Gebze Technical University
Department of Computer Engineering
CSE 531 Advanced Topics in Computer Architecture
HW3 Report

H.Yavuz ERZURUMLU
181041004

Abstract

In this project, I have designed and implemented a data and hazard detector using **Java**. The program takes simple assembly program and convert it to the dependency free version with following properties:

- All MIPS instructions are supported.(+)
- CopyPaste dynamic code editing support.(+)
- Finding Data Hazards.
- Finding Control Hazards.
- Fixing Data Hazards.
- Fixing Control Hazards.
- Visualizing input dependency graph(+)

You can access the whole source code and paper which used in this report from:

- https://github.com/freeloki/CSE531-HazardFreeAssembly-HW3

1 Introduction - Pipelining

Pipelining is an implementation technique in which multiple instructions are overlapped in execution. Today, pipelining is nearly universal.

Anyone who has done a lot of laundry has intuitively used pipelining. The nonpipelined approach to laundry would be:

- Place one dirty load of clothes in the washer.
- When the washer is finished, place the wet load in the dryer.
- When the dryer is finished, place the dry load on a table and fold.
- When folding is finished, ask your roommate to put the clothes away.

When your roommate is done, then start over with the next dirty load. The pipelined approach takes much less time, as figure shows. As soon as the washer is finished with the first load and placed in the dryer, you load the washer with the second dirty load. When the first load is dry, you place it on the table to start folding, move the wet load to the dryer, and the next dirty load into the washer. Next you have your roommate put the first load away, you start fold- ing the second load, the dryer has the third load, and you put the fourth load into the washer. At this point all steps—called stages in pipelining—are operating con- currently. As long as we have separate resources for each stage, we can pipeline the tasks.

