Project 3 – Pipelining

CS 3339

Lee Hinkle

With deep acknowledgement to Dr. Martin Burtscher and Ms. Molly O’Neil

Ahhh the hardware versus software debate

Hardware:

Costs a lot initially

Every unit produced at some incremental cost

Cost to change high

Sustaining cost low, unless you screw up

Highest performance usually

Software:

Can cost little initially

Incremental cost of one more unit essentially zero

Change expected

Sustaining cost high, especially if you screw up

LOOK AT WHAT YOU'VE DONE! BAD DESIGNER! BAD, BAD DESIGNER!
WHAT A MORON!.. HEY, BOB! YOU'LL NEVER BREAK HIM OF THE HABIT THAT WAY! YOU'VE GOT TO CATCH HIM IN THE ACT!!

Compiler NOP versus Hardware Stall

"This data hazard can be detected quite easily when the program's machine code is written by the compiler. The original Stanford RISC machine relied on the compiler to add the NOP instructions in this case, rather than having the circuitry to detect and (more taxingly) stall the first two pipeline stages. Hence the name MIPS: **M**icroprocessor without **I**nterlocked **P**ipeline **S**tages. It turned out that the extra NOP instructions added by the compiler expanded the program binaries enough that the instruction cache hit rate was reduced. The stall hardware, although expensive, was put back into later designs to improve instruction cache hit rate, at which point the acronym no longer made sense."

From https://en.wikipedia.org/wiki/Classic_RISC_pipeline accessed 2/26/2017

What you have done so far.

Project 1

- Understood and written code to decode the various fields in a MIPS 32-bit instruction
- Built a partially complete but functional disassembler

Project 2

- Used decoded information from the 32-bit instruction to setup the control signals to emulate the functionality of a processor

What you need to do

- Get your Project 2 running for all inputs
  - Use the debug files – they are your best source for pinpointing where the errors occur
- Follow instructions in Project 3 pdf on TRACS
- You will be adding a Stats.cpp/.h to your P2 code
- Instantiate a stats object in CPU.h – add code to CPU.cpp to call stats as needed and report results
- Complete the Stats.cpp (and modify Stats.h if necessary)
- Follow submission instructions carefully!

#%@%! Project 2!

```
opcode = instr >> 26;
   rs = instr >> 21 & 0x1F;
   rt = instr >> 16 & 0x1F;
   rd = instr >> 11 & 0x1F;
   shamt = instr >> 6 & 0x1F;
   funct = instr & 0x3F;
   uimm = instr & 0xFFFF;
   simm = instr & 0xFFFF;
   addr = instr & 0xFFFFFF;
```

This code has bugs
DON'T COPY IT!

simm must have proper sign extension, in this case it's always 0, should be sign bit

addr field is 26 bits using 0xFFFFFF as a mask only gives 24 bits.

#%@%! Project 2!

```
// Hint: you probably want to give all the control signals some "safe"
// default value here, and then override their values as necessary in each
// case statement below!

    opIsLoad = false;
    opIsStore = false;
    opIsMultDiv = false;
    ALU_OP aluOp;        ←
    writeDest = false;
    int destReg = 0;     ←
    aluSrc1 = 0;
    aluSrc2 = 0;
    storeData = 0;
    aluOut;      }
    writeData;   }
```

This code has bugs
DON'T COPY IT!

aluOp already exists – set it to a value

re-declares an existing value

these are not control signals

#%@%! Project 2!

```
case 0x23: D(cout << "subu " << regNames[rd] << ", " << regNames[rs] << ",
" << regNames[rt]);^M
                   writeDest =true;^M
                   aluSrc1 = regFile[rs];^M
                   aluSrc2 = regFile[rt];^M
                   destReg = rd;^M
                   break;^M
```

This code has bugs
DON'T COPY IT!

aluOp is what?

