CSc 252: Computer Organization

HW 5

due at 5pm, two days **before** the test

Turn in through GradeScope

Solutions

Policy Reminders

- You must turn in a single `PDF` to GradeScope. Other file formats will not be accepted. (Sorry, this is necessary for the sanity of the TAs!)
- You are allowed to work with other students on this homework, as we will not be grading it for correctness. However, **each student must turn in their own copy of the homework.**
- **Show your work for all problems.** While we won't be grading for correctness, you will not receive full credit unless you show your work.

  After all, showing your work is required on the test - and homeworks are intended to help you practice for the test!

Required Problems:

5.1(e-f), 5.2(d), 5.3(d), 5.4(f)

Allowable Instructions

When writing MIPS assembly, the only instructions that you are allowed to use (so far) are:

- `add, addi, sub, addu, addiu, subu`
- `beq, bne, j, jal, jr`
- `slt, slti`
- `and, andi, or, ori, nor, nori, xor, xori`
- `sll, srl, sra`
- `lw, lh, lb, sw, sh, sb`
- `la`
- `syscall`
- `mult, div, mfhi, mflo`

While MIPS has many other useful instructions (and the assembler recognizes many pseudo-instructions), do not use them! We want you to learn the **fundamentals** of how assembly language works - you can use fancy tricks after this class is over.

Problem 5.1 - Dependencies

In each part below, list all of the depdencies between instructions. I have numbered the instructions for convenience.

5.1(a)

```
add  $s2, $s5, $s4      # instruction 1
add  $s4, $s2, $s5      # instruction 2
sw   $s5, 100($s2)      # instruction 3
add  $s3, $s4, $s2      # instruction 4
```

5.1(b)

```
add  $s2, $s5, $s4      # instruction 1
add  $s5, $s2, $s5      # instruction 2
sw   $s5, 100($s2)      # instruction 3
lw   $s5, 104($s2)      # instruction 4
```

NOTE: Think carefully about what is read, and what is written, in lw/sw. Where are the dependencies?

5.1(c)

```
lw   $t0, 0($s0)        # instruction 1
sw   $t0, 0($s1)        # instruction 2
lw   $t1, 0($t0)        # instruction 3
beq  $t0, $t1, LABEL    # instruction 4
```

5.1(d)

```
add  $s2, $s0, $s1      # instruction 1
add  $s3, $s1, $s2      # instruction 2
add  $s4, $s2, $s3      # instruction 3
add  $s5, $s3, $s4      # instruction 4
```

5.1(e) - Turn in this one

```
la   $t0, foo           # instruction 1
lw   $t1, 0($t0)        # instruction 2
add  $t2, $s0, $s1      # instruction 3
sub  $t3, $t1, $t2      # instruction 4
```

Solution:

- The `rs` field in instruction 2 depends on instruction 1
- The `rs` field in instruction 4 depends on instruction 2
- The `rt` field in instruction 4 depends on instruction 3

5.1(f) - Turn in this one

```
add  $s2, $s0, $s1      # instruction 1
add  $s5, $s3, $s4      # instruction 2
sub  $s6, $s2, $s5      # instruction 3
addi $s7, $s6, 1        # instruction 4
```

Solution:

- The `rs` field in instruction 3 depends on instruction 1
- The `rt` field in instruction 3 depends on instruction 2
- The `rs` field in instruction 4 depends on instruction 3

Problem 5.2 - Converting MIPS to C

In the following problems, I will give you a snippet of MIPS assembly, which you will convert to C. Assume that all the MIPS code will use `tX` registers for temporary values (that is, values which are not given names in C), and that any `sX` registers represent variables which have names in C.

If the code includes any `.data` section, then include exactly equivalent C declarations in your code. Also, if the code uses any `sX` registers, give C declarations for matching variables. The names don't matter, but **giving the proper types is important.** The possible types are:

- `int` - MIPS words. Use this when you don't know anything else.
- `int*` - Pointers to MIPS words or arrays of words.
- `char*` - Pointers to bytes or strings.

To figure out types, you will have to use any number of clues - such as which variables are used in `la, lw, sw` instructions, how arrays are indexed, or what syscalls are used. **Use comments to clearly show what register is associated with each variable name.**

Likewise, if the assembly calls a function, give a declaration (not a definition!) for the function, including what you can figure out about the parameters and return type (if any).

