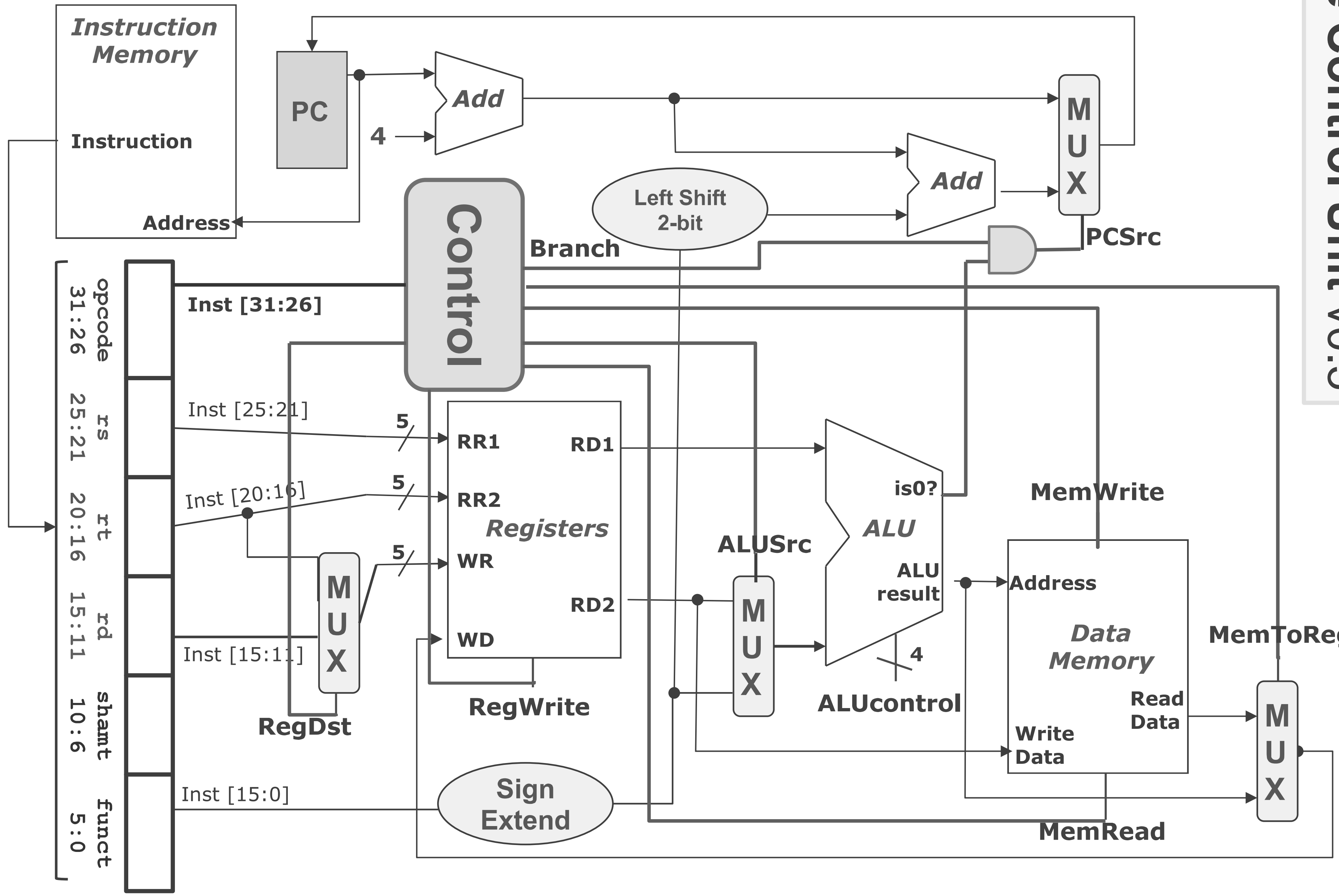
The Control Unit v0.5



6.1 Flip-flop Characteristic Tables

- Each type of flip-flop has its own behaviour, shown by its characteristic table.

J	*K*	*Q(t+1)*	Comments
0	0	*Q(t)*	No change
0	1	0	Reset
1	0	1	Set
1	1	*Q(t)'*	Toggle

S	*R*	*Q(t+1)*	Comments
0	0	*Q(t)*	No change
0	1	0	Reset
1	0	1	Set
1	1	?	Unpredictable

D	*Q(t+1)*	
0	0	Reset
1	1	Set

T	*Q(t+1)*	
0	*Q(t)*	No change
1	*Q(t)'*	Toggle

6.3 Flip-flop Excitation Tables (1/2)

- *Excitation tables*: given the required transition from present state to next state, determine the flip-flop input(s).

Q	Q^+	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

JK Flip-flop

Q	Q^+	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

SR Flip-flop

Q	Q^+	D
0	0	0
0	1	1
1	0	0
1	1	1

D Flip-flop

Q	Q^+	T
0	0	0
0	1	1
1	0	1
1	1	0

T Flip-flop

MIPS Reference Data

①

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	`add`	R	R[rd] = R[rs] + R[rt]	(1)	$0 / 20_{hex}$
Add Immediate	`addi`	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	`addiu`	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	`addu`	R	R[rd] = R[rs] + R[rt]		$0 / 21_{hex}$
And	`and`	R	R[rd] = R[rs] & R[rt]		$0 / 24_{hex}$
And Immediate	`andi`	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	`beq`	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	`bne`	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	`j`	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	`jal`	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	`jr`	R	PC=R[rs]		$0 / 08_{hex}$
Load Byte Unsigned	`lbu`	I	R[rt]={24'b0,M[R[rs] +SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	`lhu`	I	R[rt]={16'b0,M[R[rs] +SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	`ll`	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	`lui`	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	`lw`	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	`nor`	R	R[rd] = ~ (R[rs] | R[rt])		$0 / 27_{hex}$
Or	`or`	R	R[rd] = R[rs] | R[rt]		$0 / 25_{hex}$
Or Immediate	`ori`	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	`slt`	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		$0 / 2a_{hex}$
Set Less Than Imm.	`slti`	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	`sltiu`	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b_{hex}
Set Less Than Unsig.	`sltu`	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	$0 / 2b_{hex}$
Shift Left Logical	`sll`	R	R[rd] = R[rt] << shamt		$0 / 00_{hex}$
Shift Right Logical	`srl`	R	R[rd] = R[rt] >> shamt		$0 / 02_{hex}$
Store Byte	`sb`	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Conditional	`sc`	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38_{hex}
Store Halfword	`sh`	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	`sw`	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	`sub`	R	R[rd] = R[rs] - R[rt]	(1)	$0 / 22_{hex}$
Subtract Unsigned	`subu`	R	R[rd] = R[rs] - R[rt]		$0 / 23_{hex}$