#%@%! Project 2!

```
 case 0x10: D(cout << "mfhi " << regNames[rd]);^M
^M
                   aluOp = ADD;
                   writeDest = true;
                   aluSrc1 = hi;
                   destReg = rd;
                   break;^M
```

This code has bugs
DON'T COPY IT!

aluSrc2 is what? Even if default is correct value, explicitly setting it here makes the code much easier to understand and debug

#%@%! Project 2!

```
 case 0x00: D(cout << "sll " << regNames[rd] << ", " << regNames[rs] << ",
" << dec << shamt);

     writeDest = true;

     destReg = regFile[rd];

     aluOp = SHF_L;

     aluSrc1 = regFile[rs];

     aluSrc2 = shamt;
```

This code has bugs
DON'T COPY IT!

What register do the results get written to?
0x442cfff0 ??? (or something like that?)

#%@%! Project 2!

This code has bugs
DON'T COPY IT!

```
case 0x02: D(cout << "j " << hex << ((pc & 0xf0000000) | addr << 2));
           aluOp = ADD;
           aluSrc1 = pc; destReg = pc; writeDest = true;
```

read the assignment carefully.   Set pc directly as in provide example for jal – but don't copy all of the jal lines, you don't need them and they will mess up later operations

#%@%! Project 2!

This code has bugs
DON'T COPY IT!

```
case 0x2b: D(cout << "sw " << regNames[rt] << ", " << dec << simm << "(" << regNames[rs] <<
")");
    aluOp = ADD;
    aluSrc1 = regFile[rt]; aluSrc2 = regFile[simm]; destReg = rt;    writeDest = true; break;
```

no opIsStore
aluSrc2 points to what?
aluSrc 1 points to what?
destReg and writeDest for sw?
what is storeData????
Hard to find errors with single line of code

Code Format Example

```
writeDest = true; destReg = rd;
aluOp = SHF_L;
aluSrc1 = regFile[rs];
aluSrc2 = shamt;
break;
```

reminder – greensheet differs for sll and sra – follow debug comments to match binaries & generate correct output

Project 3 Tasks

Complete the provided Stats class to:

- Count cycles including flushes and bubbles
- Detect data hazards
- "Add" the correct number of bubbles and flushes to insure correct operation
  - "Add" is in quotes because you are not actually changing the operation of your single cycle emulator. Stats keeps track of how many additional cycles would be required if you implemented the design with a 8 stage pipeline and no forwarding paths.
- Report statistics on cycle counts, bubbles, flushes, memory operations, and branches.

Modify CPU.cpp as necessary to report the information needed by the stats object.

Project 3 - continued

Suggestions:

- Print Stats.h, Stats.cpp, project assignment
- memop and branch count are good starting points
- use Debug.h techniques for outputs to console
- use disassembly functions from proj1/2 to see instr
- careful with outputs – can make runtime long
- single lines help you keep track of things

```
writeDest = true; destReg = REG_RA;stats.registerDest(REG_RA);
```

if you have this, you better declare this and this

Code Format Example – stats calls

This tells stats "I am going to write to the register number rd"

```
writeDest = true; destReg = rd; stats.registerDest(rd);

aluOp = SHF_L;

aluSrc1 = regFile[rs]; stats.registerSrc(rs);

aluSrc2 = shamt; //don't need to call stats for this one

break;
```

This tells stats "I need to use the value in rs – if someone before me is still writing it, please add bubbles to make sure it's ready"

You live in decode – that is the frame of reference for everything!

From Handout

“The pipeline has the following stages:

IF1, IF2, ID, EXE1, EXE2, MEM1, MEM2, WB”

This has the same functions as the 5-stage version but is more realistic for a cycle accurate simulation.

“ID must track the destination registers and cycles-until-available information for instructions later in the pipeline, so that it can detect hazards and insert the correct number of bubbles.”

This data structure initialized in the constructor will help

```
for(int i = IF1; i < PIPESTAGES; i++) {
     resultReg[i] = -1;   }
```

Are the sources I need available?