Read the examples closely to see what I'm looking for.

5.2(a)

```
.global myFunc
myFunc:
    addiu  $sp, $sp, -24
    sw     $fp, 0($sp)
    sw     $ra, 4($sp)
    addiu  $fp, $sp, 20

    add    $t0, $a0, $a1
    add    $t1, $a2, $a3
    sub    $v0, $t0, $t1

    lw     $ra, 4($sp)
    lw     $fp, 0($sp)
    addiu  $sp, $sp, 24
    jr     $ra
```

5.2(b)

```
.data
str:
    .asciiz   ...something...
foo:
    .word     0

.text
    addi   $s0, $zero, 0
    la     $s1, str

LOOP:
    lb     $t0, 0($s1)
    beq    $t0, $zero, LOOP_END

    addi   $t1, $zero, 0x78       # 0x78 = 'x'
    bne    $t0, $t1, FALSE

    addi   $s0, $s0, 1

FALSE:
    addi   $s1, $s1, 1
    j      LOOP

LOOP_END:
    la     $t0, foo
    sw     $s0, 0($t0)
```

5.2(c)

```
addi   $a0, $zero, 1
jal    doThatThing
addi   $t0, $v0, $zero

addiu  $sp, $sp, -4
sw     $t0, 0($sp)

addi   $a0, $zero, 2
jal    doThatThing
addi   $t1, $v0, $zero

addiu  $sp, $sp, -4
sw     $t1, 0($sp)

addi   $a0, $zero, 3
jal    doThatThing
addi   $t2, $v0, $zero

lw     $t1, 0($sp)
lw     $t0, 4($sp)
addiu  $sp, $sp, 8

addi   $v0, $zero, 1
add    $a0, $t0, $t1
add    $a0, $a0, $t2
syscall

addi   $v0, $zero, 11
addi   $a0, $zero, 0xa
syscall
```

5.2(d) - Turn in this one

```
.global problemPartD
problemPartD:
    addiu  $sp, $sp, -24
    sw     $fp, 0($sp)
    sw     $ra, 4($sp)
    addiu  $fp, $sp, 20

    addiu  $sp, $sp, -4
    sw     $s0, 0($sp)

    addi   $s0, $zero, 0

LOOP:
    jal    getAValue
    beq    $v0, $zero, LOOP_END

    add    $s0, $s0, $v0
    j      LOOP

LOOP_END:
    add    $v0, $s0, $zero

    lw     $s0, 0($sp)
    addiu  $sp, $sp, 4

    lw     $ra, 4($sp)
    lw     $fp, 0($sp)
    addiu  $sp, $sp, 24
    jr     $ra
```

Solution:

```
int problemPartD()
{
    int sum = 0;

    while (1)
    {
        int t = getAValue();
        if (t == 0)
            break;

        sum += t;
    }

    return sum;
}
```

Problem 5.3 - Pipeline Diagrams

Using a drawing similar to slides 28 of deck 9, show the forwarding paths needed to execute each of the following sets of instructions. If a stall is necessary because of a memory operation, include the stall as in slide 45.

If you're doing this electronially, then you may use clip art that I've copied from the slides, and posted at `<class_website>/homeworks/pipeline_clip_art/` .

5.3(a)

```
add  $s5, $s3, $s4
add  $s4, $s5, $s6
add  $s2, $s3, $s4
```

5.3(b)

```
add  $s4, $s1, $s7
lw   $s6, 16($s4)
slt  $s2, $s1, $s5
add  $s5, $s6, $s6
```