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

BASIC INSTRUCTION FORMATS

R	opcode	rs	rt	rd	shamt	funct
	31 26	25 21	20 16	15 11	10 6	5 0
I	opcode	rs	rt	immediate		
	31 26	25 21	20 16	15 0		
J	opcode	address				
	31 26	25 0				

ARITHMETIC CORE INSTRUCTION SET

②

NAME, MNEMONIC		FOR-MAT	OPERATION		OPCODE / FMT /FT / FUNCT (Hex)
Branch On FP True	`bc1t`	FI	if(FPcond)PC=PC+4+BranchAddr	(4)	11/8/1/--
Branch On FP False	`bc1f`	FI	if(!FPcond)PC=PC+4+BranchAddr	(4)	11/8/0/--
Divide	`div`	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]		0/--/--/1a
Divide Unsigned	`divu`	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	(6)	0/--/--/1b
FP Add Single	`add.s`	FR	F[fd]= F[fs] + F[ft]		11/10/--/0
FP Add Double	`add.d`	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}		11/11/--/0
FP Compare Single	`c.`*x*`.s*`	FR	FPcond = (F[fs] *op* F[ft]) ? 1 : 0		11/10/--/*y*
FP Compare Double	`c.`*x*`.d*`	FR	FPcond = ({F[fs],F[fs+1]} *op* {F[ft],F[ft+1]}) ? 1 : 0		11/11/--/*y*
* (*x* is `eq`, `lt`, or `le`) (*op* is ==, <, or <=) (*y* is 32, 3c, or 3e)					
FP Divide Single	`div.s`	FR	F[fd] = F[fs] / F[ft]		11/10/--/3
FP Divide Double	`div.d`	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}		11/11/--/3
FP Multiply Single	`mul.s`	FR	F[fd] = F[fs] * F[ft]		11/10/--/2
FP Multiply Double	`mul.d`	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}		11/11/--/2
FP Subtract Single	`sub.s`	FR	F[fd]=F[fs] - F[ft]		11/10/--/1
FP Subtract Double	`sub.d`	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}		11/11/--/1
Load FP Single	`lwc1`	I	F[rt]=M[R[rs]+SignExtImm]	(2)	31/--/--/--
Load FP Double	`ldc1`	I	F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	(2)	35/--/--/--
Move From Hi	`mfhi`	R	R[rd] = Hi		0 /--/--/10
Move From Lo	`mflo`	R	R[rd] = Lo		0 /--/--/12
Move From Control	`mfc0`	R	R[rd] = CR[rs]		10 /0/--/0
Multiply	`mult`	R	{Hi,Lo} = R[rs] * R[rt]		0/--/--/18
Multiply Unsigned	`multu`	R	{Hi,Lo} = R[rs] * R[rt]	(6)	0/--/--/19
Shift Right Arith.	`sra`	R	R[rd] = R[rt] >>> shamt		0/--/--/3
Store FP Single	`swc1`	I	M[R[rs]+SignExtImm] = F[rt]	(2)	39/--/--/--
Store FP Double	`sdc1`	I	M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+4] = F[rt+1]	(2)	3d/--/--/--

FLOATING-POINT INSTRUCTION FORMATS

FR	opcode	fmt	ft	fs	fd	funct
	31 26	25 21	20 16	15 11	10 6	5 0
FI	opcode	fmt	ft	immediate		
	31 26	25 21	20 16	15 0		

PSEUDOINSTRUCTION SET

NAME	MNEMONIC	OPERATION
Branch Less Than	`blt`	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	`bgt`	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	`ble`	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	`bge`	if(R[rs]>=R[rt]) PC = Label
Load Immediate	`li`	R[rd] = immediate
Move	`move`	R[rd] = R[rs]

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVEDACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes

CS2100 Computer Organisation

Tutorial #10: Pipelining

(Week 12: 1 – 5 November 2021)

LumiNUS Discussion Questions

D1. Suppose the four stages in some 4-stage pipeline take the following timing: 2ns, 3ns, 4ns, and 2ns. Given 1000 instructions, what is the speedup (in two decimal places) of the pipelined processor compared to the non-pipelined single-cycle processor?

D2. Let's try to understand pipeline processor by doing a detailed trace. Suppose the pipeline registers (also known as pipeline latches) store the following information*:*

IF / ID	
No Control Signal	--------
PC+4	
OpCode	
Rs	
Rt	
Rd	
Funct	
Imm(16)	

ID / EX	
MToR	
RegWr	
MemRd	
MemWr	
Branch	
RegDst	
ALUsrc	
ALUop	
PC+4	
ALUOpr1	
ALUOpr2	
Rt	
Rd	
Imm(32)	

EX / MEM	
MToR	
RegWr	
MemRd	
MemWr	
Branch	
BrcTgt	
isZero?	
ALURes	
ALUOpr2	
DstRNum	

MEM / WB	
MToR	
RegWr	
MemRes	
ALURes	
DstRNum	

Show the progress of the following instructions through the pipeline stages by filling in the content of pipeline registers. Note that these are the same instructions from Tutorial #5 Question 1 so that you can reuse some of the answers here.

```
i.   0x8df80000  # lw $24, 0($15)      #Inst.Addr = 0x100
ii.  0x1023000C  # beq $1, $3, 12      #Inst.Addr = 0x100
iii. 0x0285c822  # sub $25, $20, $5    #Inst.Addr = 0x100
```

Assume that registers 1 to 31 have been initialized to a value that is equal to 101 + its register number. i.e. [$1] = 102, [$31] = 132 etc. You can put "X" in fields that are irrelevant for that instruction. Do note that in reality, these fields are actually generated but not utilized.

