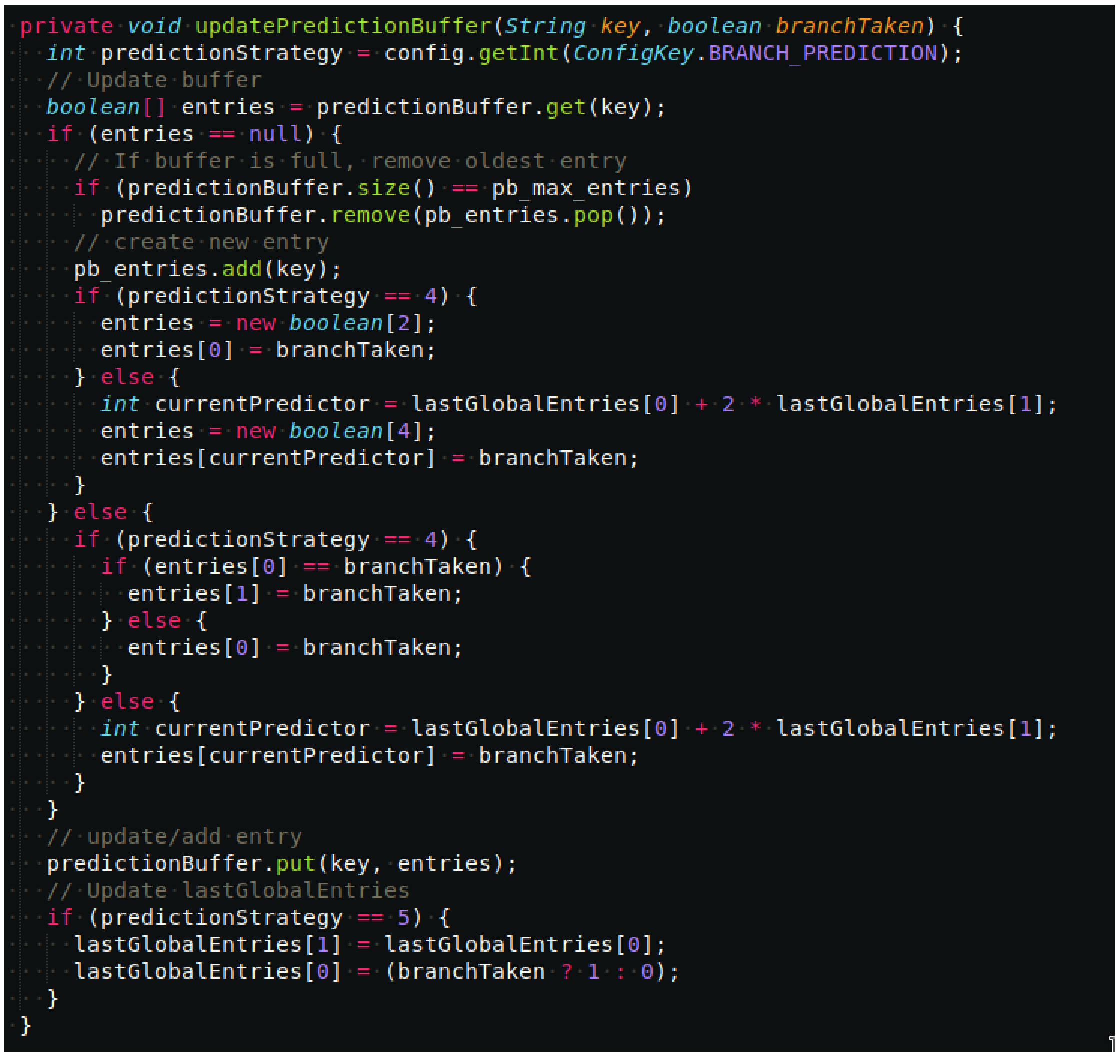
One important thing to note is that predicting that the branch will never be taken is always optimal in this simulator. This is because both the branch address and the branch condition are calculated in the same stage (IF), so predicting that the branch is not taken can not result in a misprediction stall.

For the table that shows the results of the benchmarks running with no branch prediction I chose to only show the total amount of branch related stalls, since calling something a misprediction stall when there was no prediction is not entirely correct.