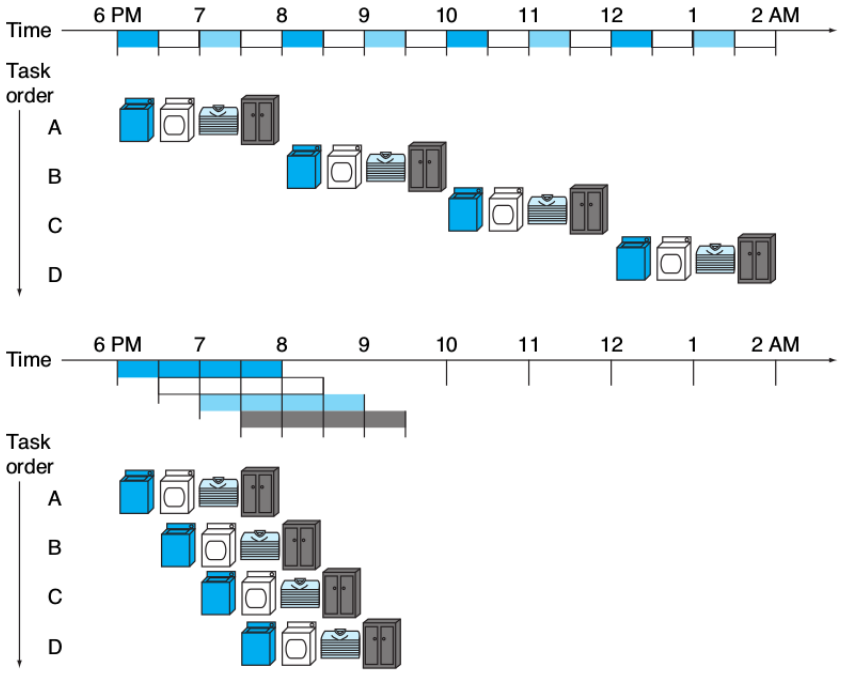

Figure 1: Pipelining Example

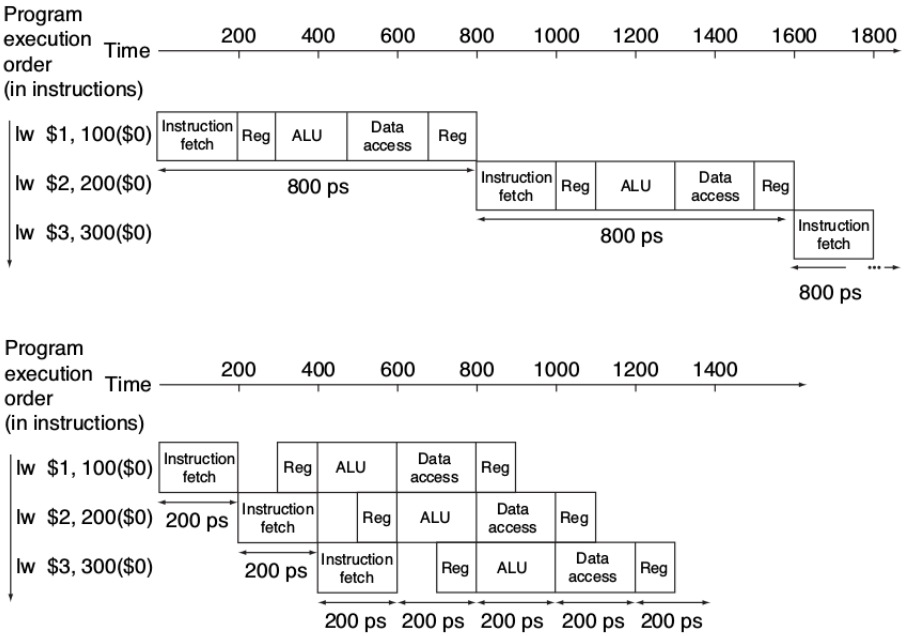

Figure 2: Pipelining on MIPS Instructions

2 Tools

Here is the list of tools which I have used in this project.

- **Netbeans IDE.**
- **MARS MIPS Compiler.**
- **Github.**

- **Java Swing Framework.**
- **Maven.**

3 Data and Control Hazards

Data hazards occur when the pipeline must be stalled because one step must wait for another to complete. Suppose you found a sock at the folding station for which no match existed. One possible strategy is to run down to your room and search through your clothes bureau to see if you can find the match. Obviously, while you are doing the search, loads that have completed drying and are ready to fold and those that have finished washing and are ready to dry must wait. In a computer pipeline, data hazards arise from the dependence of one instruction on an earlier one that is still in the pipeline (a relationship that does not really exist when doing laundry). For example, suppose we have an add instruction fol lowed immediately by a subtract instruction that uses the sum ($s0):

- add $s0, $t0, $t1
- sub $t2, $s0, $t3

Without intervention, a data hazard could severely stall the pipeline. The *add* instruction doesn't write its result until the fifth stage, meaning that we would have to waste three clock cycles in the pipeline.

A control hazard, arising from the need to make a decision based on the results of one instruction while others are executing. Suppose our laundry crew was given the happy task of cleaning the uniforms of a football team. Given how filthy the laundry is, we need to determine whether the detergent and water temperature setting we select is strong enough to get the uniforms clean but not so strong that the uniforms wear out sooner. In our laundry pipeline, we have to wait until the second stage to examine the dry uniform to see. if we need to change the washer setup or not.

