ELEN 511
Advanced Computer Architecture

Homework #2
Tomasulo Algorithm

Prof. Hoeseok Yang
Santa Clara University

Fall Quarter 2022

Homework #2 - Overview

- In this homework, you will implement Tomasulo algorithm from the given skeleton
  - With (Part 3) and without (Part 1 and 2) speculation

- Tasks – 15 + 2 points
  - Part 0: understand the given architecture – 2 points
    - Then, estimate the results (cycle by cycle)
  - Part 1: implement the basic algorithm – 5 points
    - Then, check the results
  - Part 2 (bonus credit): implement register renaming – 2 points
    - Based on the Part 1 implementation
  - Part 3: HW speculation – 4 points
    - Add Reorder Buffer (ROB) to Part 1 and extend the implementation for HW speculation
  - Report - 4 points
    - 2 points for Part 1 and 2
    - 2 points for Part 3

- Submission
  - Upload your report (in pdf) and source to Camino
  - Due: ~~Dec. 4th~~ → Dec. 11th 11:59pm (firm deadline – no extension will be allowed in any case)

ISA under Consideration

- For this assignment, suppose the following simple ISA
  - add (addition)
    - add rd, rs, rt // rd = rs + rt
  - sub (subtraction)
    - sub rd, rs, rt // rd = rs - rt
  - mul (multiplication)
    - mul rd, rs, rt // rd = rs * rt
  - div
    - div rd, rs, rt // rd = rs / rt
  - ld
    - ld rd, imm(rs) // rd = mem[rs+imm]
  - st
    - st rd, imm(rs) // mem[rs+imm] = rd
- See inst.h