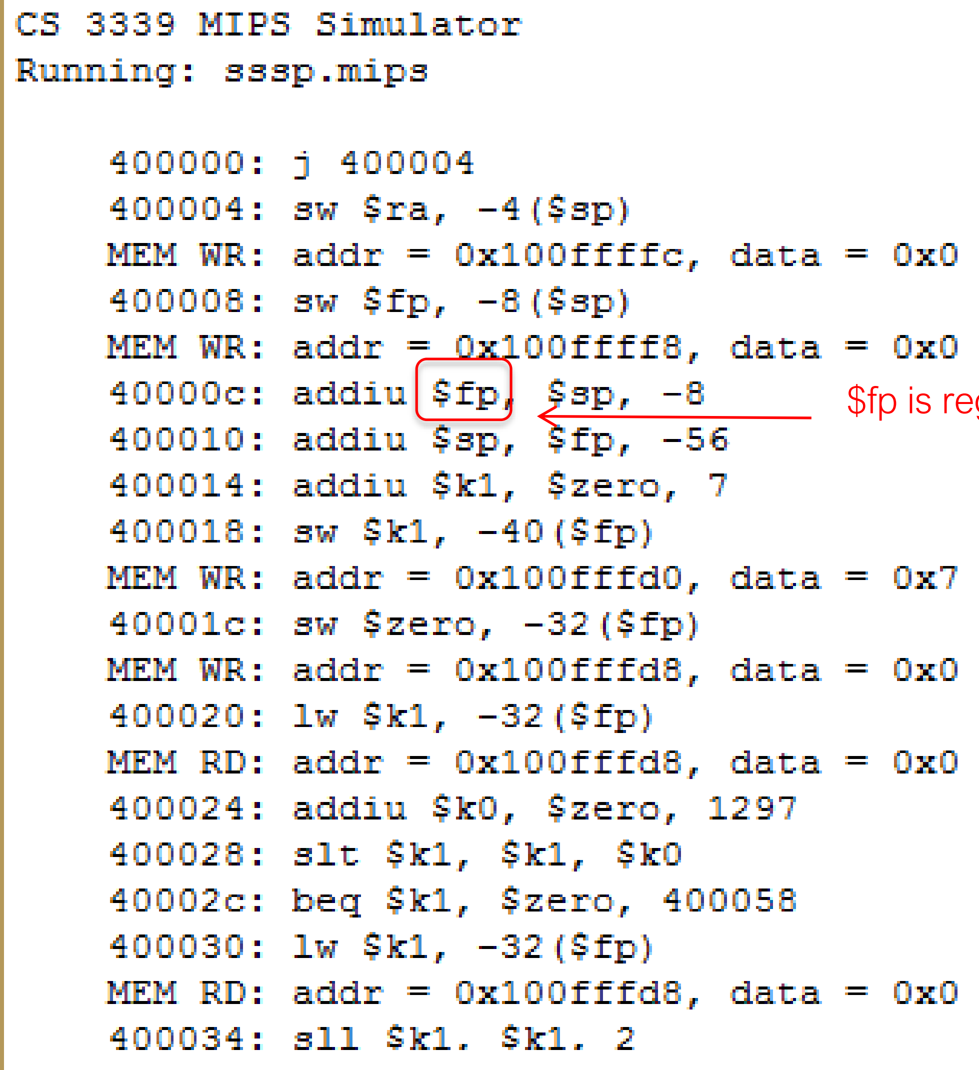
Your frame of reference is in decode (ID), MIPS ISA instructions provide all the information you need to complete the following:

I need to use rs so call `stats.registerSrc(rs);`
Is the value in that register ok or still “in-flight” from a prior instruction? i.e it has not reached wb.
If it is in-flight stats should insert the required number of bubbles to move it to wb.

Repeat for second input if applicable.

