1. ***I Amdahl-ighted with Tradeoffs (16 points):*** For the following design decisions, list a benefit and a drawback of the decision. Be brief, 1-2 sentences per benefit or drawback – but be specific in your answers. The first one is done for you as an example.

> **Example:**
>
> ***Design Decision***: Choosing to use 32 registers instead of 64 registers in an ISA.
> *Benefit*: Fewer bits required to specify a register address and simpler register file implementation
> *Drawback*: More register spilling may be required (i.e. more loads and possibly stores)

a. ***Design Decision***: Using more complex addressing modes, like memory increment.
Benefit:
Potential to ↓ IC

Drawback:
more complex instructions must be implemented → can impact CPI or CT

b. ***Design Decision***: Using the single cycle datapath we covered in class instead of the multicycle datapath we covered in class.
Benefit:
CPI is 1, control is simpler

Drawback:
CT will be set by worst case latency instr

c. ***Design Decision***: Using a register-memory machine instead of a load-store machine.
Benefit:
potential to ↓ IC since instrs can directly access memory

Drawback
complexity in decoding and operand access → implementation is more challenging

d. ***Design Decision***: Using RISC instead of using CISC.
Benefit:
fewer instructions to implement in machine, pipelining + parallelism easier

Drawback:
larger instruction foot print, larger IC

2. ***You Make the Call (12 points)***: For the following proposed design changes, indicate the impact on each component of execution time (i.e. CPI, cycle time, instruction count). Assume that we would optimize the datapath/control in cases where the change impacts machine organization. For each component, indicate whether the change ***could increase*** that component, ***could decrease*** that component, or whether the change will not impact the component and it ***will stay the same***. You should only circle ***one*** of these alternatives for each component of execution time. Assume the use of the **multicycle datapath**.

   a. We make use of a new low resistance material to reduce the latency of wires in our processor. The ISA does not change.

CPI:	could increase	could decrease	will stay the same
Cycle time:	could increase	could decrease	will stay the same
Instruction count:	could increase	could decrease	will stay the same

   b. We use two-address code instead of three-address code in the MIPS ISA. We still run the same programs, but of course must recompile for the new ISA.

more space for other fields, but can ↑ IC

CPI:	could increase	could decrease	will stay the same
Cycle time:	could increase	could decrease	will stay the same
Instruction count:	could increase	could decrease	will stay the same

   c. We extend the MIPS ISA with an instruction that can perform (a*b)+(c*d) which is currently implemented with three instructions: two multiplies and an add. We still run the same programs, but of course must recompile for the new ISA.

CPI:	could increase	could decrease	will stay the same
Cycle time:	could increase	could decrease	will stay the same
Instruction count:	could increase	could decrease	will stay the same

   d. We switch to a new compiler – one that reduces the number of loads and stores in our applications. We do not change the ISA or machine organization.

CPI:	could increase	could decrease	will stay the same
Cycle time:	could increase	could decrease	will stay the same
Instruction count:	could increase	could decrease	will stay the same

3. ***Don't Blank Out (14 points)***: The following question assumes the use of the small subset of MIPS we covered in class. We will examine the performance of a processor on a particular application, which has the following breakdown: 15% beq and bne instructions, 20% loads, 10% stores, 5% jumps, and 50% R-types. Our processor has a 2 GHz clock. The application executes 1 billion instructions. For the following questions, fill in all blanks to get full credit.

   a. First, find the execution time of the application on this processor, assuming it uses the **single cycle datapath** we covered in class.

   $1e9 \times 1.0 \times 0.5e^{-9}$

   Instruction Count = 1e9

   CPI = 1.0

   Cycle Time = 0.5 ns

   ET = 0.5 s

   b. Second we will try a new processor, based on the **multicycle datapath** we covered in class.

   $CPI = .2\times5 + (.1+.5)\times4 + (.05+.15)\times3$

   $1.0 + 2.4 + 0.6 = 4$

   Instruction Count = 1e9

   CPI = 4.0

   Cycle Time = 0.5ns

   ET = 2s

   c. Third, we will try the **multicycle datapath** again but with a change to the MIPS ISA. We will implement memory indirect addressing using the memory indirect load (*mil*) instruction, which has the specification R[rt]=M[M[R[rs]+SE[I]]]. It will take 6 cycles to execute. We observe that 25% of all loads in our program write a value to the register file that is used only once – the value is used by a dependent load instruction that is not part of that original 25%. In other words, these loads are used to perform memory indirect addressing – and pairs of loads like this can be replaced by a single *mil* instruction. All other instructions are not impacted by this change. The clock rate is not impacted by this change.

   $CPI = \frac{5}{95}\times6 + \frac{10}{95}\times5 + \frac{10+50}{95}\times4 + \frac{15+5}{95}\times3$