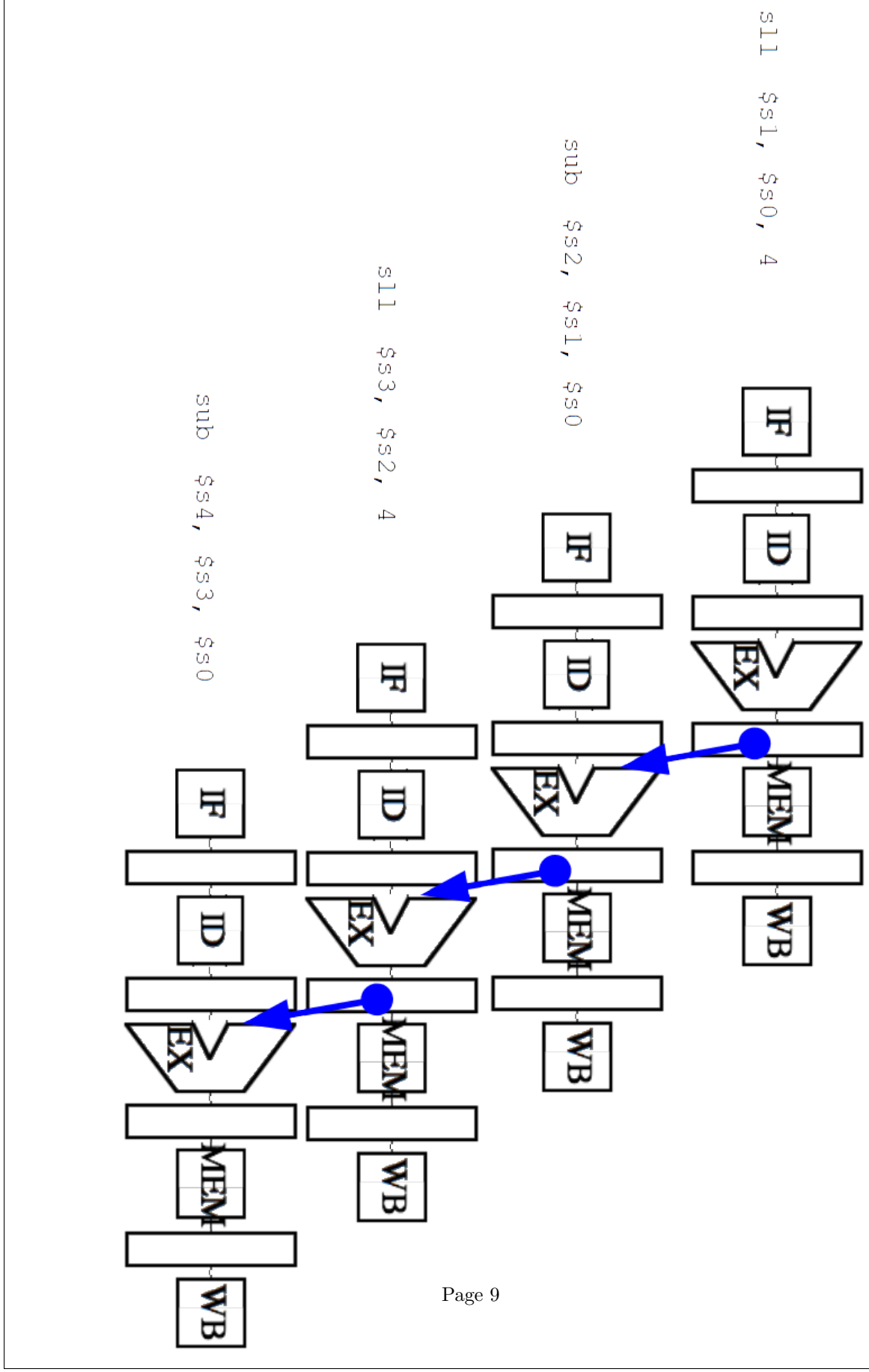
5.3(c)

```
add  $s4, $t0, $t1
add  $s5, $t2, $t3
add  $s6, $s4, $s5
slt  $s7, $zero, $s6
```

5.3(d) - Turn in this one

```
sll  $s1, $s0, 4
sub  $s2, $s1, $s0
sll  $s3, $s2, 4
sub  $s4, $s3, $s0
```

Solution:



Problem 5.4 - Pipelined Instructions

In each of these problems, I will present several instruction sequences, and then ask you what is going on in a certain clock cycle (assuming that the first instruction is fetched in cycle 1). You will consider this twice:

- There are no forwarding features; the only way that registers can be updated is in the WB phase. Stalls (NOP instructions) are inserted into the pipeline, by the control logic, if a data hazard exists.
- The CPU includes all of the forwarding features mentioned in the slides; stalls are only inserted for MEM-related delays (see slide 32).

For both situations, state what is happening in all 5 of the pipeline stages; if a stall has been inserted, then state clearly why this was, and where the NOP instruction is currently located in the pipeline.

If, in some cases, the code has run **past** the end of the given instructions (to execute other instructions not given), then clearly state that - but you don't have to give any more details.