Also let stats know if current instruction will write a register using `stats.registerDest(rd);`

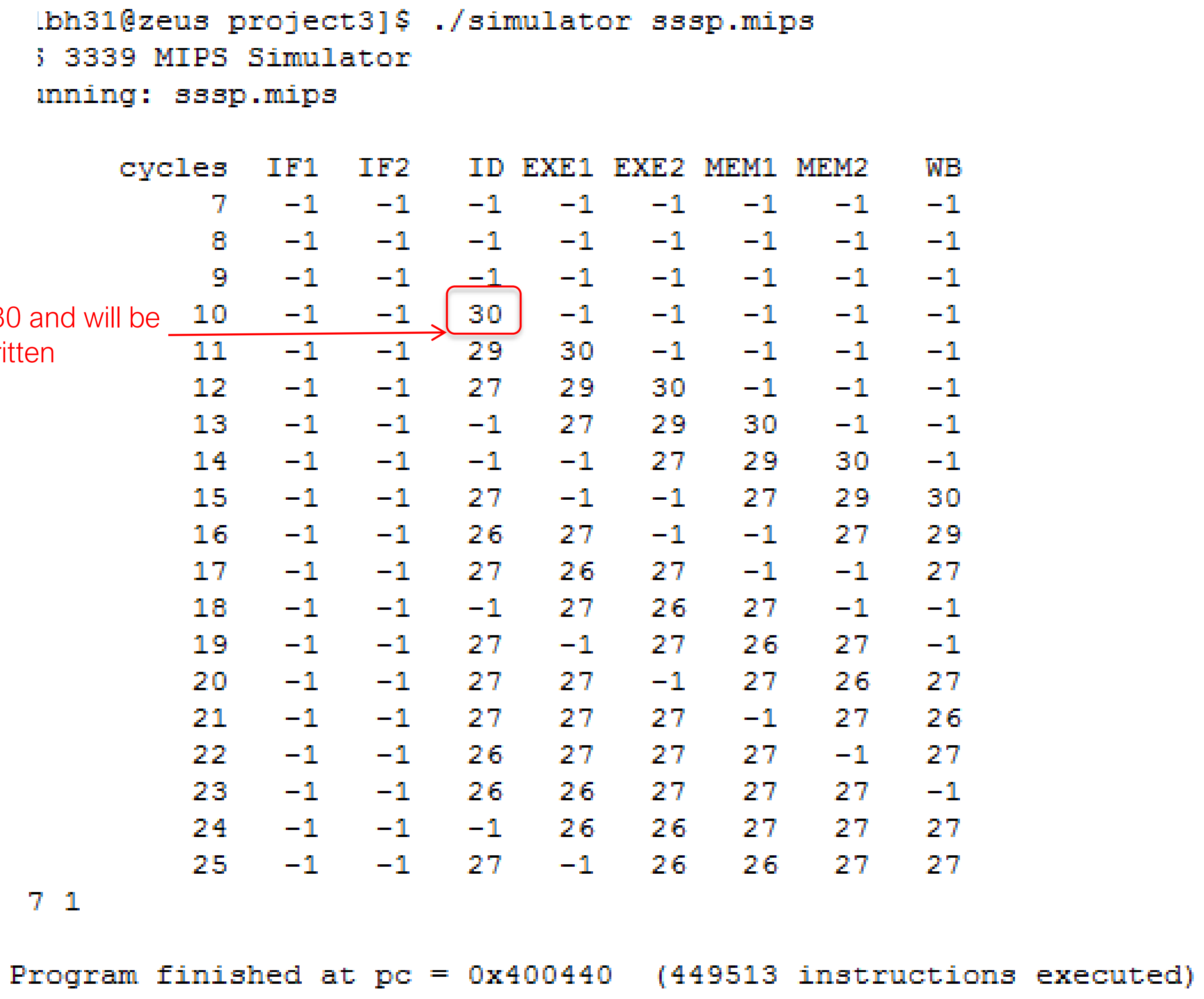
Dump Instr w/ Proj2, Print resultReg[]



```
CS 3339 MIPS Simulator
Running: sssp.mips

    400000: j 400004
    400004: sw $ra, -4($sp)
    MEM WR: addr = 0x100ffffc, data = 0x0
    400008: sw $fp, -8($sp)
    MEM WR: addr = 0x100ffff8, data = 0x0
    40000c: addiu $fp, $sp, -8
    400010: addiu $sp, $fp, -56
    400014: addiu $k1, $zero, 7
    400018: sw $k1, -40($fp)
    MEM WR: addr = 0x100fffd0, data = 0x7
    40001c: sw $zero, -32($fp)
    MEM WR: addr = 0x100fffd8, data = 0x0
    400020: lw $k1, -32($fp)
    MEM RD: addr = 0x100fffd8, data = 0x0
    400024: addiu $k0, $zero, 1297
    400028: slt $k1, $k1, $k0
    40002c: beq $k1, $zero, 400058
    400030: lw $k1, -32($fp)
    MEM RD: addr = 0x100fffd8, data = 0x0
    400034: sll $k1. $k1. 2
```

