

Computer Architectures 02GOLOV	Delivery date: October,9th2024
Laboratory 1	Expected delivery of lab_01.zip including: - lab_01.pdf (fill and export this file to pdf)

This first lab is very introductory. Through it, you are expected to learn how to use WinMIPS and do the assigned task.

- 1) The first thing to do is open the winmipstut.docx file and follow the WinMIPS documentation. Through it, you are expected to learn the basics of using WinMIPS so that you are ready to take on the next assignment.
- 2) Consider a MIPS architecture with the following characteristics:
 - Integer ALU: 1 clock cycle
 - Data memory: 1 clock cycle
 - FP arithmetic unit: pipelined, 2 clock cycles
 - FP multiplier unit: pipelined, 8 clock cycles
 - FP divider unit: not pipelined, 10 clock cycles

Given the following piece of code, run it on WinMIPS and see what it does. For each instruction, add a comment to describe what the instruction does using the same style of commenting as the example. Next, write down the number of clock cycles required to execute the first iteration of the following code. Since this is a pipelined processor, the number of clock cycles must be calculated as the number of clock cycles required in addition to the previous instruction.

; ***** MIPS64 *****

.data	comments	Clock cycles
v1: .byte 1, 2, 3, 4, 5		
v2: .byte 1, 2, 3, 4, 5		
v3: .space 5		
v4: .space 5		
.text		
daddui r5,r0,0	$r5 \leftarrow \text{pointer}(i)$	5
daddui r6,r0,5	$r6 \leftarrow 5$	1
cycle: lb r1,v1(r5)	$r1 \leftarrow v1[i] = 1$	1
lb r2,v2(r5)	$R2 \leftarrow v2[i] = 1$	1
daddu r3,r1,r2	$R3 \leftarrow r1 + r2 = 1+1 = 2$	2
dsub r4,r1,r2	$R4 \leftarrow r1 - r2 = 1-1 = 0$	1
sb r3,v3(r5)	$V3[i] \leftarrow R3$	1
sb r4,v4(r5)	$V4[i] \leftarrow R4$	1
daddui r5,r5,1	$R5 \leftarrow R5 + 1$ (increment of i)	1
daddi r6,r6,-1	$R6 \leftarrow R6 + (-1)$	1
bnez r6,cycle	If R6 != 0: goto cycle	2
halt	return	1
Total:		18

Now, translate the above code in your preferred high-level language (i.e. C, Java, Python, C++, Javascript).

; ***** Used language

Python:

```
#lists
V1 = [1, 2, 3, 4, 5]
V2 = [1, 2, 3, 4, 5]
V3 = [0, 0, 0, 0, 0]
V4 = [0, 0, 0, 0, 0]
```

```
i = 0 #r5 = i
r6 = 5
```

```
#while cycle
while r6 != 0:
    r1 = V1[i]
    r2 = V2[i]
    r3 = r1+r2
    r4 = r1 - r2
    V3[i] = r3
    V4[i] = r4
    i += 1
    r6 -= 1
```

return #halt, not necessary

3) Now, repeat the same operation done previously with this piece of code:

; ***** MIPS64 *****

.data	comments	Clock cycles
v1: .double 1, 2, 3, 4, 5		
v2: .double 5, 4, 3, 2, 1		
v3: .double 6, 7, 8, 9, 10		
v4: .double 10, 9, 8, 7, 6		
v5: .space 40		
v6: .space 40		
.text		
daddui r1,r0,0	$R1 \leftarrow \text{pointer}(i)$	5
daddui r2,r0,5	$R2 \leftarrow R0 + 5 = 0 + 5 = 5$	1
cycle:l.d f1, v1(r1)	$F1 \leftarrow V1[i] = 1.0 \text{ (floating point)}$	1

l.d f2, v2(r1)	$F2 \leftarrow V2[i] = 5.0$	1
l.d f3, v3(r1)	$F3 \leftarrow V3[i] = 6.0$	1
l.d f4, v4(r1)	$F4 \leftarrow V4[i] = 10.0$	1
mul.d f5, f1, f2	$F5 \leftarrow F1 * F2 = 1.0 * 5.0 = 5.0$	8
mul.d f6, f3, f4	$F6 \leftarrow F3 * F4 = 6.0 * 10.0 = 60.0$	1
s.d f5, v5(r1)	$V5[i] = F5 = 5.0$	1
s.d f6, v6(r1)	$V6[i] = F6 = 60.0$	1
daddui r1,r1,8	$R5 \leftarrow R5+8 = 8$ (increment of i)	1
daddi r2,r2,-1	$R \leftarrow R2-1 = 4$	1
bnez r2,cycle	If R2 != 0: goto cycle	2
halt	return	1
Total:		26

Now, translate the above code in your preferred high-level language (i.e. C, Java, Python, C++, Javascript).