Part (i) has been worked out for you.

i. `0x8df80000 # lw $24, 0($15)    #Inst.Addr = 0x100`

IF / ID	
No Control Signal	--------
PC+4	0x104
OpCode	0x23
Rs	$15
Rt	$24
Rd	X
Funct	X
Imm(16)	0

ID / EX	
MToR	1
RegWr	1
MemRd	1
MemWr	0
Branch	0
RegDst	0
ALUsrc	1
ALUop	00
PC+4	0x104
ALUOpr1	116
ALUOpr2	X
Rt	$24
Rd	X
Imm(32)	0

EX / MEM	
MToR	1
RegWr	1
MemRd	1
MemWr	0
Branch	0
BrcTgt	X
isZero?	X
ALURes	116
ALUOpr2	X
DstRNum	$24

MEM / WB	
MToR	1
RegWr	1
MemRes	Mem(116)
ALURes	X
DstRNum	$24

D2. Given the following three formulas (See Lecture #20, Section 5 Performance):

$$CT_{seq} = \sum_{k=1}^{N} T_k$$

$$CT_{pipeline} = \max(T_k) + T_d$$

$$Speedup_{pipeline} = \frac{CT_{seq} \times InstNum}{CT_{pipeline} \times (N + InstNum - 1)}$$

For each of the following processor parameters, calculate CT_{seq}, $CT_{pipeline}$ and $Speedup_{pipeline}$ (to two decimal places) for 10 instructions and for 10 million instructions.

	Stages Timing (for 5 stages, in ps)	Latency of pipeline register (in ps)
a.	300, 100, 200, 300, 100 (slow memory)	0
b.	200, 200, 200, 200, 200	40
c.	200, 200, 200, 200, 200 (ideal)	0

Tutorial Questions

1. [AY2014/5 Semester 2 Exam]
   Refer to the following MIPS program:

```
      # register $s0 contains a 32-bit value
      # register $s1 contains a non-zero 8-bit value
      #       at the right most (least significant) byte
      add  $t0, $s0, $zero      #inst A
      add  $s2, $zero, $zero    #inst B
lp:   bne  $s2, $zero, done     #inst C
      beq  $t0, $zero, done     #inst D
      andi $t1, $t0, 0xFF       #inst E
      bne  $s1, $t1, nt         #inst F
      addi $s2, $s2, 1          #inst G
nt:   srl  $t0, $t0, 8          #inst H
      j    lp                   #inst J
done:
```

   We assume that the register **$s0** contains **0xAFAFFAFA** and **$s1** contains **0xFF**.

   Given a 5-stage MIPS pipeline processor, for each of the parts below, give the total number of cycles needed for the first iteration of the execution from instructions **A** to **H** (i.e. excluding the "**j lp**" instruction). Remember to include the cycles needed for instruction **H** to finish the WB stage. Note that the questions are independent from each other.

   a. With only data forwarding mechanisms and no control hazard mechanism.
   b. With data forwarding and "assume not taken" branch prediction. Note that there is no early branching.
      [Recall that early branching means branch decision is made at stage 2 (Decode stage); no early branch means branch decision is made at stage 4 (Memory stage).]
   c. By swapping two instructions (from Instructions **A** to **H**), we can improve the performance of **early branching (with all additional forwarding paths)**. Give the two instructions that can be swapped. You only need to indicate the instruction letters in your answer.

   Give the total number of cycles needed for the execution of the whole code in the worst case for each of the following assumptions. You may assume that the jump instruction (**j**) computes the address of the instruction to jump to in the MEM stage.

   d. With only data forwarding mechanisms and no control hazard mechanism.
   e. With data forwarding and "assume not taken" branch prediction. Note that there is no early branching.

2. [AY2017/8 Semester 2 Exam]
   Refer to the MIPS code below. *A* and *B* are integer arrays whose base addresses are in **$s0** and **$s1** respectively. The arrays are of the same size *n* (number of elements). **$s2** contains the value *n*. For this question, we will focus on the code from Instruction 1 onwards.

```
 .data
A: .word 11, 9, 31, 2, 9, 1, 6, 10
B: .word 3, 7, 2, 12, 11, 41, 19, 35
n: .word 8
.text
main: la   $s0, A     # $s0 is the base address of array A
      la   $s1, B     # $s1 is the base address of array B
      la   $t0, n     # $t0 is the addr of n (size of array)
                      # $s2 is the content of n
      beq  $s2, $zero, End   # Inst1
      addi $t8, $s2, -1      # Inst2
      sll  $t8, $t8, 2       # Inst3
Loop: add  $t0, $s0, $t8     # Inst4
      add  $t1, $s1, $t8     # Inst5
      lw   $t2, 0($t0)       # Inst6
      lw   $t3, 0($t1)       # Inst7
      andi $t4, $t3, 3       # Inst8
      addi $t4, $t4, -3      # Inst9
      beq  $t4, $zero, A1    # Inst10
      add  $t2, $t2, $t3     # Inst11
      j    A2                # Inst12
A1:   addi $t2, $t2, 1       # Inst13
A2:   sw   $t2, 0($t0)       # Inst14
      addi $t8, $t8, -8      # Inst15
      slt  $t7, $t8, $zero   # Inst16
      beq  $t7, $zero, Loop  # Inst17
End:
```

Assuming a 5-stage MIPS pipeline system <u>with forwarding and early branching</u>, that is, the branch decision is made at the ID stage. No branch prediction is made and no delayed branching is used. For the jump (**j**) instruction, the computation of the target address to jump to is done at the ID stage as well.

Assume also that the first **beq** instruction begins at cycle 1.

a. Suppose arrays *A* and *B* now each contains 200 positive integers. What is the minimum number and maximum number of instructions executed? (Consider only the above code segment from Inst1 to Inst17.)

b. List out the instructions where some stall cycle(s) are inserted in executing that instruction in the pipeline. These include delay caused by data dependency and control hazard. You may write the instruction number InstX instead of writing out the instruction in full.

c. How many cycles does one iteration of the loop (from Inst1 to Inst17) take if the **beq** instruction at Inst10 branches to *A1*? You have to count until the WB stage of Inst17.

d. How many cycles does one iteration of the loop (from Inst1 to Inst17) take if the **beq** instruction at Inst10 does not branch to *A1*? You have to count until the WB stage of Inst17.