Processor under Consideration

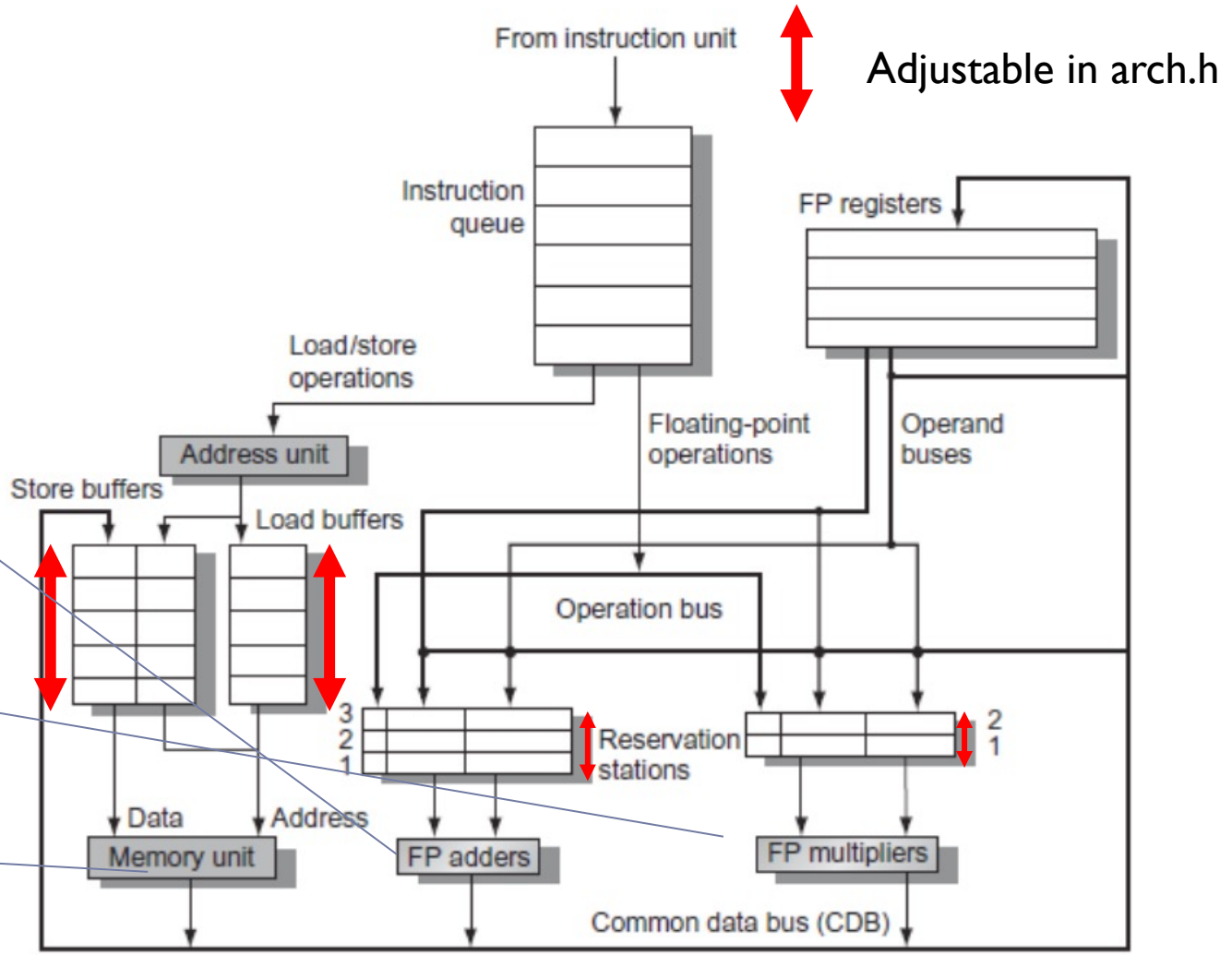
- For this assignment, suppose the following imaginary processor
  - 3 Functional Units
    - ADD (for add and sub)
    - MUL (for mul and div)
    - MEM (memory access unit)
  - 4 Reservation Stations or LD/ST Buffers
    - LD_BUF (2 entries) → connected to MEM
    - ST_BUF (2 entries) → connected to MEM
    - ADD_RS (3 entries) → connected to ADD
    - MUL_RS (2 entries) → connected to MUL
  - 16 32-bit registers
  - 1024 x 4 bytes memory
    - Word-addressed
      - i.e., mem(0) returns you first 4-bytes in your memory and mem(1) returns you the next 4-bytes

- See arch.h

Target Architecture

Adjustable in arch.h

Instr.	Exec. cycles
add	2
sub	2
mul	2
div	4
ld	1
st	1



Starting Point

- tar xvzf ./hw2.tar.gz

- Files
  - Makefile
    - Build script
  - inst.h/.c
    - Instruction information and utility functions
  - arch.h/.c
    - Architecture information and utility functions
  - tomasulo.c
    - Main body

Utility Functions

- You may use the following functions that have already been written in inst.h/.c and arch.h/.c
  - Instruction
    - init_inst(): initializing the instruction sequence
      - You may want to modify this when you try another sequence
    - print_inst(INST ins): print a single instruction "ins"
    - print_program(): print the test instruction sequence
  - Memory
    - init_mem(): initializing the memory
    - set_mem(int addr, int val): set mem[addr] = val
    - get_mem(int addr): get mem[addr]
  - Registers
    - init_regs(): initializing register file
    - print_regs(): print register status

Utility Functions (cont'd)

- You may use the following functions that have already been written in inst.h/.c and arch.h/.c
  - architectures
    - init_fu(): initializing the functional unit availabilities
    - init_rs(): initializing the reservation station entries
    - print_rs(); print the reservation station status
    - obtain_available_rs(enum rs_type t)
      - Return the RS id of the first available RS entry of rs_type t
      - If not available, return -1
    - get_rs(int id): return the pointer of the RS entry whose id is "id"
    - is_rs_active(): return whether the RS entry is busy or not
    - reset_rs_entry (RS * t): reset the RS entry that t points

Architecture Configuration (arch.h)