$fp is reg 30 and will be written



```
bh31@zeus project3]$ ./simulator sssp.mips
 3339 MIPS Simulator
nning: sssp.mips

        cycles   IF1   IF2    ID  EXE1  EXE2  MEM1  MEM2    WB
             7    -1    -1    -1    -1    -1    -1    -1    -1
             8    -1    -1    -1    -1    -1    -1    -1    -1
             9    -1    -1    -1    -1    -1    -1    -1    -1
            10    -1    -1    30    -1    -1    -1    -1    -1
            11    -1    -1    29    30    -1    -1    -1    -1
            12    -1    -1    27    29    30    -1    -1    -1
            13    -1    -1    -1    27    29    30    -1    -1
            14    -1    -1    -1    -1    27    29    30    -1
            15    -1    -1    27    -1    -1    27    29    30
            16    -1    -1    26    27    -1    -1    27    29
            17    -1    -1    27    26    27    -1    -1    27
            18    -1    -1    -1    27    26    27    -1    -1
            19    -1    -1    27    -1    27    26    27    -1
            20    -1    -1    27    27    -1    27    26    27
            21    -1    -1    27    27    27    -1    27    26
            22    -1    -1    26    27    27    27    -1    27
            23    -1    -1    26    26    27    27    27    -1
            24    -1    -1    -1    26    26    27    27    27
            25    -1    -1    27    -1    26    26    27    27
 7  1

Program finished at pc = 0x400440  (449513 instructions executed)
```

Partially complete – no bubbles or flushes…

Flush Count

The flush count is based on the configuration of the pipeline.   Specifically, the number of stages that contain "incorrect" instructions which have to be flushed due to the jump or branch taken.

For this pipeline configuration the count is ID - IF1 where ID and IF1 represent the pipestage number.   For example if we expanded to IF1, IF2, IF3 then each branch taken would require 3 flush cycles. This BTW is one reason really high pipeline counts don't work well - most code uses a lot of branch instructions with too many pipestages the inputs to the branch decision may be many clocks away and bunch of IFx stages may need to be flushed resulting in a large number of bubbles.