A blank pipeline chart is shown in the next page for your use. The Microsoft word version of it is also available on LumiNUS > Files and the CS2100 website.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30

3. [AY2020/21 Semester 2 Exam]
Study the following MIPS code on integer arrays *A* and *B* which contain the same number of elements. $s0 and $s1 contain the base addresses of *A* and *B* respectively; $s2 is the number of elements in array *A*; $s5 is *count*.

```
      add  $s5, $0, $0    # I1
      add  $t0, $0, $0    # I2
loop: slt  $t8, $t0, $s2  # I3
      beq  $t8, $0, end   # I4
      sll  $t1, $t0, 2    # I5
      add  $t3, $t1, $s0  # I6
      lw   $s3, 0($t3)    # I7
      andi $t9, $s3, 1    # I8
      beq  $t9, $0, skip  # I9
      add  $t4, $t1, $s1  # I10
      lw   $s4, 0($t4)    # I11
      sub  $s3, $s3, $s4  # I12
      sw   $s3, 0($t3)    # I13
      addi $s5, $s5, 1    # I14
skip: addi $t0, $t0, 1    # I15
      j    loop           # I16
end:
```

Assuming a 5-stage MIPS pipeline and all elements in array *A* are positive odd integers, answer the following questions. You need to count until the last stage of instruction I16.

(a) How many cycles does this code segment take to complete its execution in the first iteration (I1 to I16) in an ideal pipeline, that is, one with no delays?

For parts (b) to (d) below, given the assumption for each part, how many additional cycles does this code segment (I1 to I16) take to complete its execution in the first iteration as compared to an ideal pipeline? (For example, if part (a) takes 12 cycles and part (b) takes 20 cycles, you are to answer part (b) with the value 8 and not 20.)

(b) Assuming without forwarding and branch decision is made at MEM stage (stage 4). No branch prediction is made and no delayed branching is used.

(c) Assuming without forwarding and branch decision is made at ID stage (stage 2). No branch prediction is made and no delayed branching is used.

(d) Assuming with forwarding and branch decision is made at ID stage (stage 2). Branch prediction is made where the branch is predicted not taken, and no delayed branching is used.

(e) Assuming the setting in part (d) above and you are not allowed to modify any of the instructions, is it possible to reduce the additional delay cycles in part (d) by rearranging some instructions, and if possible, by how many cycles? Explain your answer. (Answer with no explanation will not be awarded any mark.)

CS2100 Computer Organisation

Tutorial #10: Pipelining

Answers to Selected Questions

Tutorial Questions

1. [AY2014/5 Semester 2 Exam]
   Refer to the following MIPS program:

```
     # register $s0 contains a 32-bit value
     # register $s1 contains a non-zero 8-bit value
     #       at the right most (least significant) byte
     add  $t0, $s0, $zero      #inst A
     add  $s2, $zero, $zero    #inst B
lp:  bne  $s2, $zero, done     #inst C
     beq  $t0, $zero, done     #inst D
     andi $t1, $t0, 0xFF       #inst E
     bne  $s1, $t1, nt         #inst F
     addi $s2, $s2, 1          #inst G
nt:  srl  $t0, $t0, 8          #inst H
     j    lp                   #inst J
done:
```

We assume that the register **$s0** contains **0xAFAFFAFA** and **$s1** contains **0xFF**.

Given a 5-stage MIPS pipeline processor, for each of the parts below, give the total number of cycles needed for the first iteration of the execution from instructions **A** to **H** (i.e. excluding the "**j lp**" instruction). Remember to include the cycles needed for instruction **H** to finish the WB stage. Note that the questions are independent from each other.

a. With only data forwarding mechanisms and no control hazard mechanism.

b. With data forwarding and "assume not taken" branch prediction. Note that there is no early branching.
   [Recall that early branching means branch decision is made at stage 2 (Decode stage); no early branch means branch decision is made at stage 4 (Memory stage).]

c. By swapping two instructions (from Instructions **A** to **H**), we can improve the performance of **early branching (with all additional forwarding paths)**. Give the two instructions that can be swapped. You only need to indicate the instruction letters in your answer.

Give the total number of cycles needed for the execution of the whole code in the worst case for each of the following assumptions. You may assume that the jump instruction (**j**) computes the address of the instruction to jump to in the MEM stage.

d. With only data forwarding mechanisms and no control hazard mechanism.

e. With data forwarding and "assume not taken" branch prediction. Note that there is no early branching.

Answers:

(a) 20 cycles

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
add	F	D	E	M	W															
add		F	D	E	M	W														
bne			F	D	E	M	W													
beq							F	D	E	M	W									
andi											F	D	E	M	W					
bne												F	D	E	M	W				
~~addi~~	The addi instruction is not executed.																			
srl																F	D	E	M	W

(b) 14 cycles

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
add	F	D	E	M	W									
add		F	D	E	M	W								
bne			F	D	E	M	W							
beq				F	D	E	M	W						
andi					F	D	E	M	W					
bne						F	D	E	M	W				
addi							F	D	E	*	*			
srl								F	D	*	*	*		
j									F	*	*	*	*	
srl										F	D	E	M	W

Cost of wrong prediction

(c) Swap instructions A and B to reduce the delay between instructions B and C.

```
     add  $s2, $zero, $zero    #inst B
     add  $t0, $s0, $zero      #inst A
lp:  bne  $s2, $zero, done     #inst C
     beq  $t0, $zero, done     #inst D
     andi $t1, $t0, 0xFF       #inst E
     bne  $s1, $t1, nt         #inst F
     addi $s2, $s2, 1          #inst G
nt:  srl  $t0, $t0, 8          #inst H
     j    lp                   #inst J
done:
```

(d)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
add	F	D	E	M	W																
add		F	D	E	M	W															
bne			F	D	E	M	W														
beq							F	D	E	M	W										
andi											F	D	E	M	W						
bne												F	D	E	M	W					
~~addi~~																					
srl																F	D	E	M	W	
j																	F	D	E	M	W
bne																					F

In the worst case, 4 bytes of the data are examined → 4 iterations. The loop from Instructions C to J (one iteration) takes 18 cycles (not counting the WB stage of the j instruction which overlaps with the bne instruction). There are 2 cycles before the first iteration, and 9 cycles for Instructions C and D in the fifth iteration.
Therefore, total = 2 + (4×18) + 9 = **83 cycles**.