- (IMPORTANT) Note that the register number starts from 0

```
/*****************************************
   architecture definition
*****************************************/
/* num of RS or load/store buffer entries  */
enum rs_type {LD_BUF, ST_BUF, ADD_RS, MUL_RS};
#define NUM_LD_BUF 2
#define NUM_ST_BUF 2
#define NUM_ADD_RS 3
#define NUM_MUL_RS 2
#define NUM_RS_ENTRIES (NUM_LD_BUF+NUM_ST_BUF+NUM_ADD_RS+NUM_MUL_RS)

extern bool is_add_available;
extern bool is_mul_available;
extern bool is_mem_available;

#define MEM_SIZE 1024 /* memory size in word */
extern int mem_arr[MEM_SIZE];

/* registers */
#define NUM_REGS 16

/* execution unit latencies */
#define LAT_ADD 2 /* executed on ADD */
#define LAT_SUB 2 /* executed on ADD */
#define LAT_MUL 2 /* executed on MUL */
#define LAT_DIV 4 /* executed on MUL */
#define LAT_LD 1 /* executed on Memory Unit */
#define LAT_ST 1 /* executed on Memory Unit */
```

Data Structure for Instructions (inst.h)

- (IMPORTANT) Note that the instruction number starts from 1

```
#define NUM_OF_OP_TYPES 6
enum op_type {ADD, SUB, MUL, DIV, LD, ST};

/* data structure for an instruction */
typedef struct instruction {
  int num; /* number: starting from 1 */
  enum op_type op; /* operation type */
  int rd; /* destination register id */
  int rs; /* source regsiter id or base register for ld/st */
  int rt; /* target regsiter id or addr offset for ld/st */
} INST;


#define NUM_OF_INST 6
extern INST inst[NUM_OF_INST]; /* instruction array */

void init_inst();
void print_inst(INST ins);
void print_program();
```

Data Structure for Tomasulo Algorithm (arch.h)

- (IMPORTANT) Note that RS id starts from 1

```
typedef struct a_reg {
  int num; /* register id starting from 0 */
  int val; /* value */
  int Qi; /* the number of the RS entry that contains the operation whose
} REG;

extern REG regs[NUM_REGS];

/* data structure for an RS/LB/SB entry */
typedef struct reservation_station {
  int id;               /* RS id starting from 1 */
  bool is_busy;         /* is busy? */
  enum rs_type type;            /* 0:LD, 1:ST, 2:ADD, 3:MUL */
  enum op_type op;      /* ADD, SUB, MUL, DIV, LD, ST */
  int Qj, Qk;           /* ids of the RS entries */
  int Vj, Vk;           /* values */
  int A;                /* address for LD/ST */
  int exec_cycles;      /* remaining execution cycles */
  int result;           /* result */
  bool in_exec;         /* is in execution? */
  bool is_result_ready; /* is result ready? */
  int inst_num;         /* instruction number */
} RS;

extern RS rs_array[NUM_RS_ENTRIES];
```

Part 0: Understand the given architecture

- Read the given codes and understand the basic architecture and configurations
- Instruction sequence under test is defined in init_inst() in inst.c

```
void init_inst()
{
  inst[0].num=1; inst[0].op=LD; inst[0].rd=6; inst[0].rs=12; inst[1].rt=32; // ld r6,32(r12)
  inst[1].num=2; inst[1].op=LD; inst[1].rd=2; inst[1].rs=13; inst[1].rt=44; // ld r2,44(r13)
  inst[2].num=3; inst[2].op=MUL; inst[2].rd=0; inst[2].rs=2; inst[2].rt=4; // mul r0, r2, r4
  inst[3].num=4; inst[3].op=SUB; inst[3].rd=8; inst[3].rs=2; inst[3].rt=6; // sub r8, r2, r6
  inst[4].num=5; inst[4].op=DIV; inst[4].rd=10; inst[4].rs=0; inst[4].rt=6; // div r10, r0, r6
  inst[5].num=6; inst[5].op=ADD; inst[5].rd=11; inst[5].rs=0; inst[5].rt=6; // add r11, r0, r6
  return;
}
```

