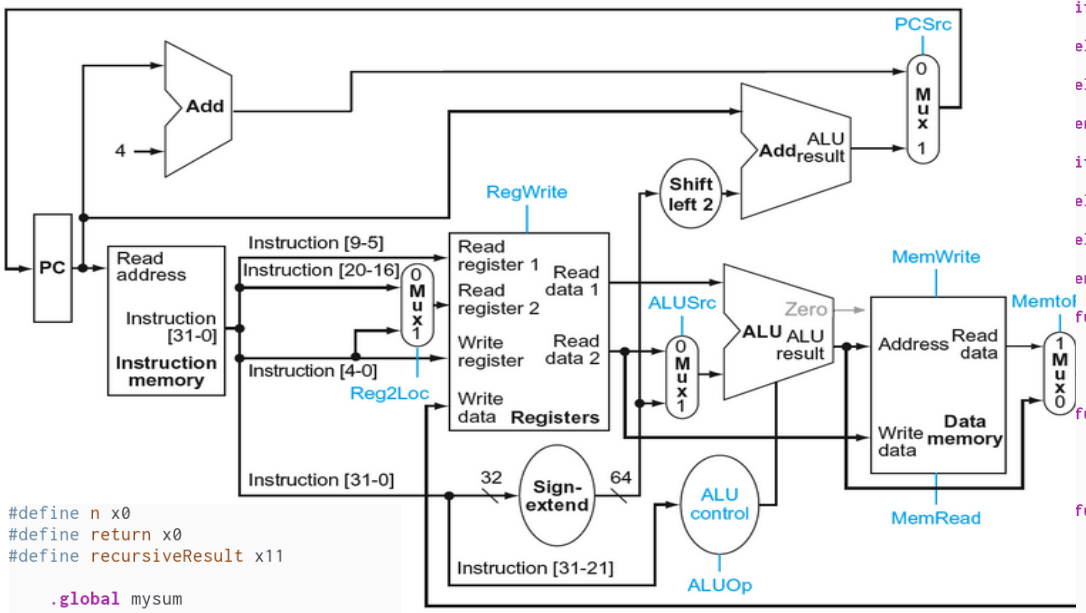
Input or output	Signal name	R-format	LDUR	STUR	CBZ
Inputs	I[31]	1	1	1	1
	I[30]	X	1	1	0
	I[29]	X	1	1	1
	I[28]	0	1	1	1
	I[27]	1	1	1	0
	I[26]	0	0	0	1
	I[25]	1	0	0	0
	I[24]	X	0	0	0
	I[23]	0	0	0	X
	I[22]	0	1	0	X
	I[21]	0	0	0	X
Outputs	Reg2Loc	0	X	1	1
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Signal name	Effect when deasserted	Effect when asserted
Reg2Loc	The register number for Read register 2 comes from the Rm field (bits 20:16).	The register number for Read register 2 comes from the Rt field (bits 4:0).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 32 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

MEM Hazard: As mentioned above, there is no hazard in the WB stage, because we assume that the register file supplies the correct result if the instruction in the ID stage reads the same register written by the instruction in the WB stage. Such a register file performs another form of forwarding, but it occurs within the register file.

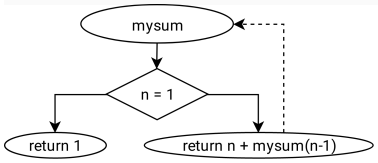
Instruction	ALUOp	Instruction operation	Opcode field	Desired ALU action	ALU control input
LDUR	00	load register	XXXXXXXXXXX	add	0010
STUR	00	store register	XXXXXXXXXXX	add	0010
CBZ	01	compare and branch on zero	XXXXXXXXXXX	pass input b	0111
R-type	10	ADD	10001011000	add	0010
R-type	10	SUB	11001011000	subtract	0110
R-type	10	AND	10001010000	AND	0000
R-type	10	ORR	10101010000	OR	0001



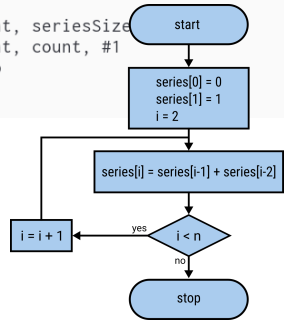

```
if isExecutionHazardFromN then
  forwardA <= "10";
elsif isMemoryHazardFromN then
  forwardA <= "01";
else
  forwardA <= "00";
end if;

if isExecutionHazardFromM then
  forwardB <= "10";
elsif isMemoryHazardFromM then
  forwardB <= "01";
else
  forwardB <= "00";
end if;
```

```
function isExecutionHazardFromN return boolean
  return exMemRegWrite = '1' and
    exMemD /= 31 and
    exMemD = n;

function isExecutionHazardFromM return boolean
  return exMemRegWrite = '1' and
    exMemD /= 31 and
    exMemD = m;

function isMemoryHazardFromN return boolean
  return memWbrRegWrite = '1' and
    memWbrD /= 31 and
    not (exMemRegWrite = '1' and
         exMemD /= 31 and
         exMemD = n) and
    memWbrD = n;

function isMemoryHazardFromM return boolean
  return memWbrRegWrite = '1' and
    memWbrD /= 31 and
    not (exMemRegWrite = '1' and
         exMemD /= 31 and
         exMemD = m) and
    memWbrD = m;
```

```
#define n x0
#define return x0
#define recursiveResult x11

    .global mysum
mysum:
    // if n == 1, branch to "baseCase"
    cmp n, 1
    b.eq baseCase

    // push n and lr to stack
    sub sp, sp, #(8*2)
    str n, [sp, #0]
    str x30, [sp, #8]

    // recursive function call
    sub n, n, 1
    bl mysum
    mov recursiveResult, return

    // pop n and lr from stack
    ldr x30, [sp, #8]
    ldr n, [sp, #0]
    add sp, sp, #(8*2)

    // return n + mysum(n-1)
    add return, n, recursiveResult
    ret

baseCase:
    mov return, 1
    ret
```