; ***** Used language

Python:

```

#lists
V1 = [1, 2, 3, 4, 5]
V2 = [5, 4, 3, 2, 1]
V3 = [6, 7, 8, 9, 10]
V4 = [10, 9, 8, 7, 6]
V5 = V6 = [0] * 40

i = 0 #r1 = i
r2 = 5

#while cycle
while r2 !=0:
    f1 = V1[i] #these are float variables
    f2 = V2[i]
    f3 = V3[i]
    f4 = V4[i]
    f5 = f1 * f2
    f6 = f3 * f4
    V5[i] = f5
    V6[i] = f6
    i+=8
    r2-=1

return #halt, not necessary
  
```

Relative to the above code, how much would the performance of the software increase if you could improve the functional unit related to the floating point multiplication of 4 clock cycles?

After the optimization, the floating-point multiplier will have a latency of 4 rather than 8 clock cycles.

I checked on the simulator and the first floating point multiplication (mul.d f5, f1, f2) is now completed in 4 clock cycles rather than 8!

Now, with a latency of 8, the program is executed in 106 clock cycles. After the enhancement, the program is executed in 86 clock cycles.

Using Amdahl's law:

- With a latency of 8 multiplications take $(8+1) * 5 = 45$ cycles [because we have 2 mul.d instructions per cycle, for 5 cycles]
- $p = 45 / 106 = 0.42$
- With a latency of 4 multiplications take $(4+1)*5 = 25$ cycles
- $s = 45 / 25 = 1.8$
- Overall speedup = $1/((1-p) + p/s) = 1.229 = 1.23$

Appendix: winMIPS64 Instruction Set

WinMIPS64

The following assembler directives are supported

.data - start of data segment
 .text - start of code segment
 .code - start of code segment (same as .text)
 .org <n> - start address
 .space<n> - leave n empty bytes
 .ascii<s> - enters zero terminated ascii string
 .asciiz<s> - enter ascii string
 .align<n> - align to n-byte boundary
 .word<n1>,<n2>.. - enters word(s) of data (64-bits)
 .byte<n1>,<n2>.. - enter bytes
 .word32 <n1>,<n2>.. - enters 32 bit number(s)
 .word16 <n1>,<n2>.. - enters 16 bit number(s)
 .double<n1>,<n2>.. - enters floating-point number(s)

where <n> denotes a number like 24, <s> denotes a string like "fred", and
 <n1>,<n2>.. denotes numbers separated by commas.

The following instructions are supported

lb - load byte
 lbu - load byte unsigned
 sb - store byte
 lh - load 16-bit half-word
 lhu - load 16-bit half word unsigned
 sh - store 16-bit half-word
 lw - load 32-bit word
 lwu - load 32-bit word unsigned
 sw - store 32-bit word
 ld - load 64-bit double-word
 sd - store 64-bit double-word
 ld - load 64-bit floating-point
 sd - store 64-bit floating-point
 halt - stops the program

daddi - add immediate
 daddui - add immediate unsigned

andi - logical and immediate
 ori - logical or immediate
 xori - exclusive or immediate
 lui - load upper half of register immediate
 slti - set if less than or equal immediate
 sltiu - set if less than or equal immediate unsigned
 beq - branch if pair of registers are equal
 bne - branch if pair of registers are not equal
 beqz - branch if register is equal to zero
 bnez - branch if register is not equal to zero
 j - jump to address
 jr - jump to address in register
 jal - jump and link to address (call subroutine)
 jalr - jump and link to address in register (call subroutine)
 dsll - shift left logical
 dsrl - shift right logical
 dsra - shift right arithmetic
 dsllv - shift left logical by variable amount
 dsrlv - shift right logical by variable amount
 dsrav - shift right arithmetic by variable amount
 movz - move if register equals zero
 movn - move if register not equal to zero
 nop - no operation
 and - logical and
 or - logical or
 xor - logical xor
 slt - set if less than
 sltu - set if less than unsigned
 dadd - add integers
 daddu - add integers unsigned
 dsub - subtract integers
 dsubu - subtract integers unsigned
 add.d - add floating-point
 sub.d - subtract floating-point

mul.d - multiply floating-point
div.d - divide floating-point
mov.d - move floating-point
cvt.d.l - convert 64-bit integer to a double FP format
cvt.l.d - convert double FP to a 64-bit integer format
c.lt.d - set FP flag if less than
c.le.d - set FP flag if less than or equal to

c.eq.d - set FP flag if equal to
bc1f - branch to address if FP flag is FALSE
bc1t - branch to address if FP flag is TRUE
mtc1 - move data from integer register to FP register
mfc1 - move data from FP register to integer register