- Estimate how many cycles it takes in this architecture to complete this code and justify your estimation

Part 1: Implement the basic Tomasulo algorithm

- Complete the while loop in the main function
- Why do they appear in a reverse order?
  - Step 3: Write result
  - Step 2: Execution
  - Step 1: Issue
- For each of the three steps, refer to the following pages

```
/* simulation loop main */
while(!done){

  /* increment the cycle */
  cycle++;

  /*******************************
   *     Step III: Write result
   *******************************/
   for(i=0;i<NUM_RS_ENTRIES;i++) {
     // complete STEP III here
   }


  /*******************************
   *     Step II: Execute
   *******************************/
   for(i=0;i<NUM_RS_ENTRIES;i++) {
     // complete STEP II here
   }

  /*******************************
   *     Step I: Issue
   *******************************/

  /*  wait if no RS entry is available */
  if(num_issued_inst < NUM_OF_INST) {
    int cand_rs_id;
    if(inst[num_issued_inst].op==ADD) cand_rs_id = obtain_available_rs(ADD_RS);
    else if(inst[num_issued_inst].op==SUB) cand_rs_id = obtain_available_rs(ADD_RS);
    else if(inst[num_issued_inst].op==MUL) cand_rs_id = obtain_available_rs(MUL_RS);
    else if(inst[num_issued_inst].op==DIV) cand_rs_id = obtain_available_rs(MUL_RS);
    else if(inst[num_issued_inst].op==LD) cand_rs_id = obtain_available_rs(LD_BUF);
    else if(inst[num_issued_inst].op==ST) cand_rs_id = obtain_available_rs(ST_BUF);

    /* if there is an available RS entry */
    if(cand_rs_id!=-1) {
```

Part 1: Step 1) Issue

- Due to some slight differences in the ISA, following modifications need to be made
  - Step I implementation is given

Instruction state	Wait until	Action or bookkeeping

Issue
FP operation
	Station r empty	if (RegisterStat[rs].Qi≠0)
		{RS[r].Qj ← RegisterStat[rs].Qi}
		else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0};
		if (RegisterStat[rt].Qi≠0)
		{RS[r].Qk ← RegisterStat[rt].Qi
		else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};
		RS[r].Busy ← yes; RegisterStat[rd].Q ← r;

Load or store	Buffer r empty	if (RegisterStat[rs].Qi≠0) {RS[r].Qj ← RegisterStat[rs].Qi} else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0}; RS[r].A ← imm; RS[r].Busy ← yes;
Load only		RegisterStat[rt].Qi ← r;
Store only		if (RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rs].Qi} else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};

rd

rd

Part 1: Step 2) Execution

- For simplicity, forget about the memory access ordering

Execute FP operation	(RS[r].Qj = 0) and (RS[r].Qk = 0)	Compute result: operands are in Vj and Vk
Load/storestep 1	RS[r].Qj = 0 ~~& r is head of load-store queue~~	RS[r].A ← RS[r].Vj + RS[r].A;
Load step 2	Load step 1 complete	Read from Mem[RS[r].A]

- But, you do need to check the functional unit is available before you start execution

Part 1: Step 3) Write Result

Write result FP operation or load	Execution complete at r & CDB available	∀x(if (RegisterStat[x].Qi=r) {Regs[x] ← result; RegisterStat[x].Qi ← 0}); ∀x(if (RS[x].Qj=r) {RS[x].Vj ← result;RS[x].Qj ← 0}); ∀x(if (RS[x].Qk=r) {RS[x].Vk ← result;RS[x].Qk ← 0}); RS[r].Busy ← no;
Store	Execution complete at r & RS[r].Qk = 0	Mem[RS[r].A] ← RS[r].Vk; RS[r].Busy ← no;