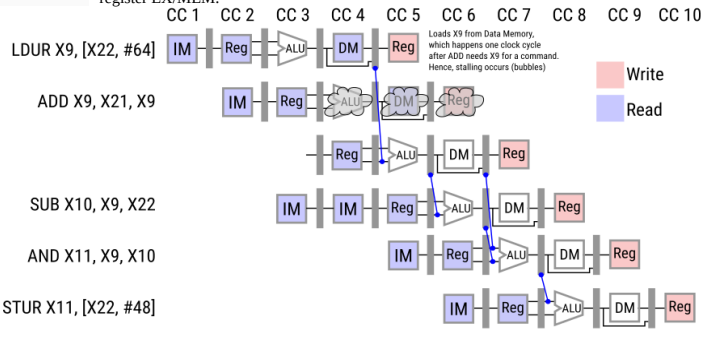



```
#define seriesSize x0
#define seriesPointer x1

#define fibonacciNumber x11
#define count x12
#define backTwo x13

    .global fibonacci
fibonacci:
    mov fibonacciNumber, 0
    stur fibonacciNumber, [seriesPointer, #0]
    add seriesPointer, seriesPointer, #8

    mov fibonacciNumber, 1
    stur fibonacciNumber, [seriesPointer, #0]
    add seriesPointer, seriesPointer, #8

    mov count, 2

loop:
    ldur backTwo, [seriesPointer, #-16]
    add fibonacciNumber, fibonacciNumber, backTwo
    stur fibonacciNumber, [seriesPointer, #0]
    add seriesPointer, seriesPointer, #8

    cmp count, seriesSize
    add count, count, #1
    blt loop

    br lr
```



Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

Ex Hazard: This case forwards the result from the previous instruction to either input of the ALU. If the previous instruction is going to write to the register file, and the write register number matches the read register number of ALU inputs A or B, provided it is not register 31, then steer the multiplexor to pick the value instead from the pipeline register EX/MEM.



1.5 [4] <COD §1.6> Consider three different processors P1, P2, and P3 executing the same instruction set. P1 has a 3 GHz clock rate and a CPI of 1.5. P2 has a 2.5 GHz clock rate and a CPI of 1.0. P3 has a 4.0 GHz clock rate and has a CPI of 2.2.

a. Which processor has the highest performance expressed in instructions per second?
b. If the processors each execute a program in 10 seconds, find the number of cycles and the number of instructions.
c. We are trying to reduce the execution time by 30% but this leads to an increase of 20% in the CPI. What clock rate should we have to get this time reduction?

Problem 1.5

Answer:

	Clock rate	CPI	IC
P1	3GHz	1.5	Same
P2	2.5GHz	1.0	Same
P3	4.0GHz	2.2	Same

a) Instructions per second = Clock rate / CPI
Instructions per second (P1) = 3GHz / 1.5 = 2 X 10^9 = 2000 MIPS (million instructions per second)
Instructions per second (P2) = 2.5GHz / 1.0 = 2.5 X 10^9 = 2500 MIPS
Instructions per second (P3) = 4.0GHz / 2.2 = 1.818 X 10^9 = 1818 MIPS
From this sense, P2 has the highest performance in terms of instructions per second.

b) If the total execution time is 10 seconds, then
Number of cycles (P1) = 10 X 3GHz = 3 x 10^{10} cycles
Number of instructions (P1) = Number of cycles / CPI = $3x10^{10}/1.5 = 2x10^{10}$
cycles (P2) = 10 X 2.5GHz = 2.5 x 10^{10} cycles
Number of instructions (P2) = Number of cycles / CPI = $2.5x10^{10}/1.0 = 2.5x10^{10}$
cycles (P3) = 10 X 4.0GHz = 4 x 10^{10} cycles
Number of instructions (P3) = Number of cycles / CPI = $4x10^{10}/2.2 = 1.818x10^{10}$

c) If we reduce the execution time by 30% while increase the CPI by 20%,
Execution time (old) = IC (old) x CPI (old) / Clock Rate(old)
Execution time (new) = IC (new) x CPI (new) / Clock Rate(new)

$$\frac{\text{Execution time (old)}}{\text{Execution time (new)}} = \frac{\text{IC (old) x CPI (old) / Clock Rate(old)}}{\text{IC (new) x CPI (new) / Clock Rate(new)}}$$