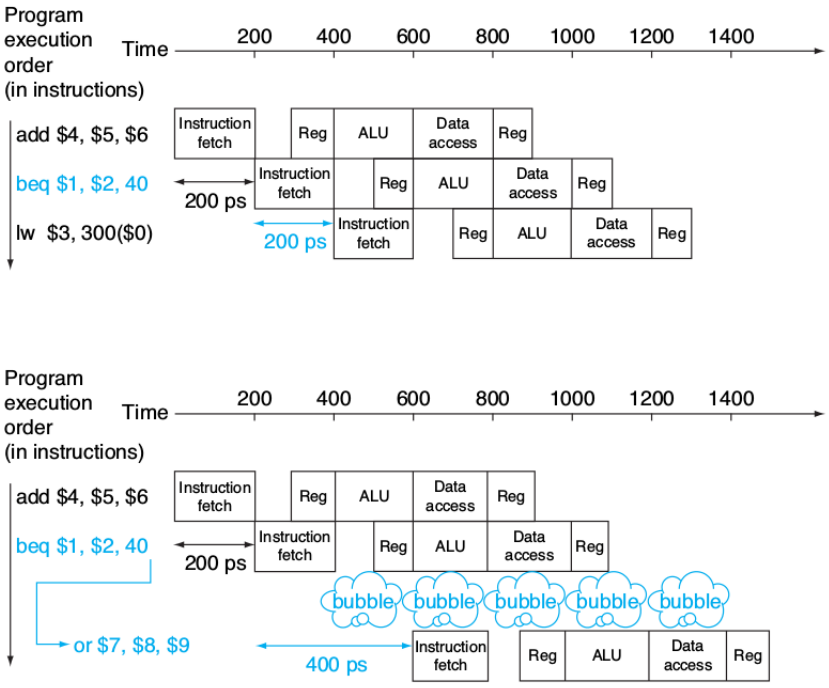

Figure 3: Control Hazard

3.1 Parsing Input

All inputs are parsing line by line and extracting according to their *rd*, *rs*, *rt* specs. Here is a sample result for parsing:

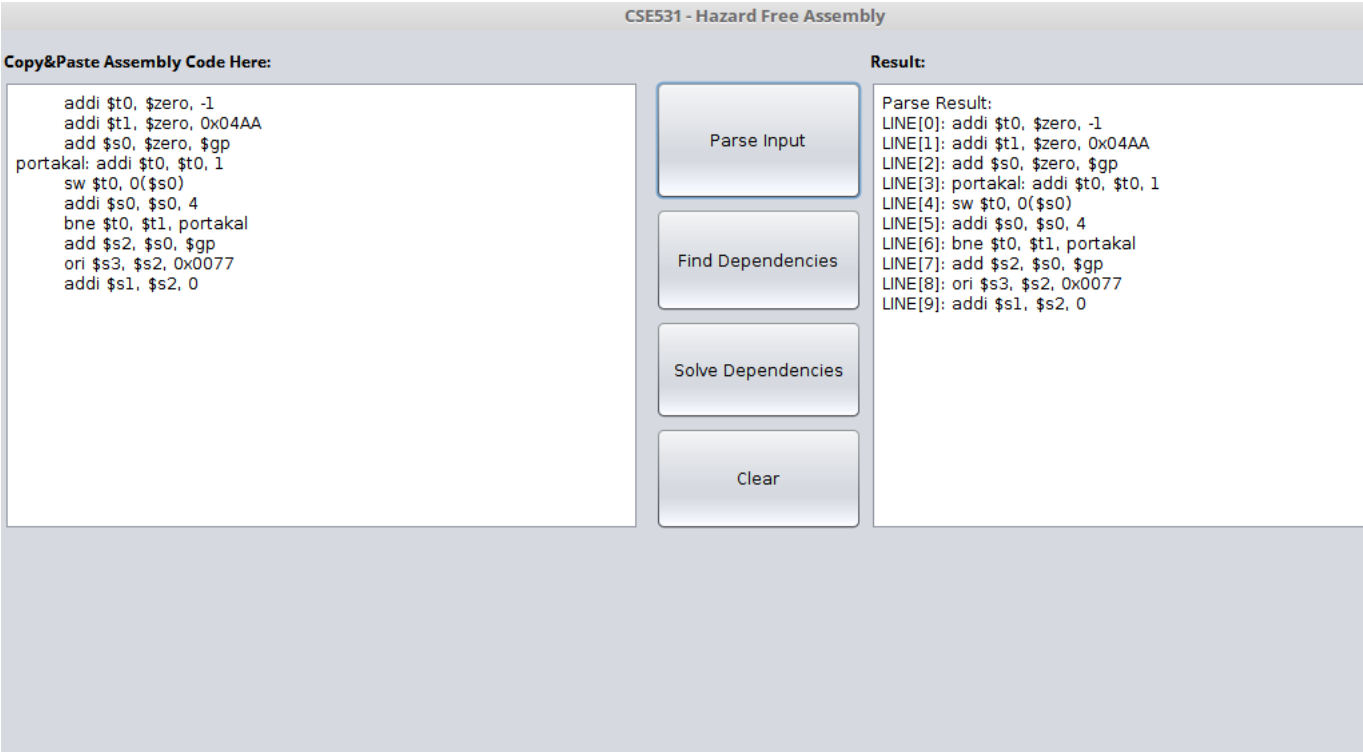

Figure 4: Successful Parse

3.2 Finding Dependencies

For each line data and control hazards are checking separately. A dependency arraylist keeps track of these dependencies.

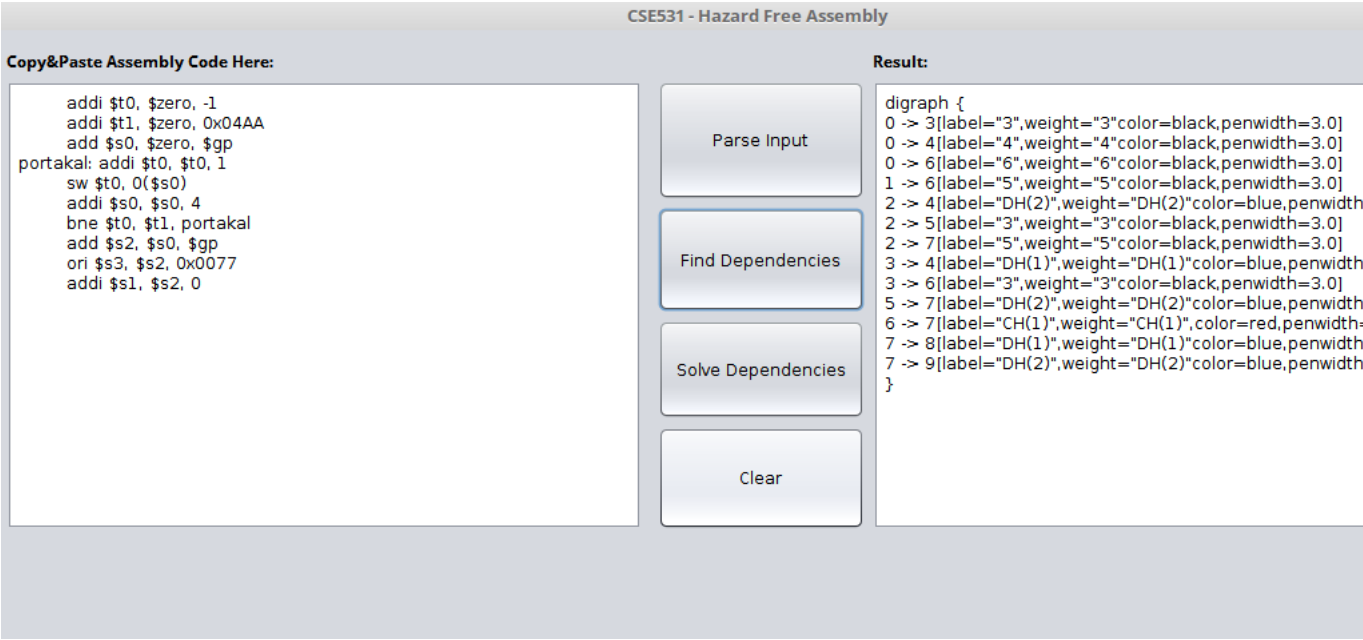

Figure 5: Dependencies

3.3 Solving Dependencies

After creating dependency graph it's easier to solve data and control hazards. Here is the dependency solving algorithm:

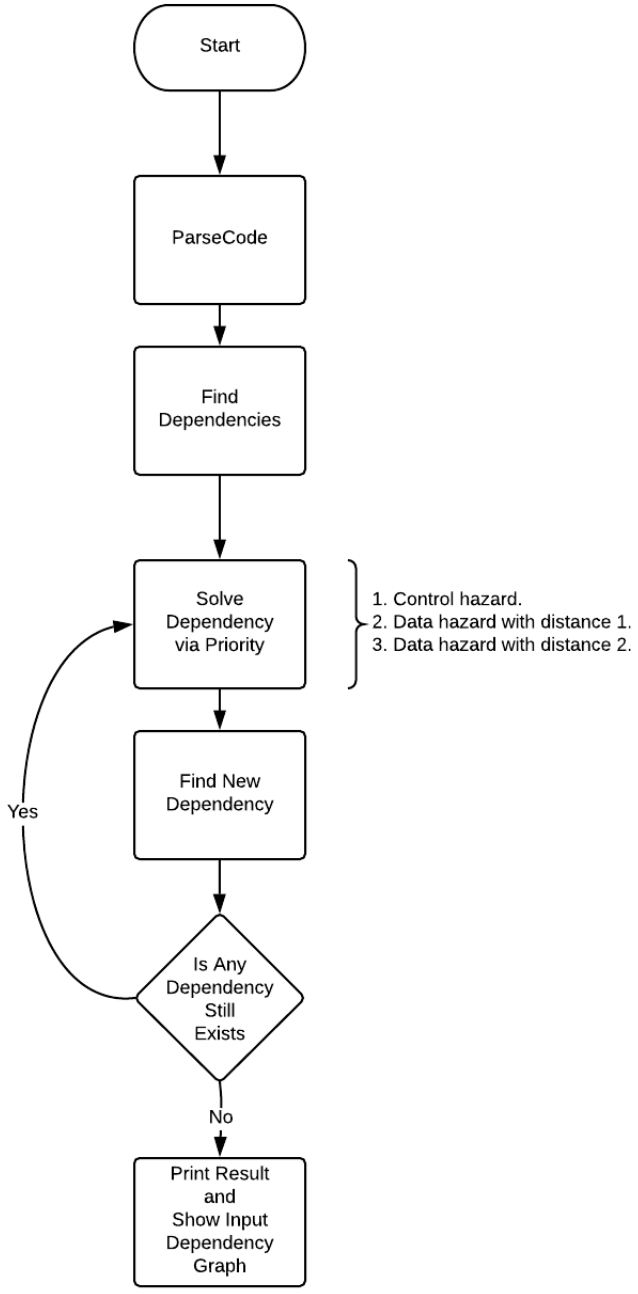

Figure 6: Dependency Solving Sequence

3.4 Visualizing Dependency Graph

I have used **GraphViz** library for visualizing dependecy graph. This library takes the **.dot** which one the popular graph representation language and generates the directed graph.

Here is sample output for input dependency:

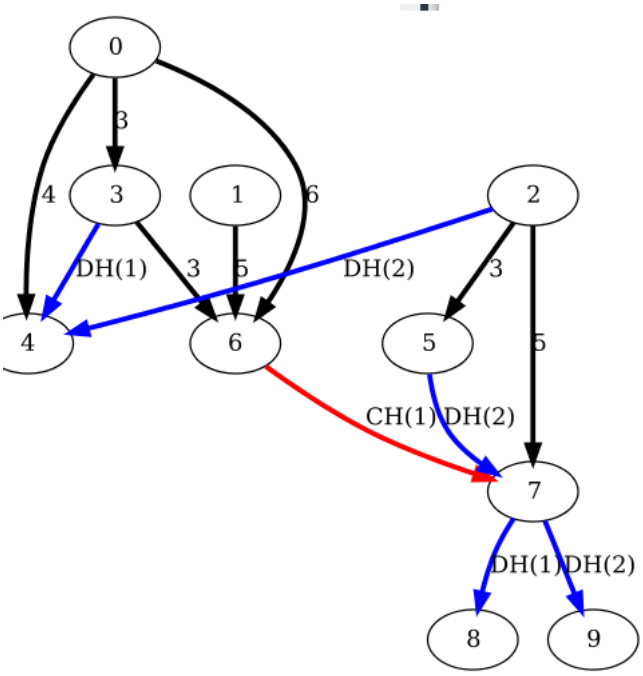

Figure 7: Input dependency graph using GraphViz

4 Appendix

4.1 Supported MIPS Instructions

Arithmetic and Logical Instructions

Instruction	Operation
add $d, $s, $t	`$d = $s + $t`
addu $d, $s, $t	`$d = $s + $t`
addi $t, $s, i	`$t = $s + SE(i)`
addiu $t, $s, i	`$t = $s + SE(i)`
and $d, $s, $t	`$d = $s & $t`
andi $t, $s, i	`$t = $s & ZE(i)`
div $s, $t	`lo = $s / $t; hi = $s % $t`
divu $s, $t	`lo = $s / $t; hi = $s % $t`
mult $s, $t	`hi:lo = $s * $t`
multu $s, $t	`hi:lo = $s * $t`
nor $d, $s, $t	`$d = ~($s | $t)`
or $d, $s, $t	`$d = $s | $t`
ori $t, $s, i	`$t = $s | ZE(i)`
sll $d, $t, a	`$d = $t << a`
sllv $d, $t, $s	`$d = $t << $s`
sra $d, $t, a	`$d = $t >> a`
srav $d, $t, $s	`$d = $t >> $s`
srl $d, $t, a	`$d = $t >>> a`
srlv $d, $t, $s	`$d = $t >>> $s`
sub $d, $s, $t	`$d = $s - $t`
subu $d, $s, $t	`$d = $s - $t`
xor $d, $s, $t	`$d = $s ^ $t`
xori $d, $s, i	`$d = $s ^ ZE(i)`

Constant-Manipulating Instructions

Instruction	Operation
lhi $t, i	`HH($t) = i`
llo $t, i	`LH($t) = i`

Comparison Instructions

Instruction	Operation
slt $d, $s, $t	`$d = ($s < $t)`
sltu $d, $s, $t	`$d = ($s < $t)`
slti $t, $s, i	`$t = ($s < SE(i))`
sltiu $t, $s, i	`$t = ($s < SE(i))`

Branch Instructions

Instruction	Operation
beq $s, $t, label	`if ($s == $t) pc += i << 2`
bgtz $s, label	`if ($s > 0) pc += i << 2`
blez $s, label	`if ($s <= 0) pc += i << 2`
bne $s, $t, label	`if ($s != $t) pc += i << 2`

Jump Instructions

Instruction	Operation
j label	`pc += i << 2`
jal label	`$31 = pc; pc += i << 2`
jalr $s	`$31 = pc; pc = $s`
jr $s	`pc = $s`

Load Instructions

Instruction	Operation
lb $t, i($s)	`$t = SE (MEM [$s + i]:1)`
lbu $t, i($s)	`$t = ZE (MEM [$s + i]:1)`
lh $t, i($s)	`$t = SE (MEM [$s + i]:2)`
lhu $t, i($s)	`$t = ZE (MEM [$s + i]:2)`
lw $t, i($s)	`$t = MEM [$s + i]:4`

Store Instructions

Instruction	Operation
sb $t, i($s)	`MEM [$s + i]:1 = LB ($t)`
sh $t, i($s)	`MEM [$s + i]:2 = LH ($t)`
sw $t, i($s)	`MEM [$s + i]:4 = $t`

Data Movement Instructions

Instruction	Operation
mfhi $d	`$d = hi`
mflo $d	`$d = lo`
mthi $s	`hi = $s`
mtlo $s	`lo = $s`

Exception and Interrupt Instructions

Instruction	Operation
trap 1	Print integer value in $4
trap 5	Read integer value into $2
trap 10	Terminate program execution
trap 101	Print ASCII character in $4
trap 102	Read ASCII character into $2