Clock, Flush, Bubble

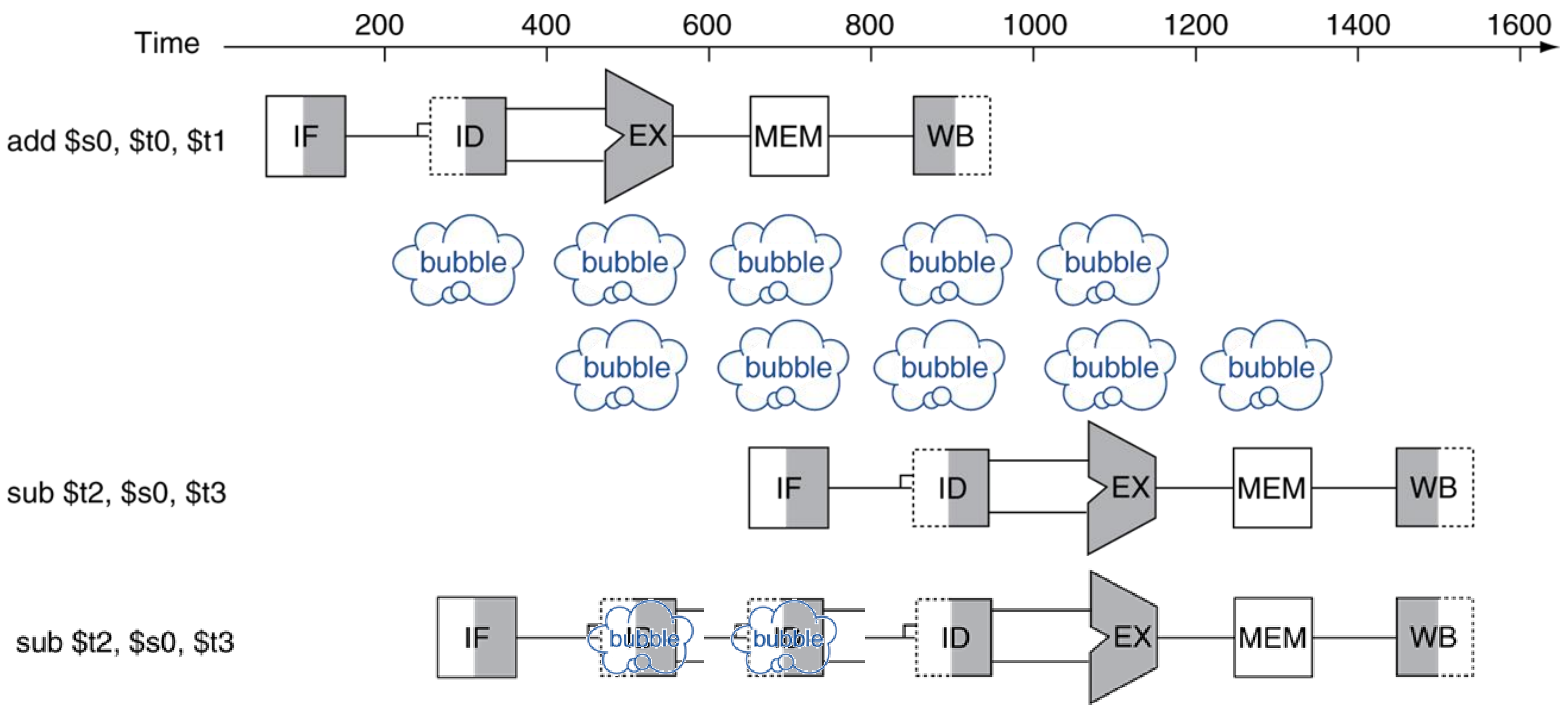
Clock advances the entire pipeline once it is clear that the current instruction in decode can proceed.

Flush also advances the entire pipeline, but for a different reason - to clear out incorrect instructions loaded in IFx. Therefore, the number of flushes required on a jump or branch taken equals the number of IF stages.

Bubbles advance only the pipeline stages after ID.   IFx and ID are ‘frozen’, while EXEx, MEMx, and WB continue. This allows any in-process register write(s) to complete so the register value will be ready by the time the current instruction needs it.

What this means for our simulation is that clock and flush methods are nearly identical, bubble is slightly different but very similar.

The 5 stage MIPS pipeline



Scripting and Submission

Writing short script files in Linux can save you a ton of time.

Good introduction to this here:

https://ryanstutorials.net/bash-scripting-tutorial/bash-script.php

We use scripts to automate grading which is one reason following the assignment instructions is critical. I have provided a test submission script in the assignment that you should use to verify your submitted tar file.

Files that don't untar and compile may received 0 points.