(e)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
add	F	D	E	M	W										
add		F	D	E	M	W									
bne			F	D	E	M	W								
beq				F	D	E	M	W							
andi					F	D	E	M	W						
bne						F	D	E	M	W					
addi							F	D	E	*	*				
srl								F	D	*	*	*			
j									F	*	*	*	*		
srl										F	D	E	M	W	
j											F	D	E	M	W
bne															F

In the worst case, 4 bytes of the data are examined → 4 iterations. The loop from Instructions C to J (one iteration) takes 12 cycles (not counting the WB stage of the j instruction which overlaps with the bne instruction). There are 2 cycles before the first iteration, and 6 cycles for Instructions C and D in the fifth iteration.
Therefore, total = 2 + (4×12) + 6 = **56 cycles**.

2. [AY2017/8 Semester 2 Exam]
   Refer to the MIPS code below. *A* and *B* are integer arrays whose base addresses are in **$s0** and **$s1** respectively. The arrays are of the same size *n* (number of elements). **$s2** contains the value *n*. For this question, we will focus on the code from Instruction 1 onwards.

```
 .data
A: .word 11, 9, 31, 2, 9, 1, 6, 10
B: .word 3, 7, 2, 12, 11, 41, 19, 35
n: .word 8
.text
main: la   $s0, A     # $s0 is the base address of array A
      la   $s1, B     # $s1 is the base address of array B
      la   $t0, n     # $t0 is the addr of n (size of array)
                      # $s2 is the content of n
      beq  $s2, $zero, End   # Inst1
      addi $t8, $s2, -1      # Inst2
      sll  $t8, $t8, 2       # Inst3
Loop: add  $t0, $s0, $t8     # Inst4
      add  $t1, $s1, $t8     # Inst5
      lw   $t2, 0($t0)       # Inst6
      lw   $t3, 0($t1)       # Inst7
      andi $t4, $t3, 3       # Inst8
      addi $t4, $t4, -3      # Inst9
      beq  $t4, $zero, A1    # Inst10
      add  $t2, $t2, $t3     # Inst11
      j    A2                # Inst12
A1:   addi $t2, $t2, 1       # Inst13
A2:   sw   $t2, 0($t0)       # Inst14
      addi $t8, $t8, -8      # Inst15
      slt  $t7, $t8, $zero   # Inst16
      beq  $t7, $zero, Loop  # Inst17
End:
```

Assuming a 5-stage MIPS pipeline system <u>with forwarding and early branching</u>, that is, the branch decision is made at the ID stage. No branch prediction is made and no delayed branching is used. For the jump (**j**) instruction, the computation of the target address to jump to is done at the ID stage as well.

Assume also that the first **beq** instruction begins at cycle 1.

a. Suppose arrays *A* and *B* now each contains 200 positive integers. What is the minimum number and maximum number of instructions executed? (Consider only the above code segment from Inst1 to Inst17.)

b. List out the instructions where some stall cycle(s) are inserted in executing that instruction in the pipeline. These include delay caused by data dependency and control hazard. You may write the instruction number InstX instead of writing out the instruction in full.

c. How many cycles does one iteration of the loop (from Inst1 to Inst17) take if the **beq** instruction at Inst10 branches to *A1*? You have to count until the WB stage of Inst17.

d. How many cycles does one iteration of the loop (from Inst1 to Inst17) take if the **beq** instruction at Inst10 does not branch to *A1*? You have to count until the WB stage of Inst17.

Answers:

The code does this:

```
int i;
for (i=n-1; i>=0; i-=2) {
   if (B[i]%4 == 3)
      A[i] = A[i] + 1;
   else
      A[i] = A[i] + B[i];
}
```

(a) Minimum = 3 + 100 × 12 = 1203

Maximum = 3 + 100 × 13 = 1303

In the loop (Inst4 to Inst17), there are two paths after Inst10: one that skips Inst11 and Inst12, and the other skips Inst13.

(b) Due to control: Inst2, Inst4, Inst11, Inst13

Due to data: Inst8, Inst10, Inst17

(c) 24 cycles

(d) 26 cycles

3. [AY2020/21 Semester 2 Exam]
Study the following MIPS code on integer arrays *A* and *B* which contain the same number of elements. $s0 and $s1 contain the base addresses of *A* and *B* respectively; $s2 is the number of elements in array *A*; $s5 is *count*.

```
      add  $s5, $0, $0    # I1
      add  $t0, $0, $0    # I2
loop: slt  $t8, $t0, $s2  # I3
      beq  $t8, $0, end   # I4
      sll  $t1, $t0, 2    # I5
      add  $t3, $t1, $s0  # I6
      lw   $s3, 0($t3)    # I7
      andi $t9, $s3, 1    # I8
      beq  $t9, $0, skip  # I9
      add  $t4, $t1, $s1  # I10
      lw   $s4, 0($t4)    # I11
      sub  $s3, $s3, $s4  # I12
      sw   $s3, 0($t3)    # I13
      addi $s5, $s5, 1    # I14
skip: addi $t0, $t0, 1    # I15
      j    loop           # I16
end:
```

Assuming a 5-stage MIPS pipeline and all elements in array *A* are positive odd integers, answer the following questions. You need to count until the last stage of instruction I16.

(a) How many cycles does this code segment take to complete its execution in the first iteration (I1 to I16) in an ideal pipeline, that is, one with no delays?

For parts (b) to (d) below, given the assumption for each part, how many <u>additional cycles</u> does this code segment (I1 to I16) take to complete its execution in the first iteration as compared to an ideal pipeline? (For example, if part (a) takes 12 cycles and part (b) takes 20 cycles, you are to answer part (b) with the value 8 and not 20.)

(b) Assuming <u>without forwarding and branch decision is made at MEM stage (stage 4)</u>. No branch prediction is made and no delayed branching is used.