5.4(a)

What is happening on the 3rd clock cycle of this instruction sequence?

```
add   $t2, $s1, $s3        # instruction 1
lw    $s1, 640($t2)        # instruction 2
sub   $s3, $t2, $s1        # instruction 3
sw    $s2, 72($sp)         # instruction 4
or    $t5, $s2, $t7        # instruction 5
```

5.4(b)

What is happening on the 7th clock cycle of the instruction sequence from part (a)?

5.4(c)

What is happening on the 10th clock cycle of this instruction sequence?

```
add   $s2, $s0, $s1        # instruction 1
add   $s3, $s0, $s1        # instruction 2
add   $s4, $s2, $s3        # instruction 3
add   $s5, $s2, $s3        # instruction 4
```

5.4(d)

What is happening on the 6th clock cycle of this instruction sequence?

```
lw    $s1, 640($t5)        # instruction 1
lw    $s2, 648($t5)        # instruction 2
lw    $s3, 656($t5)        # instruction 3
add   $t5, $s1, $s2        # instruction 4
```

5.4(e)

What is happening on the 7th clock cycle of this instruction sequence?

```
addi  $s1, $zero, 10       # instruction 1
lui   $at, 0x123           # instruction 2
ori   $s2, $at, 0x4567     # instruction 3
slt   $s3, $s1, $s2        # instruction 4
```

5.4(f) - Turn in this one

What is happening on the 7th clock cycle of this instruction sequence?

```
add  $s2, $s0, $s1      # instruction 1
add  $s5, $s3, $s4      # instruction 2
sub  $s6, $s2, $s5      # instruction 3
addi $s7, $s6, 1        # instruction 4
```

Solution:

NO FORWARDING

This sequence has three dependencies: instruction 3 is dependent on instruction 1 (through `$s2`), instruction 3 is also dependent on instruction 2 (through `$s5`), and also instruction 4 is dependent on instruction 3 (through `$s6`). Thus, we will need two stalls after instruction 2, and another two after instruction 3.

Thus, instruction 1 is in the WB phase in clock cycle 5, and instruction 2 is in that phase on clock cycle 6. Since instruction 3 must stall twice, it reads the registers in ID in clock cycle 6.

Thus, in clock cycle 7, instruction 3 is in the EX phase, and instruction 4 is in the ID phase (but stalled).

WITH FORWARDING

With forwarding, there are no stalls at all, because there are no `lw` instructions. Thus, all 4 instructions issue in order, without stalls.

By clock cycle 7, WB is executing instruction 3, MEM is executing instruction 4, and the rest of the phases are executing the instructions that follow.

EXAMPLES

Example: Problem 5.1(a)

- The `rs` field in instruction 2 depends on instruction 1
- The `rs` field in instruction 3 depends on instruction 1
- The `rt` field in instruction 4 depends on instruction 1
- The `rs` field in instruction 4 depends on instruction 2

Example: Problem 5.1(b)

- The `rs` field in instruction 2 depends on instruction 1
- The `rs` field in instruction 3 depends on instruction 1
- The `rt` field in instruction 3 depends on instruction 2
- The `rs` field in instruction 4 depends on instruction 1

Example: Problem 5.1(c)

```
lw   $t0, 0($s0)        # instruction 1
sw   $t0, 0($s1)        # instruction 2
lw   $t1, 0($t0)        # instruction 3
beq  $t0, $t1, LABEL    # instruction 4
```

- The `rt` field in instruction 2 depends on instruction 1
- The `rs` field in instruction 3 depends on instruction 1
- The `rs` field in instruction 4 depends on instruction 1
- The `rt` field in instruction 4 depends on instruction 3

Example: Problem 5.1(d)

```
add  $s2, $s0, $s1      # instruction 1
add  $s3, $s1, $s2      # instruction 2
add  $s4, $s2, $s3      # instruction 3
add  $s5, $s3, $s4      # instruction 4
```

- The `rt` field in instruction 2 depends on instruction 1
- The `rs` field in instruction 3 depends on instruction 1
- The `rt` field in instruction 3 depends on instruction 2
- The `rs` field in instruction 4 depends on instruction 2
- The `rt` field in instruction 4 depends on instruction 3

Example: Problem 5.2(a)

```
int myFunc(int a, int b, int c, int d)
{
    return (a+b)-(c+d);
}
```