All in all the results show that the dynamic prediction techniques are pretty good, but in this simulator it is always best to predict that the branch is not taken. The dynamic prediction techniques also require more memory.

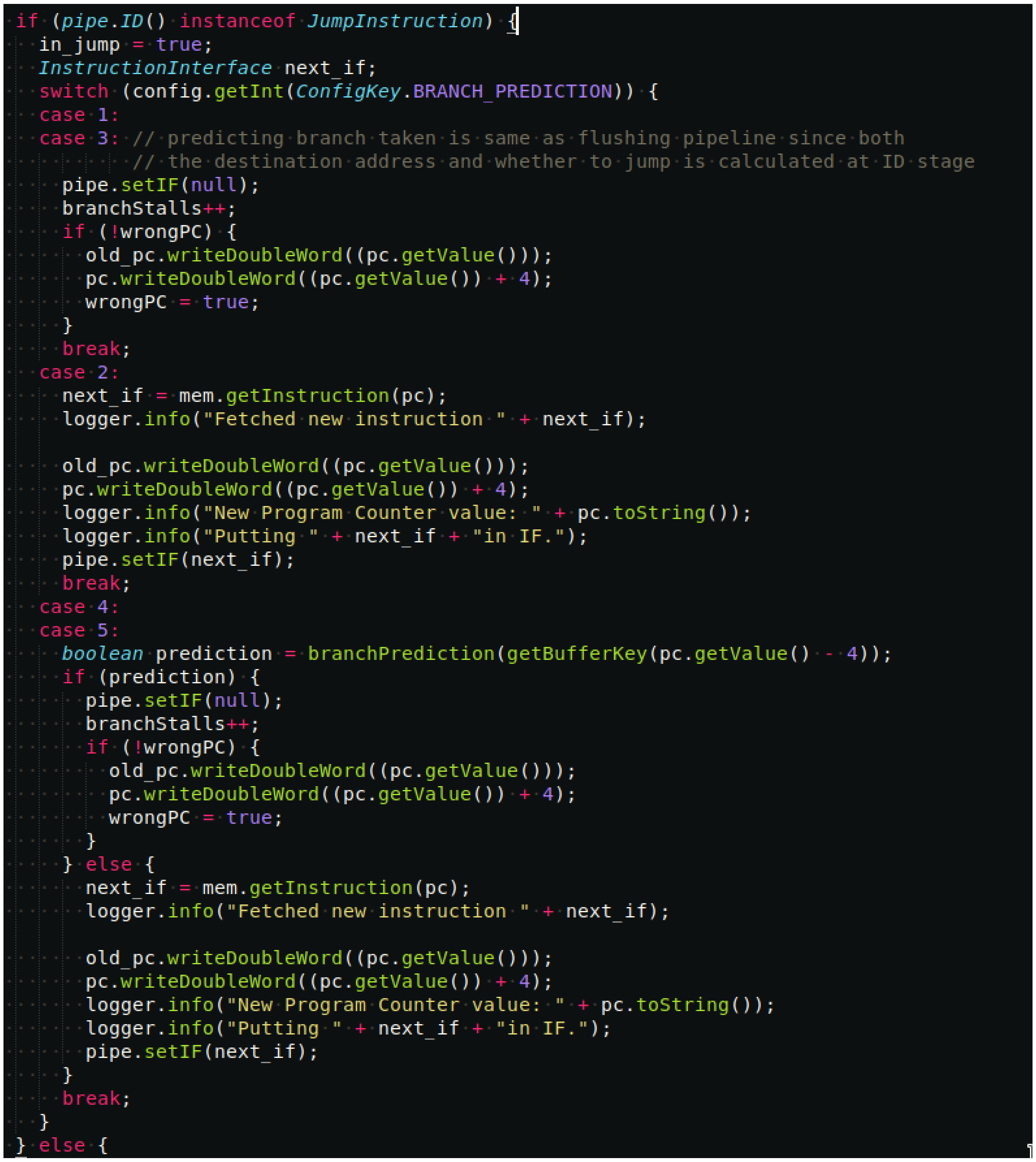
Screen shots

Here are some screen shots of important parts of the code and the program:



```
private void updatePredictionBuffer(String key, boolean branchTaken) {
    int predictionStrategy = config.getInt(ConfigKey.BRANCH_PREDICTION);
    // Update buffer
    boolean[] entries = predictionBuffer.get(key);
    if (entries == null) {
        // If buffer is full, remove oldest entry
        if (predictionBuffer.size() == pb_max_entries)
            predictionBuffer.remove(pb_entries.pop());
        // create new entry
        pb_entries.add(key);
        if (predictionStrategy == 4) {
            entries = new boolean[2];
            entries[0] = branchTaken;
        } else {
            int currentPredictor = lastGlobalEntries[0] + 2 * lastGlobalEntries[1];
            entries = new boolean[4];
            entries[currentPredictor] = branchTaken;
        }
    } else {
        if (predictionStrategy == 4) {
            if (entries[0] == branchTaken) {
                entries[1] = branchTaken;
            } else {
                entries[0] = branchTaken;
            }
        } else {
            int currentPredictor = lastGlobalEntries[0] + 2 * lastGlobalEntries[1];
            entries[currentPredictor] = branchTaken;
        }
    }
    // update/add entry
    predictionBuffer.put(key, entries);
    // Update lastGlobalEntries
    if (predictionStrategy == 5) {
        lastGlobalEntries[1] = lastGlobalEntries[0];
        lastGlobalEntries[0] = (branchTaken ? 1 : 0);
    }
}
```