Part 1: Report

- Your report must include
  - The overview of your implementation and
  - The results
    - Currently, at the end of each cycle it prints out the RS and register information

- You also have to try the following experiments
  1. Increase LD_LAT to 2 and test the same sequence
     - Report the result
  2. Define your own sequence and report the result with it
     - You have to use all six instruction types
     - At least three data dependencies

Output Example – Initial State

```
hyang8@ENGR-L-01613 tomasulo_sol % ./tomasulo
============== TEST INSTRUCTION SEQUENCE ===========
I#1     ld      r6,0(r12)
I#2     ld      r2,44(r13)
I#3     mul     r0,r2,r4
I#4     sub     r8,r2,r6
I#5     div     r10,r0,r6
I#6     add     r11,r0,r6
* CYCLE 0 (initial state)
=================================================================================================
RS_id   type    Busy    inst#   Op      Vj      Vk      Qj      Qj      A       Exec    Done
-------------------------------------------------------------------------------------------------
#1      LD      No      I#-1    NONE    -1      -1      #0      #0      -1      No      No
#2      LD      No      I#-1    NONE    -1      -1      #0      #0      -1      No      No
#3      ST      No      I#-1    NONE    -1      -1      #0      #0      -1      No      No
#4      ST      No      I#-1    NONE    -1      -1      #0      #0      -1      No      No
#5      ADD     No      I#-1    NONE    -1      -1      #0      #0      -1      No      No
#6      ADD     No      I#-1    NONE    -1      -1      #0      #0      -1      No      No
#7      ADD     No      I#-1    NONE    -1      -1      #0      #0      -1      No      No
#8      MUL     No      I#-1    NONE    -1      -1      #0      #0      -1      No      No
#9      MUL     No      I#-1    NONE    -1      -1      #0      #0      -1      No      No
=================================================================================================
=================================================================
Registers
-----------------------------------------------------------------
        r0      r1      r2      r3      r4      r5      r6      r7
-----------------------------------------------------------------
val     0       1       2       3       4       5       6       7
Qi      #0      #0      #0      #0      #0      #0      #0      #0
-----------------------------------------------------------------
        r8      r9      r10     r11     r12     r13     r14     r15
-----------------------------------------------------------------
val     8       9       10      11      12      13      14      15
Qi      #0      #0      #0      #0      #0      #0      #0      #0
-----------------------------------------------------------------
```

Output Example – Cycle 1

```
=================================================================
* CYCLE 1
=================================================================================================
RS_id   type    Busy    inst#   Op      Vj      Vk      Qj      Qj      A       Exec    Done
-------------------------------------------------------------------------------------------------
#1      LD      Yes     I#-1    ld      12      -1      #0      #0      0       No      No
#2      LD      No      I#-1    NONE    -1      -1      #0      #0      -1      No      No
#3      ST      No      I#-1    NONE    -1      -1      #0      #0      -1      No      No
#4      ST      No      I#-1    NONE    -1      -1      #0      #0      -1      No      No
#5      ADD     No      I#-1    NONE    -1      -1      #0      #0      -1      No      No
#6      ADD     No      I#-1    NONE    -1      -1      #0      #0      -1      No      No
#7      ADD     No      I#-1    NONE    -1      -1      #0      #0      -1      No      No
#8      MUL     No      I#-1    NONE    -1      -1      #0      #0      -1      No      No
#9      MUL     No      I#-1    NONE    -1      -1      #0      #0      -1      No      No
=================================================================================================
=================================================================
Registers
-----------------------------------------------------------------
        r0      r1      r2      r3      r4      r5      r6      r7
-----------------------------------------------------------------
val     0       1       2       3       4       5       6       7
Qi      #0      #0      #0      #0      #0      #0      #1      #0
-----------------------------------------------------------------
        r8      r9      r10     r11     r12     r13     r14     r15
-----------------------------------------------------------------
val     8       9       10      11      12      13      14      15
Qi      #0      #0      #0      #0      #0      #0      #0      #0
-----------------------------------------------------------------
=================================================================
```