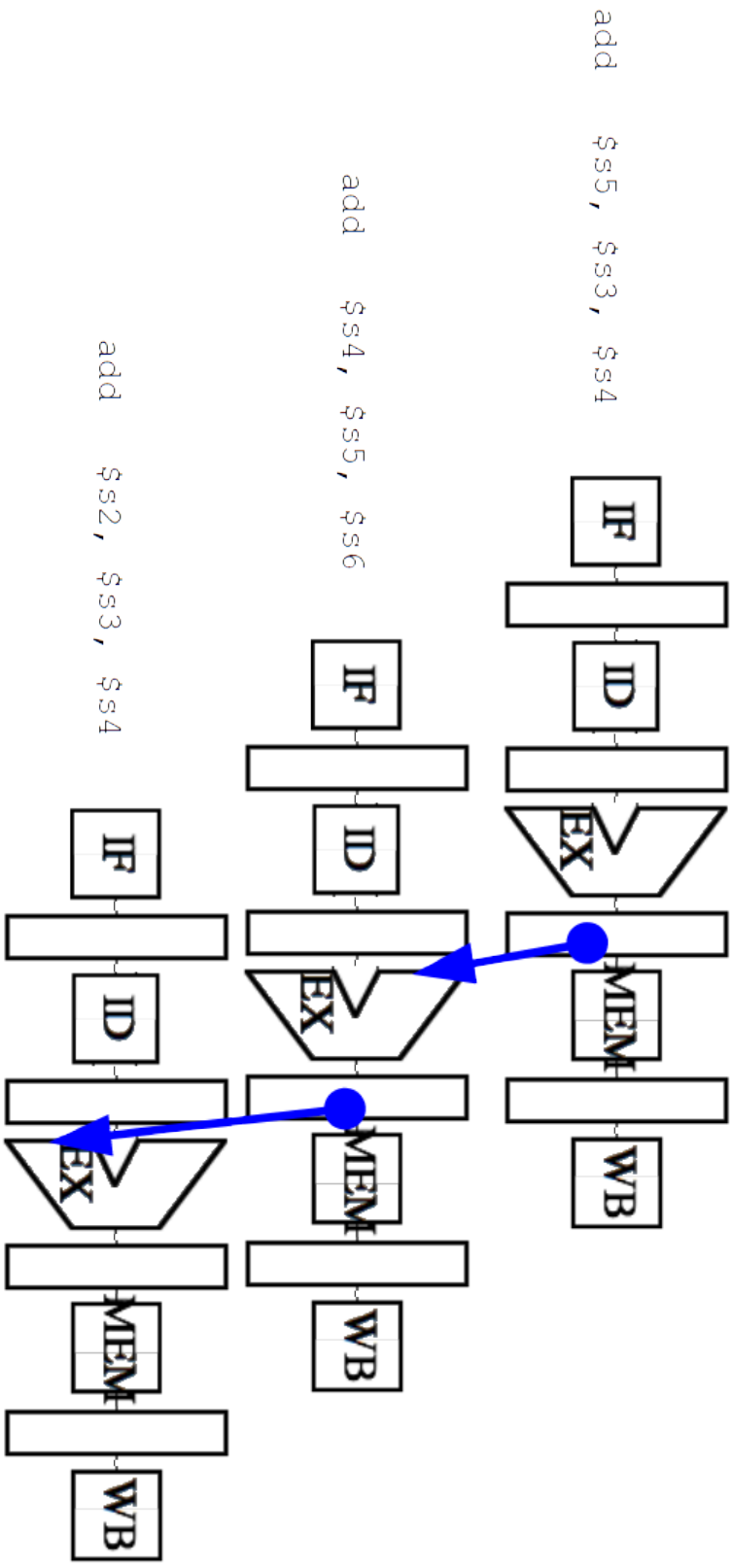
Example: Problem 5.2(b)

```
char *str = ...something...
int   foo = 0;

...

int   count = 0;
char *ptr = str;

while (*ptr != '\0')
{
    if (*ptr == 'x')
        count++;
    ptr++;
}

foo = count;
```