Figure 1: Code for how the buffer is updated for the dynamic predictors

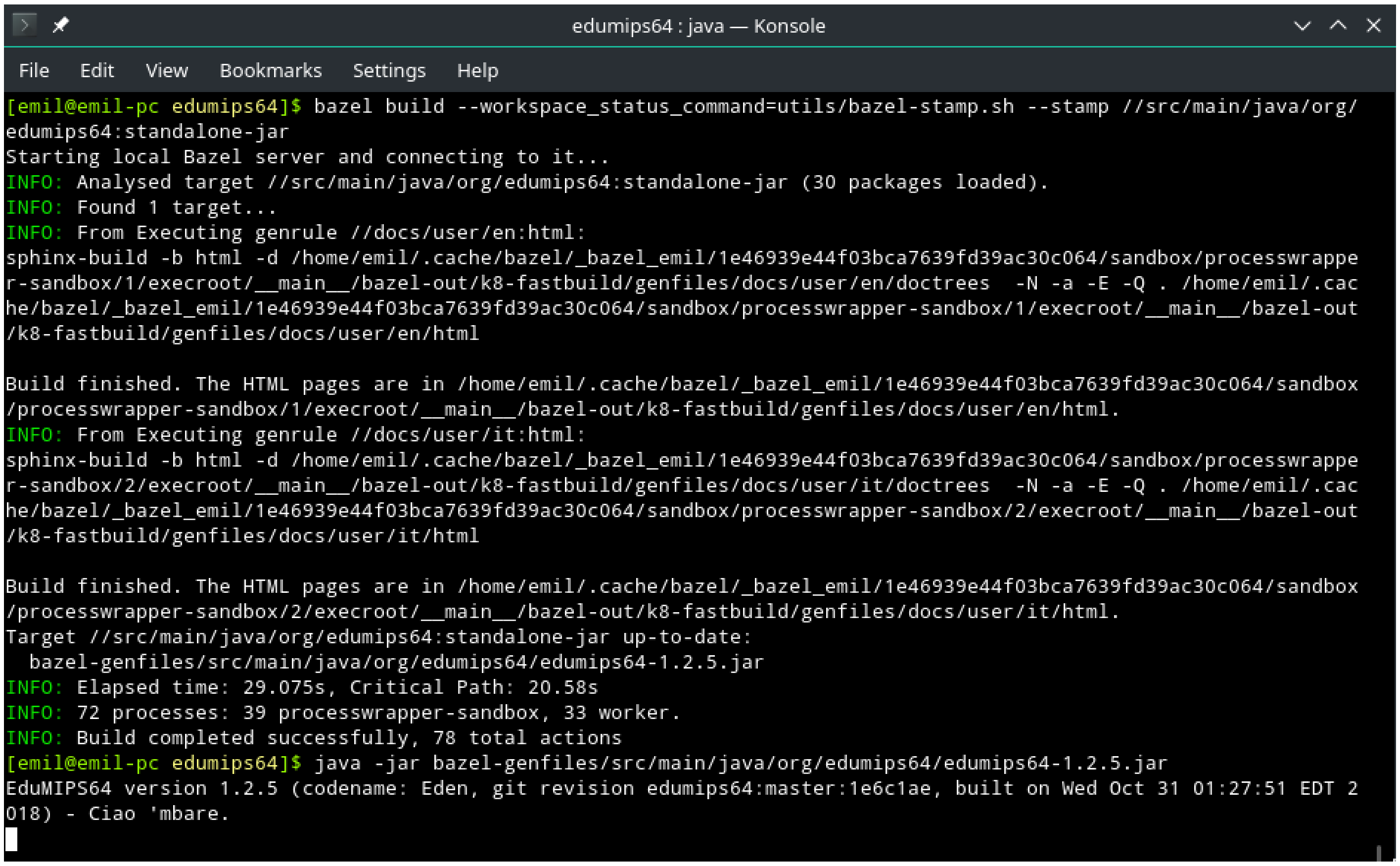


```java
if (pipe.ID() instanceof JumpInstruction) {
  in_jump = true;
  InstructionInterface next_if;
  switch (config.getInt(ConfigKey.BRANCH_PREDICTION)) {
  case 1:
  case 3: // predicting branch taken is same as flushing pipeline since both
          // the destination address and whether to jump is calculated at ID stage
    pipe.setIF(null);
    branchStalls++;
    if (!wrongPC) {
      old_pc.writeDoubleWord((pc.getValue()));
      pc.writeDoubleWord((pc.getValue()) + 4);
      wrongPC = true;
    }
    break;
  case 2:
    next_if = mem.getInstruction(pc);
    logger.info("Fetched new instruction " + next_if);

    old_pc.writeDoubleWord((pc.getValue()));
    pc.writeDoubleWord((pc.getValue()) + 4);
    logger.info("New Program Counter value: " + pc.toString());
    logger.info("Putting " + next_if + "in IF.");
    pipe.setIF(next_if);
    break;
  case 4:
  case 5:
    boolean prediction = branchPrediction(getBufferKey(pc.getValue() - 4));
    if (prediction) {
      pipe.setIF(null);
      branchStalls++;
      if (!wrongPC) {
        old_pc.writeDoubleWord((pc.getValue()));
        pc.writeDoubleWord((pc.getValue()) + 4);
        wrongPC = true;
      }
    } else {
      next_if = mem.getInstruction(pc);
      logger.info("Fetched new instruction " + next_if);

      old_pc.writeDoubleWord((pc.getValue()));
      pc.writeDoubleWord((pc.getValue()) + 4);
      logger.info("New Program Counter value: " + pc.toString());
      logger.info("Putting " + next_if + "in IF.");
      pipe.setIF(next_if);
    }
    break;
  }
} else {
```

Figure 2: Code for how the prediction is done if there is a branch instruction in the ID stage