Output Example – Cycle 2

```
* CYCLE 2
=================================================================================================
RS_id   type    Busy    inst#   Op      Vj      Vk      Qj      Qj      A       Exec    Done
-------------------------------------------------------------------------------------------------
#1      LD      Yes     I#-1    ld      12      -1      #0      #0      12      Yes     Yes
#2      LD      Yes     I#-1    ld      13      -1      #0      #0      44      No      No
#3      ST      No      I#-1    NONE    -1      -1      #0      #0      -1      No      No
#4      ST      No      I#-1    NONE    -1      -1      #0      #0      -1      No      No
#5      ADD     No      I#-1    NONE    -1      -1      #0      #0      -1      No      No
#6      ADD     No      I#-1    NONE    -1      -1      #0      #0      -1      No      No
#7      ADD     No      I#-1    NONE    -1      -1      #0      #0      -1      No      No
#8      MUL     No      I#-1    NONE    -1      -1      #0      #0      -1      No      No
#9      MUL     No      I#-1    NONE    -1      -1      #0      #0      -1      No      No
=================================================================================================
================================================================
Registers
----------------------------------------------------------------
        r0      r1      r2      r3      r4      r5      r6      r7
----------------------------------------------------------------
val     0       1       2       3       4       5       6       7
Qi      #0      #0      #2      #0      #0      #0      #1      #0
----------------------------------------------------------------
        r8      r9      r10     r11     r12     r13     r14     r15
----------------------------------------------------------------
val     8       9       10      11      12      13      14      15
Qi      #0      #0      #0      #0      #0      #0      #0      #0
----------------------------------------------------------------
================================================================
```

Output Example – Cycle 3

```
* CYCLE 3
=================================================================================================
RS_id   type    Busy    inst#   Op      Vj      Vk      Qj      Qj      A       Exec    Done
-------------------------------------------------------------------------------------------------
#1      LD      No      I#-1    NONE    -1      -1      #0      #0      -1      No      No
#2      LD      Yes     I#-1    ld      13      -1      #0      #0      57      Yes     Yes
#3      ST      No      I#-1    NONE    -1      -1      #0      #0      -1      No      No
#4      ST      No      I#-1    NONE    -1      -1      #0      #0      -1      No      No
#5      ADD     No      I#-1    NONE    -1      -1      #0      #0      -1      No      No
#6      ADD     No      I#-1    NONE    -1      -1      #0      #0      -1      No      No
#7      ADD     No      I#-1    NONE    -1      -1      #0      #0      -1      No      No
#8      MUL     Yes     I#-1    mul     -1      4       #2      #0      -1      No      No
#9      MUL     No      I#-1    NONE    -1      -1      #0      #0      -1      No      No
=================================================================================================
================================================================
Registers
----------------------------------------------------------------
        r0      r1      r2      r3      r4      r5      r6      r7
----------------------------------------------------------------
val     0       1       2       3       4       5       12      7
Qi      #8      #0      #2      #0      #0      #0      #0      #0
----------------------------------------------------------------
        r8      r9      r10     r11     r12     r13     r14     r15
----------------------------------------------------------------
val     8       9       10      11      12      13      14      15
Qi      #0      #0      #0      #0      #0      #0      #0      #0
----------------------------------------------------------------
================================================================
```

Part 2 (optional): Register renaming

- Current implementation does not perform register renaming
  - For instance, consider the following sequence

```
ld    r6, 32(r12)
ld    r2, 44(r13)
mul   r0, r2, r4
sub   r8, r0, r6
div   r0, r2, r6
add   r6, r0, r6
```