(c) Assuming <u>without forwarding and branch decision is made at ID stage (stage 2)</u>. No branch prediction is made and no delayed branching is used.

(d) Assuming <u>with forwarding and branch decision is made at ID stage (stage 2)</u>. Branch prediction is made where the branch is predicted not taken, and no delayed branching is used.

(e) Assuming the setting in part (d) above and you are not allowed to modify any of the instructions, is it possible to reduce the additional delay cycles in part (d) by rearranging some instructions, and if possible, by how many cycles? Explain your answer. (Answer with no explanation will not be awarded any mark.)

Answers:

(a) 16 + 5 – 1 = **20** cycles.

Delays are highlighted under the columns (b), (c), (d) for parts (b),(c),(d) below respectively.

(b) **+24** cycles.

(c) **+20** cycles.

(d) **+4** cycles.

```
                                               (b)   (c)   (d)
      add  $s5, $0, $0      # I1
      add  $t0, $0, $0      # I2
loop: slt  $t8, $t0, $s2    # I3     +2    +2
      beq  $t8, $0, end     # I4     +2    +2    +1
      sll  $t1, $t0, 2      # I5     +3    +1
      add  $t3, $t1, $s0    # I6     +2    +2
      lw   $s3, 0($t3)      # I7     +2    +2
      andi $t9, $s3, 1      # I8     +2    +2    +1
      beq  $t9, $0, skip    # I9     +2    +2    +1
      add  $t4, $t1, $s1    # I10    +3    +1
      lw   $s4, 0($t4)      # I11    +2    +2
      sub  $s3, $s3, $s4    # I12    +2    +2    +1
      sw   $s3, 0($t3)      # I13    +2    +2
      addi $s5, $s5, 1      # I14
skip: addi $t0, $t0, 1      # I15
      j    loop             # I16
end:
                                Total:   +24   +20   +4
```

(e) Other answers possible. Example:

Move I14 (addi $s5, $s5, 1) to between I11 (lw $s4, 0($t4)) and I12 (sub $s3, $s3, $s4) to remove the 1 cycle of delay at I12.

CS2100 Computer Organisation

Tutorial #11: Cache

(Week 13: 8 – 12 November 2021)

LumiNUS Discussion Question

D1. [CS2100 AY2007/8 Semester 2 Exam Question]
A machine with a word size of 16 bits and address width of 32 bits has a **direct-mapped cache** with 16 blocks and a block size of 2 words, initially empty.

(a) Given a sequence of memory references as shown below, where each reference is given as a byte address in both decimal and hexadecimal forms, indicate whether the reference is a hit (H) or a miss (M).

Memory address		**Hit (H) or Miss (M)?**	**(For reference)**
(in decimal)	**(in hexadecimal)**		
4	0x4	**M**	0000 … 0000 0100
92	0x5C		0000 … 0101 1100
7	0x7		0000 … 0000 0111
146	0x92		0000 … 1001 0010
30	0x1E		0000 … 0001 1110
95	0x5F		0000 … 0101 1111
176	0xB0		0000 … 1011 0000
93	0x5D		0000 … 0101 1101
145	0x91		0000 … 1001 0001
264	0x108		0000 … 1 0000 1000
6	0x6		0000 … 0000 0110

(b) Given the above sequence of memory references, fill in the final contents of the cache. Use the notation M[*i*] to denote the word starting at memory address *i*, where *i* is in hexadecimal. If a block is replaced, cross out the content in the cache and write the new content over it.

Index	**Tag value**	**Word 0**	**Word 1**
0			
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

Tutorial Questions

1. Here is a series of address references in decimal: 4, 16, 32, 20, 80, 68, 76, 224, 36, 44, 16, 172, 20, 24, 36, and 68 in a MIPS machine. Assuming a **direct-mapped cache** with 16 one-word blocks that is initially empty, label each address reference as a hit or miss and show the content of the cache.

   You may write the data word starting at memory address X as M[X]. (For example, data word starting at memory address 12 is written as M[12]. This implies that the word includes the 4 bytes of data at addresses 12, 13, 14 and 15.) You may write the tag values as decimal numbers. If a block is replaced in the cache, cross out the corresponding content in the cache, and write the new content over it.

2. Use the series of references given in question 1 above: 4, 16, 32, 20, 80, 68, 76, 224, 36, 44, 16, 172, 20, 24, 36, and 68 in a MIPS machine. Assuming a **two-way set-associative cache** with two-word blocks and a total size of 16 words that is initially empty, label each address reference as a hit or miss and show the content of the cache. Assume **LRU** replacement policy.

   You may write the data word starting at memory address X as M[X]. (For example, data word starting at memory address 12 is written as M[12]. This implies that the word includes the 4 bytes of data at addresses 12, 13, 14 and 15.) You may write the tag values as decimal numbers. If a block is replaced in the cache, cross out the corresponding content in the cache, and write the new content over it.

3. Although we use only data memory as example in the cache lecture, the principle covered is equally applicable to the instruction memory. This question takes a look at both the instruction cache and data cache.

   The code below is from Tutorial 3 Question 1 (***palindrome checking***) with the following variable mappings:

   low → \$s0, high→ \$s1, matched → \$s3, base of string[]→ \$s4, size → \$s5

#	Code	Comment
i0	`[some instruction]`	
i1	`addi $s0, $zero, 0`	`# low = 0`
i2	`addi $s1, $s5, -1`	`# high = size-1`
i3	`addi $s3, $zero, 1`	`# matched = 1`
	`loop:`	
i4	`slt  $t0, $s0, $s1`	`# (low < high)?`
i5	`beq  $t0, $zero, exit`	`# exit if (low >= high)`
i6	`beq  $s3, $zero, exit`	`# exit if (matched == 0)`
i7	`add  $t1, $s4, $s0`	`# address of string[low]`
i8	`lb   $t2, 0($t1)`	`# t2 = string[low]`
i9	`addi $t3, $s4, $s1`	`# address of string[high]`
i10	`lb   $t4, 0($t3)`	`# t4 = string[high]`
i11	`beq  $t2, $t4, else`	
i12	`addi $s3, $zero, 0`	`# matched = 0`
i13	`j    endW`	`# can be "j loop"`
	`else:`	
i14	`addi $s0, $s0, 1`	`# low++`
i15	`addi $s1, $s1, -1`	`# high–`
	`endW:`	
i16	`j    loop`	`# end of while`
	`exit:`	
i17	`[some instruction]`	

Parts (a) to (d) assume that instruction i0 is stored at memory address 0x0.