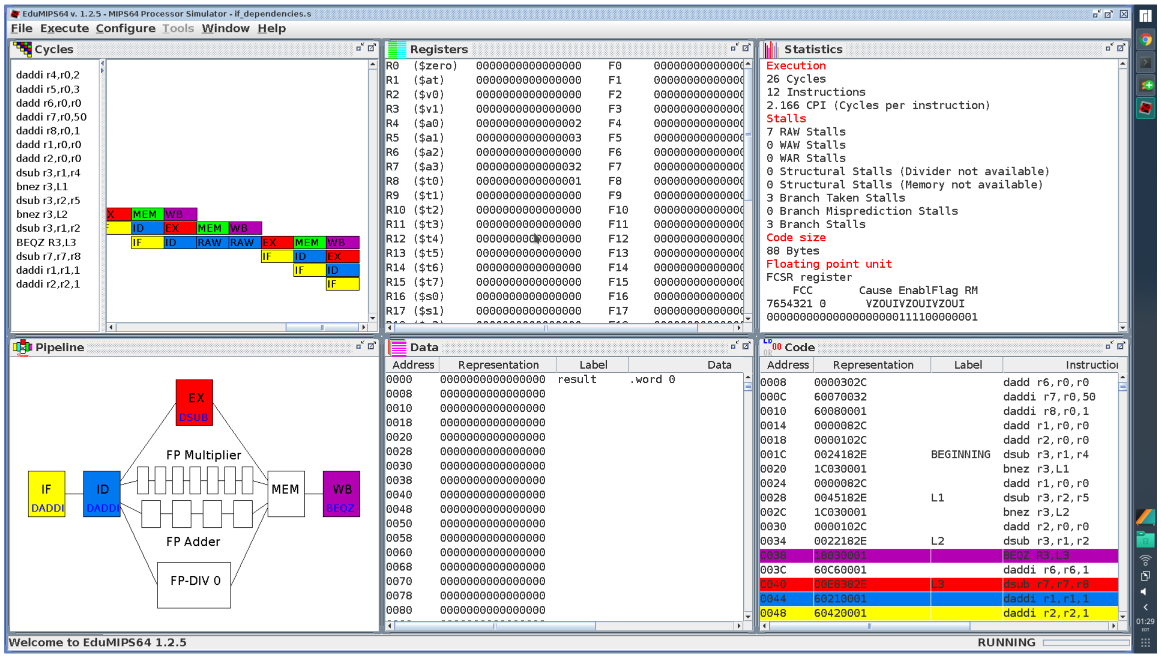

edumips64 : java — Konsole
File Edit View Bookmarks Settings Help

```
[emil@emil-pc edumips64]$ bazel build --workspace_status_command=utils/bazel-stamp.sh --stamp //src/main/java/org/
edumips64:standalone-jar
Starting local Bazel server and connecting to it...
INFO: Analysed target //src/main/java/org/edumips64:standalone-jar (30 packages loaded).
INFO: Found 1 target...
INFO: From Executing genrule //docs/user/en:html:
sphinx-build -b html -d /home/emil/.cache/bazel/_bazel_emil/1e46939e44f03bca7639fd39ac30c064/sandbox/processwrappe
r-sandbox/1/execroot/__main__/bazel-out/k8-fastbuild/genfiles/docs/user/en/doctrees  -N -a -E -Q . /home/emil/.cac
he/bazel/_bazel_emil/1e46939e44f03bca7639fd39ac30c064/sandbox/processwrapper-sandbox/1/execroot/__main__/bazel-out
/k8-fastbuild/genfiles/docs/user/en/html

Build finished. The HTML pages are in /home/emil/.cache/bazel/_bazel_emil/1e46939e44f03bca7639fd39ac30c064/sandbox
/processwrapper-sandbox/1/execroot/__main__/bazel-out/k8-fastbuild/genfiles/docs/user/en/html.
INFO: From Executing genrule //docs/user/it:html:
sphinx-build -b html -d /home/emil/.cache/bazel/_bazel_emil/1e46939e44f03bca7639fd39ac30c064/sandbox/processwrappe
r-sandbox/2/execroot/__main__/bazel-out/k8-fastbuild/genfiles/docs/user/it/doctrees  -N -a -E -Q . /home/emil/.cac
he/bazel/_bazel_emil/1e46939e44f03bca7639fd39ac30c064/sandbox/processwrapper-sandbox/2/execroot/__main__/bazel-out
/k8-fastbuild/genfiles/docs/user/it/html

Build finished. The HTML pages are in /home/emil/.cache/bazel/_bazel_emil/1e46939e44f03bca7639fd39ac30c064/sandbox
/processwrapper-sandbox/2/execroot/__main__/bazel-out/k8-fastbuild/genfiles/docs/user/it/html.
Target //src/main/java/org/edumips64:standalone-jar up-to-date:
  bazel-genfiles/src/main/java/org/edumips64/edumips64-1.2.5.jar
INFO: Elapsed time: 29.075s, Critical Path: 20.58s
INFO: 72 processes: 39 processwrapper-sandbox, 33 worker.
INFO: Build completed successfully, 78 total actions
[emil@emil-pc edumips64]$ java -jar bazel-genfiles/src/main/java/org/edumips64/edumips64-1.2.5.jar
EduMIPS64 version 1.2.5 (codename: Eden, git revision edumips64:master:1e6c1ae, built on Wed Oct 31 01:27:51 EDT 2
018) - Ciao 'mbare.
```