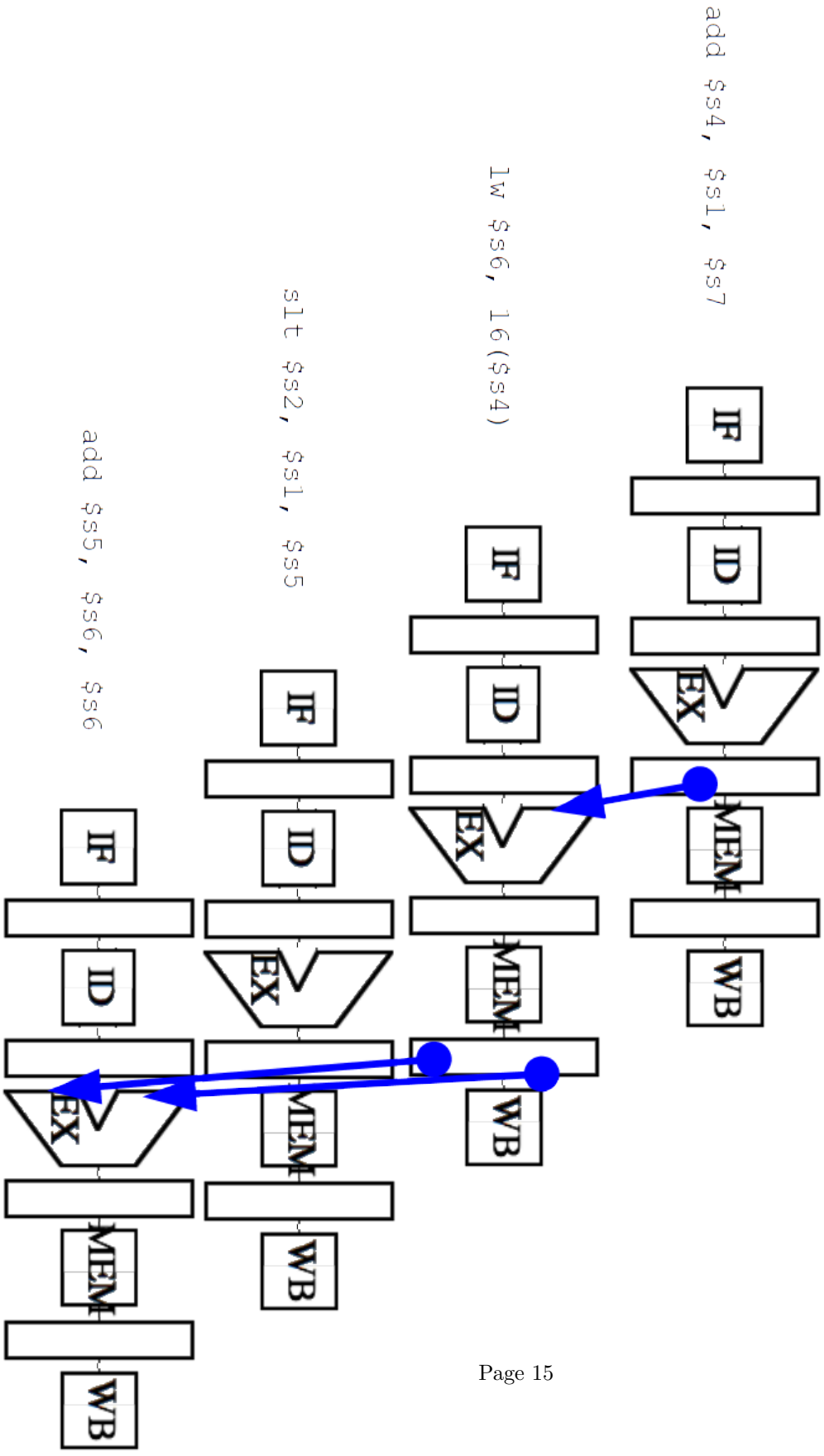
Example: Problem 5.2(c)

```
printf("%d\n", doThatThing(1) + doThatThing(2) + doThatThing(3));
```