(a) Instruction cache: **Direct mapped with 2 blocks of 16 bytes each** (i.e. each block can hold 4 consecutive instructions).

   Starting with an empty cache, the fetching of instruction i1 will cause a cache miss. After the cache miss is resolved, we now have the following instructions in the instruction cache:

Instruction Cache Block 0	[i0, **i1**, **i2**, **i3**]
Instruction Cache Block 1	[empty]

   Fetching of i2 and i3 are all cache hits as they can be found in the cache.

   Assuming the string being checked is a palindrome. Show the instruction cache block content **at the end of the 1st iteration (i.e. up to instruction i16).**

(b) If the loop is executed for a total of 10 iterations, what is the total number of cache hits (i.e. after the 10th "j loop" is fetched)?

(c) Suppose we change the instruction cache to:

- **Direct mapped with 4 blocks of 8 bytes each** (i.e. each block can hold 2 consecutive instructions).

Assuming the string being checked is a palindrome. Show the instruction cache block content **at the end of the 1st iteration (i.e. up to instruction i16).**

Instruction Cache Block 0	
Instruction Cache Block 1	
Instruction Cache Block 2	
Instruction Cache Block 3	

(d) If the loop is executed for a total of 10 iterations, what is the total number of cache hits (i.e. after the 10th "j loop" is fetched)?

Let us now turn to the study of **data cache**. We will assume the following scenario for parts (e) to (g):

- The string being checked is **64-character long**. The first character is located at location **0x1000**.
- The string is a palindrome (i.e. it will go through 32 iterations of the code).

(e) Given a **direct mapped data cache with 2 cache blocks, each block is 8 bytes**, what is the final content of the data cache at the end of the code execution (after the code failed the beq at i5)? Use **s[X..Y]** to indicate the data **string[X]** to **string[Y].**

Data Cache Block #0	
Data Cache Block #1	

(f) What is the hit rate of (e)? Give your answer in a fraction or a percentage correct to two decimal places.

(g) Suppose the string is now **72-character long**, the first character is still located at location **0x1000** and the string is still a palindrome, what is the hit rate at the end of the execution?

CS2100 Computer Organisation

Tutorial #11: Cache

Answers to Selected Questions

Tutorial Questions

2. Use the series of references given in question 1 above: 4, 16, 32, 20, 80, 68, 76, 224, 36, 44, 16, 172, 20, 24, 36, and 68 in a MIPS machine. Assuming a **two-way set-associative cache** with two-word blocks and a total size of 16 words that is initially empty, label each address reference as a hit or miss and show the content of the cache. Assume **LRU** replacement policy.

   You may write the data word starting at memory address X as M[X]. (For example, data word starting at memory address 12 is written as M[12]. This implies that the word includes the 4 bytes of data at addresses 12, 13, 14 and 15.) You may write the tag values as decimal numbers. If a block is replaced in the cache, cross out the corresponding content in the cache, and write the new content over it.

   Answer:

   Since this is a MIPS machine, a word consists of 4 bytes or 32 bits.
   Should first work out the tag, set index, and offset fields:

27 bits	2 bits	3
Tag	Set Index	Offset

4:	00…000	00	100	← Miss
16:	00…000	10	000	← Miss
32:	00…001	00	000	← Miss
20:	00…000	10	100	← Hit
80:	00…010	10	000	← Miss
68:	00…010	00	100	← Miss
76:	00…010	01	100	← Miss
224:	00…111	00	000	← Miss
36:	00…001	00	100	← Miss
44:	00…001	01	100	← Miss
16:	00…000	10	000	← Hit
172:	00…101	01	100	← Miss
20:	00…000	10	100	← Hit
24:	00…000	11	000	← Miss
36:	00…001	00	100	← Hit
68:	00…010	00	100	← Miss

Cache set	Valid bit	Tag	Word0	Word1	Valid bit	Tag	Word0	Word1
0	~~0~~ 1	~~0~~ ~~2~~ 1	~~M[0]~~ ~~M[64]~~ M[32]	~~M[4]~~ ~~M[68]~~ M[36]	~~0~~ 1	~~1~~ ~~7~~ 2	~~M[32]~~ ~~M[224]~~ M[64]	~~M[36]~~ ~~M[228]~~ M[68]
1	~~0~~ 1	~~2~~ 5	~~M[72]~~ M[168]	~~M[76]~~ M[172]	~~0~~ 1	1	M[40]	M[44]
2	~~0~~ 1	0	M[16]	M[20]	~~0~~ 1	2	M[80]	M[84]
3	~~0~~ 1	0	M[24]	M[28]	0			

3. Although we use only data memory as example in the cache lecture, the principle covered is equally applicable to the instruction memory. This question takes a look at both the instruction cache and data cache.

   The code below is from Tutorial 8 Question 1 (***palindrome checking***) with the following variable mappings:

   low → \$s0, high→ \$s1, matched → \$s3, base of string[]→ \$s4, size → \$s5

#	Code	Comment
i0	`    [some instruction]`	
i1	`    addi $s0, $zero, 0`	`# low = 0`
i2	`    addi $s1, $s5, -1`	`# high = size-1`
i3	`    addi $s3, $zero, 1`	`# matched = 1`
	`loop:`	
i4	`    slt  $t0, $s0, $s1`	`# (low < high)?`
i5	`    beq  $t0, $zero, exit`	`# exit if (low >= high)`
i6	`    beq  $s3, $zero, exit`	`# exit if (matched == 0)`
i7	`    add  $t1, $s4, $s0`	`# address of string[low]`
i8	`    lb   $t2, 0($t1)`	`# t2 = string[low]`
i9	`    addi $t3, $s4, $s1`	`# address of string[high]`
i10	`    lb   $t4, 0($t3)`	`# t4 = string[high]`
i11	`    beq  $t2, $t4, else`	
i12	`    addi $s3, $zero, 0`	`# matched = 0`
i13	`    j    endW`	`# can be "j loop"`
	`else:`	
i14	`    addi $s0, $s0, 1`	`# low++`
i15	`    addi $s1, $s1, -1`	`# high–`
	`endW:`	
i16	`    j    loop`	`# end of while`
	`exit:`	
i17	`    [some instruction]`	

Parts (a) to (d) assume that instruction i0 is stored at memory address 0x0.