  - By renaming r0 in div, we may obtain a bigger degree of ILP

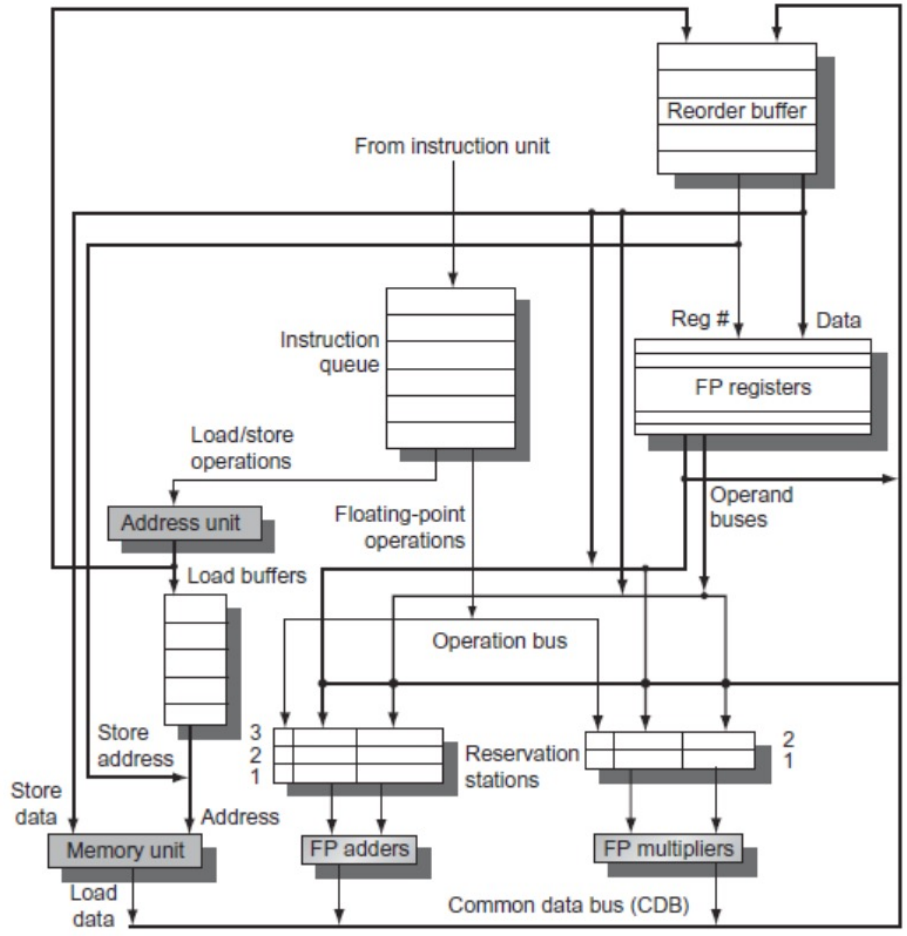
Part 2 (optional): Register renaming (cont'd)

- Properly extend the data structure to support register renaming
  - You may want to add additional information in the register data structure to keep track of the usage of registers
    - Which one is currently actively used or not

- Your report should include
  - the extended data structure,
  - the register renaming algorithm (that you devised),
  - and the result with register renaming

Part 3: HW Speculation

- Extend your implementation to support HW speculation



Part 3: HW Speculation (cont'd)

- (Important) Before you start Part 3, duplicate the entire folder and keep the separate workspace
  - The changes you will make in Part 3 would affect Part 1 and 2.

- Extend the instruction set to additionally support
  - addi rd, rs, imm // rd = rs + imm
  - bne rs, rt, imm // branch if rs!=rt

  - Properly modify inst.h and inst.c

- You need to define the data structure for Reorder Buffer (ROB)
  - Each ROB entry should have
    - Instruction type
    - Destination field (reigister number or mem addr.)
    - Value
    - Ready

  - Properly modify arch.h and arch.c

Part 3: HW Speculation (cont'd)

- Modify the tomasulo algorithm properly
  - Operand source is now ROB (not the result stored in RS)
  - Add a fourth step (Commit)
    - For each of the ROB entries, in case of mispredicted bne, discard all following entries
    - Assume that branch is always predicted to be taken