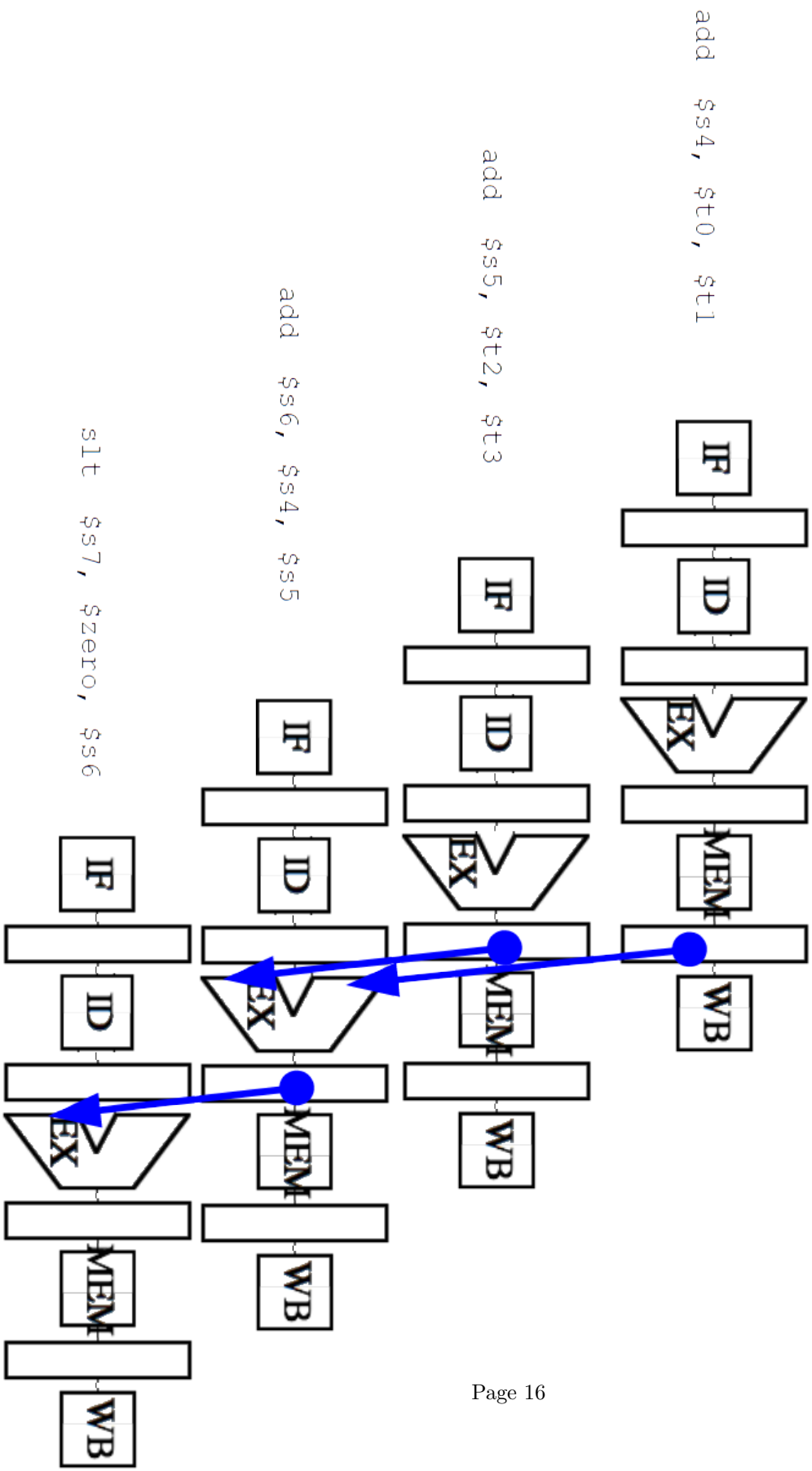
Example: Problem 5.3(a)



Example: Problem 5.3(b)



Example: Problem 5.3(c)



Example: Problem 5.4(a)

NO FORWARDING

Instruction 1 is in the EX stage, since it was fetched in cycle 1.

Instruction 2 is in the ID stage, since it was fetched in cycle 2; it will stall (not advance) this cycle, because it is waiting on the value of `$t2`, which will be written by instruction 1.

Instruction 3 is in the IF stage, it will stall because instruction 2 is going to stall.

The MEM and WB stages are idle, because the first instruction has not reached them yet.

WITH FORWARDING

In the third cycle, the instructions are in the same pipeline phases as the no-fowarding answer above; however, instructions 2 and 3 will **not** stall on the next cycle, because the forwarding hardware will be able to solve the problem that instruction 2 has.

Example: Problem 5.4(b)

NO FORWARDING

Instruction 1 has completed already; it executed the WB phase back in cycle 5.

Instruction 2 had to wait for instruction 1 to enter the WB phase; thus, it was stalled in the ID phase for a while; it finally completed ID in the same cycle when instruction 1 completed WB (cycle 5). It is thus now in the MEM phase.