Figure 8: Supported Instructions

4.2 Sample Results and Output

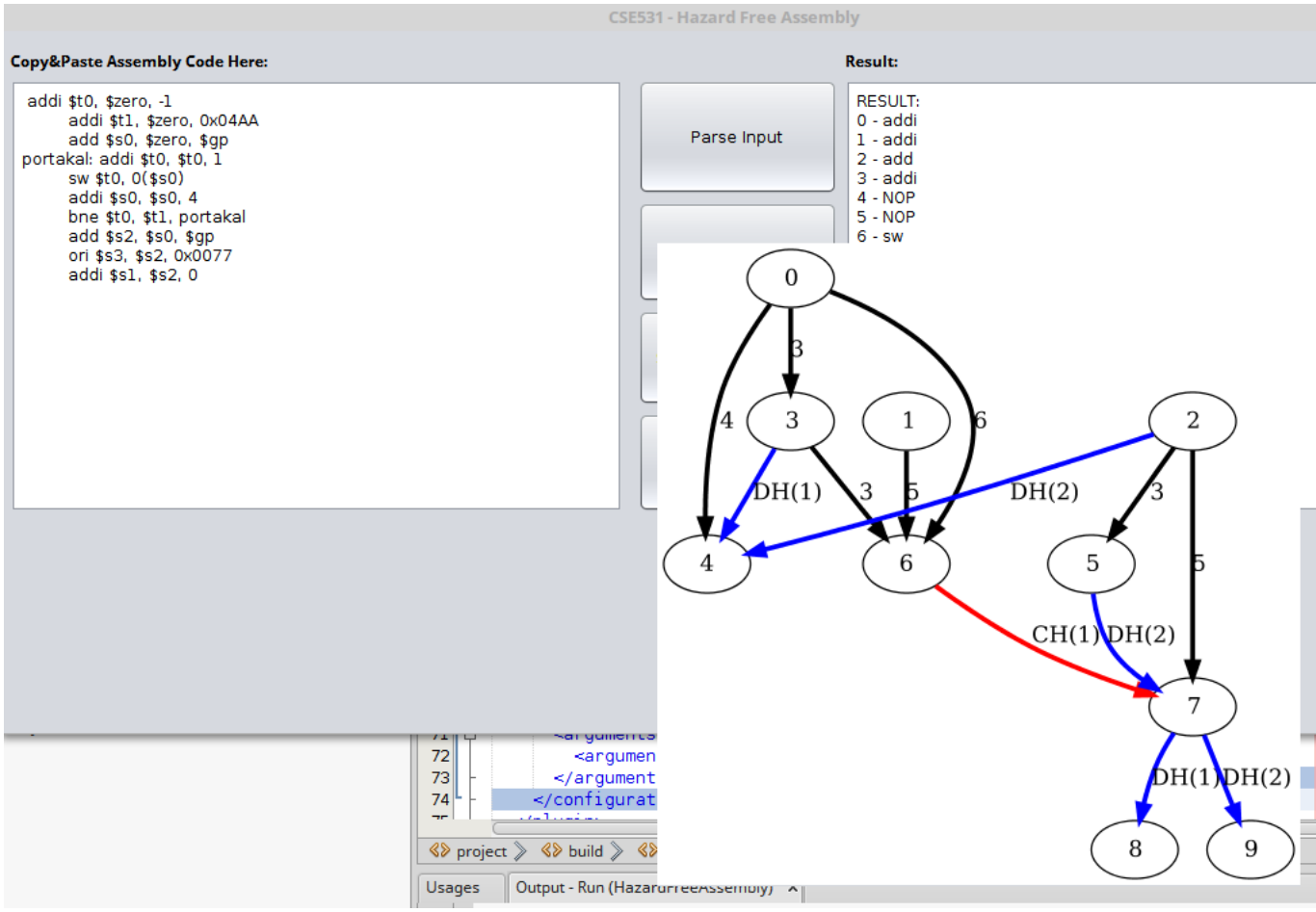

Figure 9: Dependency Graph and Result

Figure 10: Result