Part 3: HW Speculation (cont'd)

- Revised Tomasulo Algorithm

Status	Wait until	Action or bookkeeping
Issue all instructions		`if (RegisterStat[rs].Busy)/*in-flight instr. writes rs*/ {h ← RegisterStat[rs].Reorder; if (ROB[h].Ready)/* Instr completed already */ {RS[r].Vj ← ROB[h].Value; RS[r].Qj ← 0;} else {RS[r].Qj ← h;} /* wait for instruction */ } else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0;};`
	Reservation station (r) and ROB (b)	`RS[r].Busy ← yes; RS[r].Dest ← b; ROB[b].Instruction ← opcode; ROB[b].Dest ← rd; ROB[b].Ready ← no;`
FP operations and stores	both available	`if (RegisterStat[rt].Busy) /*in-flight instr writes rt*/ {h ← RegisterStat[rt].Reorder; if (ROB[h].Ready)/* Instr completed already */ {RS[r].Vk ← ROB[h].Value; RS[r].Qk ← 0;} else {RS[r].Qk ← h;} /* wait for instruction */ } else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0;};`
FP operations		`RegisterStat[rd].Reorder ← b; RegisterStat[rd].Busy ← yes; ROB[b].Dest ← rd;`
Loads		`RS[r].A ← imm; RegisterStat[rt].Reorder ← b; RegisterStat[rt].Busy ← yes; ROB[b].Dest ← rt;`
Stores		`RS[r].A ← imm;`

Part 3: HW Speculation (cont'd)

- Revised Tomasulo Algorithm

Execute FP op	`(RS[r].Qj == 0)` and `(RS[r].Qk == 0)`	Compute results—operands are in Vj and Vk
Load step 1	`(RS[r].Qj == 0)` and there are no stores earlier in the queue	`RS[r].A ← RS[r].Vj + RS[r].A;`
Load step 2	Load step 1 done and all stores earlier in ROB have different address	Read from `Mem[RS[r].A]`
Store	`(RS[r].Qj == 0)` and store at queue head	`ROB[h].Address ← RS[r].Vj + RS[r].A;`

Write result all but store	Execution done at *r* and CDB available	`b ← RS[r].Dest; RS[r].Busy ← no;`
		`∀x(if (RS[x].Qj==b) {RS[x].Vj ← result; RS[x].Qj ← 0});`
		`∀x(if (RS[x].Qk==b) {RS[x].Vk ← result; RS[x].Qk ← 0});`
		`ROB[b].Value ← result; ROB[b].Ready ← yes;`

Store	Execution done at *r* and `(RS[r].Qk == 0)`	`ROB[h].Value ← RS[r].Vk;`

Commit	Instruction is at the head of the ROB (entry h) and `ROB[h].ready == yes`	`d ← ROB[h].Dest; /* register dest, if exists */`
		`if (ROB[h].Instruction==Branch)`
		`{if (branch is mispredicted)`
		`{clear ROB[h], RegisterStat; fetch branch dest;};}`
		`else if (ROB[h].Instruction==Store)`
		`{Mem[ROB[h].Destination] ← ROB[h].Value;}`
		`else /* put the result in the register destination */`
		`{Regs[d] ← ROB[h].Value;};`
		`ROB[h].Busy ← no; /* free up ROB entry */`
		`/* free up dest register if no one else writing it */`
		`if (RegisterStat[d].Reorder==h) {RegisterStat[d].Busy ← no;};`

Part 3: Report

- Try the following sequence and report the result
  - Loop: ld r0, 0(r1)
  - mul r4, r0, r2
  - st r4, 0(r1)
  - addi r1, r1, 1
  - bne r1, r2, Loop

- The first prediction of bne will be wrong
  - Show what happens after the mispredction in your tomasulo architecture

- Your report must include
  - The overview of your implementation and
  - The results