(Because of this, the WB pipeline phase is currently executing a NOP.)

Instruction 3 is waiting for instruction 2 to update register `$s1`; thus, it is stalled in the ID phase.

(Because of this, the EX pipeline phase is currently executing a NOP.)

Instruction 4 is in the IF phase; it is stalled because instruction 3 is stalled.

WITH FORWARDING

Instruction 1 completed on clock cycle 5.

Because of the fowarding hardware, instruction 2 was not required to stall, even though it needed the value of `$t2` written by instruction 1. Thus, instruction 2 completed on clock cycle 6.

Instruction 3 had to insert one stall cycle, because it was waiting for the result of a lw instruction (writing to `$s1`). Thus, the WB pipeline phase is currently executing a NOP.

However, instruction 3 only waited a single cycle, and so it is now executing the MEM phase.

Instructions 4 and 5 have some depdendencies on instructions before them, but none are waiting on a lw - so neither stall. Therefore, instruction 4 is in the EX phase, and instruction 5 is in the ID phase.

The IF phase is executing some **other** instruction (instruction 6), not listed in this problem.

Example: Problem 5.4(c)

NO FORWARDING

First, we note dependencies: instructions 1 and 2 are independent; they will not stall. However, instructions 3 and 4 both wait on **both** of the results from instructions 1 and 2, and thus will stall until 2 is in WB.

Thus, instruction 1 was in IF in clock cycle 1, and in WB in clock cycle 5; instruction 2 was in IF in clock cycle 2, and in WB in clock cycle 6.

Instruction 3 stalled in the ID phase until instruction 2 was in WB; thus, it did not enter EX until clock cycle 7, and was in WB in clock cycle 9.

Instruction 4 stalled behind instruction 3. While it had its own dependencies (on instructions 1 and 2), both were completed by the time that it reached the ID phase (clock cycle 7), and so it did not add any stalls of its own. Thus, it was in the WB phase in clock cycle 10.

Thus, in clock cycle 10, the ID through MEM phases are all executing instructions not given in this problem.

WITH FORWARDING

None of these instructions are lw instructions; therefore, there is no reason to insert any stalls. Therefore, instruction 1 was in IF in clock cycle 1 and completed in clock cycle 5; instruction 4 was in IF in clock cycle 4 and completed in clock cycle 8.

By clock cycle 10, the entire processor is doing other instructions, not given in this problem.

Example: Problem 5.4(d)

NO FORWARDING

There are no dependencies in this sequence, except that instruction 4 is dependent on instructions 1 and 2. Thus, instructions 1,2,3 all are fetched and execute without stalls.

Thus, by clock cycle 6, instruction 1 has finished, instruction 2 is in the WB phase, and instruction 3 is in the MEM phase.

Instruction 4 had stalled (meaning that there is a NOP in the EX phase), however, on clock cycle 6, it can execute the ID phase (because its last dependency - instruction 2 - is in the WB phase). Thus, instruction 4 will complete the ID phase this cycle.

The IF phase is executing a 5th instruction, which is not listed in this problem.

WITH FORWARDING

With forwarding, there are no stalls at all; while there are lw instructions, we never use the result from a lw in the next instruction. Thus, all 4 instructions issue in order, without stalls.

By clock cycle 6, WB is executing instruction 2, MEM is executing instruction 3, EX is executing instruction 4, and ID and IF are both executing other instructions, which are not in this problem.

5.4(e)

NO FORWARDING

This sequence has two dependencies: instruction 3 is dependent on instruction 2 (through `$at`), and instruction 4 is dependent on instruction 3 (through `$s2`). Thus, we will need two stalls after the `lui`, and another two after the `ori`.

Thus, instruction 1 is in the WB phase in clock cycle 5, and instruction 2 is in that phase on clock cycle 6. Since `ori` must stall twice, `ori` reads the registers in ID in clock cycle 6.

Thus, in clock cycle 7, instruction 3 is in the EX phase, and instruction 4 is in the ID phase (but stalled).

WITH FORWARDING

With forwarding, there are no stalls at all, because there are no `lw` instructions. Thus, all 4 instructions issue in order, without stalls.

By clock cycle 7, WB is executing instruction 3, MEM is executing instruction 4, and the rest of the phases are executing the instructions that follow.