(a) Instruction cache: **Direct mapped with 2 blocks of 16 bytes each** (i.e. each block can hold 4 consecutive instructions).

Starting with an empty cache, the fetching of instruction i1 will cause a cache miss. After the cache miss is resolved, we now have the following instructions in the instruction cache:

Instruction Cache Block 0	[i0, **i1**, **i2**, **i3**]
Instruction Cache Block 1	[empty]

Fetching of i2 and i3 are all cache hits as they can be found in the cache.

Assuming the string being checked is a palindrome. Show the instruction cache block content **at the end of the 1st iteration (i.e. up to instruction i16).**

Answer:

Instruction Cache Block 0	**[i16, ……..]**
Instruction Cache Block 1	**[i12, i13, i14, i15]**

Working: Instructions executed = i1 to i11, i14 to i16

Block #0, Cache index = 0	[i0, i1, i2, i3]
Block #1, Cache index = 1	[i4, i5, i6, i7]
Block #2, Cache index = 0	[i8, i9, i10, i11]
Block #3, Cache index = 1	[i12, i13, i14, i15]
Block #4, Cache index = 0	[i16, other….]

(b) If the loop is executed for a total of 10 iterations, what is the total number of cache hits (i.e. after the 10th "j loop" is fetched)?

Answer:

Working (1st Iteration):

i1	i2	i3	i4	i5	i6	i7	i8	i9	i10	i11	i14	i15	i16
M	H	H	M	H	H	H	M	H	H	H	M	H	M

Working (2nd iteration onward):

i4	i5	i6	i7	i8	i9	i10	i11	i14	i15	i16
M	H	H	H	M	H	H	H	M	H	M

Total hits = 9 (1st iteration) + 7×9 (remaining 9 iterations) = **72**

(c) Suppose we change the instruction cache to:

- **Direct mapped with 4 blocks of 8 bytes each** (i.e. each block can hold 2 consecutive instructions).

Assuming the string being checked is a palindrome. Show the instruction cache block content **at the end of the 1st iteration (i.e. up to instruction i16).**

Answer:

Instruction Cache Block 0	**[i16, …]**
Instruction Cache Block 1	**[i10, i11]**
Instruction Cache Block 2	**[i4, i5]**
Instruction Cache Block 3	**[i14, i15]**

Working:

First, find out the block information for the full code:

Block #0, Cache index = 0	[i0, i1]
Block #1, Cache index = 1	[i2, i3]
Block #2, Cache index = 2	[i4, i5]
Block #3, Cache index = 3	[i6, i7]
Block #4, Cache index = 0	[i8, i9]
Block #5, Cache index = 1	[i10, i11]
Block #6, Cache index = 2	[i12, i13]
Block #7, Cache index = 3	[i14, i15]
Block #8, Cache index = 0	[i16, …]

Second, use the execution pattern to find out what is accessed, since we execute i1 to i11 (Block #0 to Block #5) then i14 to i16 (Block #7 and Block #8), we get the final cache content as shown. You should note that Block #6 [i12, i13] is not accessed in this particular execution.

(d) If the loop is executed for a total of 10 iterations, what is the total number of cache hits (i.e. after the 10th "j loop" is fetched)?

Answer:

Working (1st Iteration):

i1	i2	i3	i4	i5	i6	i7	i8	i9	i10	i11	i14	i15	i16
M	M	H	M	H	M	H	M	H	M	H	M	H	M

Working (2nd iteration onward):

i4	i5	i6	i7	i8	i9	i10	i11	i14	i15	i16
H	H	M	H	M	H	H	H	M	H	M

Total hits = 6 (1st iteration) + 7×9 (remaining 9 iterations) = **69**

Let us now turn to the study of **data cache**. We will assume the following scenario for parts (e) to (g):

- The string being checked is **64-character long**. The first character is located at location **0x1000**.
- The string is a palindrome (i.e. it will go through 32 iterations of the code).

(e) Given a **direct mapped data cache with 2 cache blocks, each block is 8 bytes**, what is the final content of the data cache at the end of the code execution (after the code failed the beq at i5)? Use **s[X..Y]** to indicate the data **string[X]** to **string[Y].**

Answer:

Data Cache Block #0	**s[32..39]**
Data Cache Block #1	**s[24..31]**

Access patterns = s[0], s[63], s[1], s[62], …, s[31], s[32]

Blocks information (blocks that can go into the same cache location are listed together):

Cache index = 0	s[0..7] [16..23] [32..39] [48..55]
Cache index = 1	s[8..15] [24..31] [40..47] [56..63]

(f) What is the hit rate of (e)? Give your answer in a fraction or a percentage correct to two decimal places.

Answer:

Observation: the access pattern nicely alternates between Block0-Block1 and Block1-Block0. So, in general, other than the first miss to bring in a block, the remaining 7 accesses on the block are all hits.

Hence, hit rate = **7/8** or **87.50%**

(g) Suppose the string is now **72-character long**, the first character is still located at location **0x1000** and the string is still a palindrome, what is the hit rate at the end of the execution?

Answer:

Access patterns = s[0], s[71], s[1], s[70], …, s[35], s[36]

Blocks information (blocks that can go into the same cache location are listed together):

Cache index = 0	s[0..7] [16..23] [32..39] [48..55] [64..71]
Cache index = 1	s[8..15] [24..31] [40..47] [56…63]

Observation: the access pattern is either Block0-Block0 or Block1-Block1. So, every access is a miss, except the last block [32..39]! This is an example of ***cache thrashing*** (you can imagine the cache is "beaten up" pretty badly ☺).

Hence, hit rate = **7/72** (the last 7 accesses on block [32..39]) or **9.72%**