IC (old) and IC (new) is the same, so it is cancelled out, we get

$$\frac{\text{Execution time (old)}}{\text{Execution time (new)}} = \frac{\text{CPI (old) x Clock Rate(new)}}{\text{CPI (new) x Clock Rate(old)}}$$

Execution time (old) / Execution time (new) = 1/0.7 = 1.43
CPI (new) /CPI (old) = 1.2

$$1.43 = \frac{\text{Clock Rate(new)}}{1.2 \text{ x Clock Rate(old)}}$$

So clock rate (new) = 1.42 x 1.2 x Clock rate (old) = 1.714 x Clock Rate (old)
Clock rate new (P1) = 1.714 x 3GHz = 5.143 GHz
Clock rate new (P2) = 1.714 x 2.5GHz = 4.286 GHz
Clock rate new (P3) = 1.714 x 4GHz = 6.857 GHz

1.11 The results of the SPEC CPU2006 bzip2 benchmark running on an AMD Barcelona has an instruction count of 2.389E12, an execution time of 750 s, and a reference time of 9650 s.

1.11.1 [5] <COD §§1.6, 1.9> Find the CPI if the clock cycle time is 0.333 ns.

1.11.2 [5] <COD §1.9> Find the SPECratio.

1.11.3 [5] <COD §§1.6, 1.9> Find the increase in CPU time if the number of instructions of the benchmark is increased by 10% without affecting the CPI.

1.11.4 [5] <COD §§1.6, 1.9> Find the increase in CPU time if the number of instructions of the benchmark is increased by 10% and the CPI is increased by 5%.

Problem 1.11.1 – 1.11.4

Answer:

1.11.1 Execution time = IC X CPI X Clock cycle time
CPI = Execution time / (IC X Clock cycle time) = 750 / (2.389E12 X 0.333ns) = 0.943

1.11.2 SPECratio = Reference time / Execution time = 9650/750 = 12.87

1.11.3 If the IC increases by 10%, while CPI and clock cycle time is the same, then the CPU time is also increased by 10%.

1.11.4 If the IC increases by 10%, and CPI increases by 5%, and clock cycle time is the same, then
Execution time (new) = IC(new) x CPI(new) x Clock cycle time
= 1.1 x IC(old) x 1.05 x CPI(old) x Clock cycle time
= 1.155 Execution time (old)
So the execution time is increased by 15.5%.

1.12 COD Section 1.10 (Fallacies and pitfalls) cites as a pitfall the utilization of a subset of the performance equation as a performance metric. To illustrate this, consider the following two processors. P1 has a clock rate of 4 GHz, average CPI of 0.9, and requires the execution of 5.0E9 instructions. P2 has a clock rate of 3 GHz, an average CPI of 0.75, and requires the execution of 1.0E9 instructions.

1.12.1 [5] <COD §§1.6, 1.10> One usual fallacy is to consider the computer with the largest clock rate as having the highest performance. Check if this is true for P1 and P2.

Problem 1.12.1

Answer:

	Clock rate	CPI	IC
P1	4GHz	0.9	5.0E9
P2	3GHz	0.75	1.0E9

Execution time = IC X CPI / clock rate

Execution time (P1) = 5.0E9 x 0.9 / 4GHz = 1.125 (seconds)
Execution time (P2) = 1.0E9 x 0.75 / 3GHz = 0.25 (second)

So even though P1 has a faster clock rate (4GHz vs 3GHz), but it needs to execute 5 times the instructions as that of P2, so overall P2 has a better performance.

1.13 Another pitfall cited in COD Section 1.10 (Fallacies and pitfalls) is expecting to improve the overall performance of a computer by improving only one aspect of the computer. Consider a computer running a program that requires 250 s, with 70 s spent executing FP instructions, 85 s executed L/S instructions, and 40 s spent executing branch instructions.

1.13.1 [5] <COD §1.10> By how much is the total time reduced if the time for FP operations is reduced by 20%?

1.13.2 [5] <COD §1.10> By how much is the time for INT operations reduced if the total time is reduced by 20%?

1.13.3 [5] <COD §1.10> Can the total time can be reduced by 20% by reducing only the time for branch instructions?

Problem 1.13

Answer:
Total time 250 s; Floating point (FP), 70 s; Load/Store (LS), 85 s; Branch (B), 40 s; Integer(INT), 55s

1.13.1
If FP is reduced by 20%, then the new FP time is 70 x 80% = 56 s
The new total time is 56 + 85 + 40 + 55 = 236 seconds
The speedup = 250 / 236 = 1.05

1.13.2
If the total time is reduced by 20%, then it is 250 x 20% = 50 s. INT was 55 s in total, minus 50 s, so now INT need to be finished in only 5 s.

1.13.3
If the total time is reduced by 20%, then it is 250 x 20% = 50 s. Branch only takes 40 s in total, so the answer is no, we cannot reduce the total time by 20% by reducing only the time for branch instructions.