Figure 3: Compiling and running the program


EduMIPS64 v. 1.2.5 - MIPS64 Processor Simulator - if_dependencies.s
File Execute Configure Tools Window Help
Cycles
MEM
WB
ID
EX
RAW
IF
Pipeline
EX
DSUB
FP Multiplier
IF
DADDI
ID
DADDI
MEM
WB
BEQZ
FP Adder
FP-DIV 0
Registers
Statistics
Data
Code
Welcome to EduMIPS64 1.2.5
RUNNING

```
daddi r4,r0,2
daddi r5,r0,3
dadd r6,r0,r0
daddi r7,r0,50
daddi r8,r0,1
dadd r1,r0,r0
dadd r2,r0,r0
dsub r3,r1,r4
bnez r3,L1
dsub r3,r2,r5
bnez r3,L2
dsub r3,r1,r2
BEQZ R3,L3
dsub r7,r7,r8
daddi r1,r1,1
daddi r2,r2,1
```

```
R0   ($zero)   0000000000000000    F0    000000000000000
R1   ($at)     0000000000000000    F1    000000000000000
R2   ($v0)     0000000000000000    F2    000000000000000
R3   ($v1)     0000000000000000    F3    000000000000000
R4   ($a0)     0000000000000002    F4    000000000000000
R5   ($a1)     0000000000000003    F5    000000000000000
R6   ($a2)     0000000000000000    F6    000000000000000
R7   ($a3)     0000000000000032    F7    000000000000000
R8   ($t0)     0000000000000001    F8    000000000000000
R9   ($t1)     0000000000000000    F9    000000000000000
R10  ($t2)     0000000000000000    F10   000000000000000
R11  ($t3)     0000000000000000    F11   000000000000000
R12  ($t4)     0000000000000000    F12   000000000000000
R13  ($t5)     0000000000000000    F13   000000000000000
R14  ($t6)     0000000000000000    F14   000000000000000
R15  ($t7)     0000000000000000    F15   000000000000000
R16  ($s0)     0000000000000000    F16   000000000000000
R17  ($s1)     0000000000000000    F17   000000000000000
```

```
Execution
26 Cycles
12 Instructions
2.166 CPI (Cycles per instruction)
Stalls
7 RAW Stalls
0 WAW Stalls
0 WAR Stalls
0 Structural Stalls (Divider not available)
0 Structural Stalls (Memory not available)
3 Branch Taken Stalls
0 Branch Misprediction Stalls
3 Branch Stalls
Code size
88 Bytes
Floating point unit
FCSR register
    FCC       Cause EnablFlag RM
7654321 0      VZOUIVZOUIVZOUI
0000000000000000000000111100000001
```

```
Address  Representation    Label     Data
0000     0000000000000000  result    .word 0
0008     0000000000000000
0010     0000000000000000
0018     0000000000000000
0020     0000000000000000
0028     0000000000000000
0030     0000000000000000
0038     0000000000000000
0040     0000000000000000
0048     0000000000000000
0050     0000000000000000
0058     0000000000000000
0060     0000000000000000
0068     0000000000000000
0070     0000000000000000
0078     0000000000000000
0080     0000000000000000
```

```
Address  Representation  Label      Instructio
0008     0000302C                   dadd r6,r0,r0
000C     60070032                   daddi r7,r0,50
0010     60080001                   daddi r8,r0,1
0014     0000082C                   dadd r1,r0,r0
0018     0000102C                   dadd r2,r0,r0
001C     0024182E        BEGINNING  dsub r3,r1,r4
0020     1C030001                   bnez r3,L1
0024     0000082C                   dadd r1,r0,r0
0028     0045182E        L1         dsub r3,r2,r5
002C     1C030001                   bnez r3,L2
0030     0000102C                   dadd r2,r0,r0
0034     0022182E        L2         dsub r3,r1,r2
0038     18030001                   BEQZ R3,L3
003C     60C60001                   daddi r6,r6,1
0040     00E8382E        L3         dsub r7,r7,r8
0044     60210001                   daddi r1,r1,1
0048     60420001                   daddi r2,r2,1
```

Figure 4: Program running without a prediction strategy (flushing the pipeline)