   $= \frac{30+50+240+60}{95} = \frac{380}{95} = 4$

   Instruction Count = 9.5e8

   CPI = 4.0

   Cycle Time = 0.5ns

   ET = 1.9s

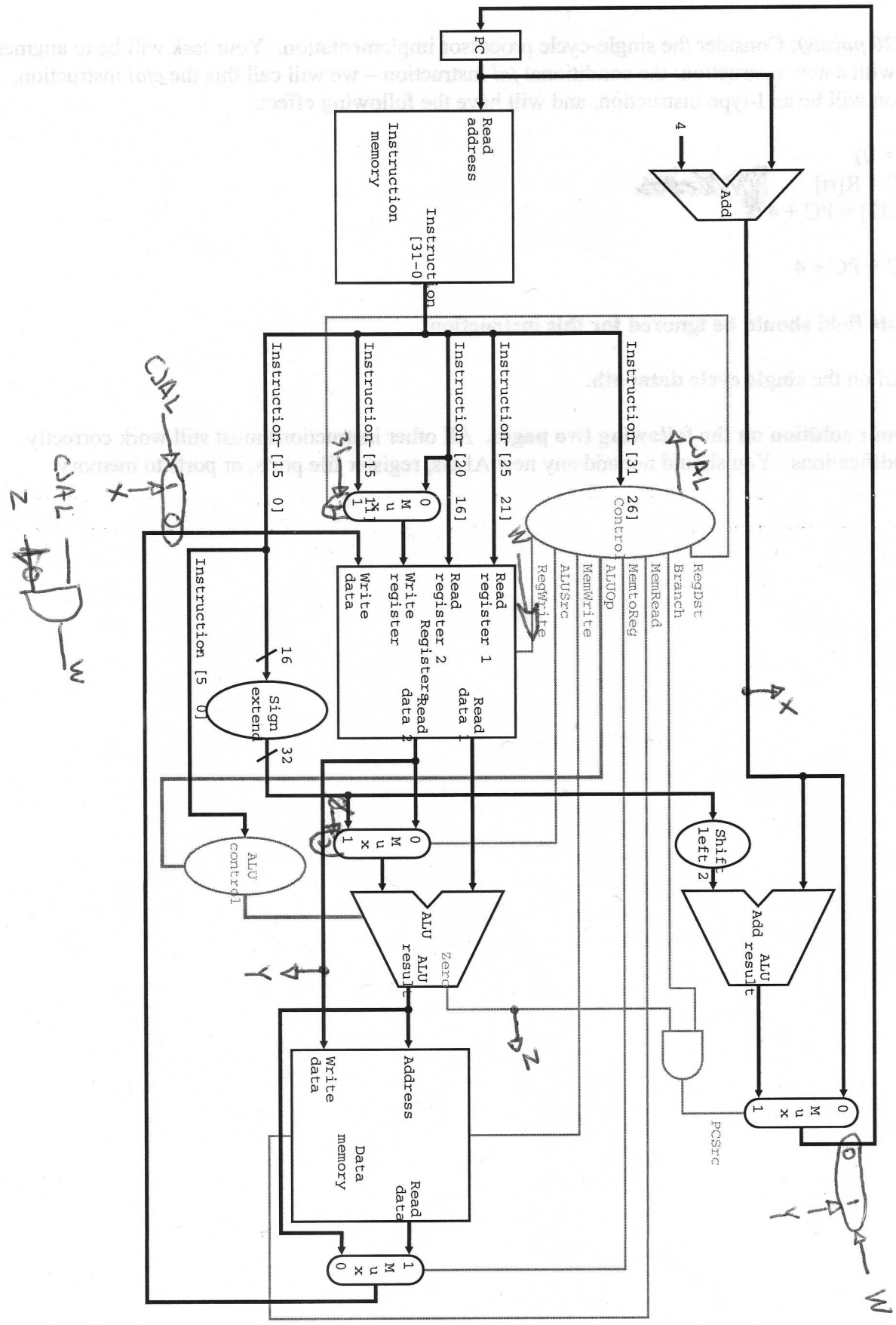
4. ***C-Jal Later (20 points)***: Consider the single-cycle processor implementation. Your task will be to augment this datapath with a new instruction: the conditional *jal* instruction – we will call this the *cjal* instruction. This instruction will be an I-type instruction, and will have the following effect:

```
If (R[rs] != 0)
        PC = R[rt]
        R[31] = PC + 4
else
        PC = PC + 4
```

The immediate field should be ignored for this instruction.

Implement *cjal* on the **single cycle datapath**.

Implement your solution on the following two pages. All other instructions must still work correctly after your modifications. You should not add any new ALUs, register file ports, or ports to memory.


PC
Read address
Instruction memory
Instruction [31-0]
4
Add
Instruction [31-26]
Instruction [25-21]
Instruction [20-16]
Instruction [15-0]
Instruction [5-0]
Control
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
16
Sign extend
32
Shift left 2
Add
ALU result
ALU
zero
ALU control
Address
Data memory
Read data
Write data
M u x
PCSrc
CJAL
X
Y
Z
W

Main Controller

Input or Output	Signal Name	R-format	lw	sw	Beq	JAL
Inputs	Op5	0	1	1	0	0
	Op4	0	0	0	0	1
	Op3	0	0	1	0	0
	Op2	0	0	0	1	1
	Op1	0	1	1	0	0
	Op0	0	1	1	0	1
Outputs	RegDst	1	0	X	X	10
	ALUSrc	0	1	1	0	10
	MemtoReg	0	1	X	X	X
	RegWrite	1	1	0	0	0
	MemRead	0	1	0	0	0
	MemWrite	0	0	1	0	0
	Branch	0	0	0	1	0
	ALUOp1	1	0	0	0	0
	ALUOp0	0	0	0	1	1
	JAL	0	0	0	0	1

Any thing here

ALU Controller

Opcode	**ALUOp**	**instruction**	**function**	**ALU Action**	**ALUCtrl**
Lw	00	load word	XXXXXX	add	0010
Sw	00	store word	XXXXXX	add	0010
Beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	Add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	SLT	101010	SLT	0111

5. ***This Question is De-lay-Mux One (18 points)***: Assume for the rest of this problem that all logic gates have the following delays:

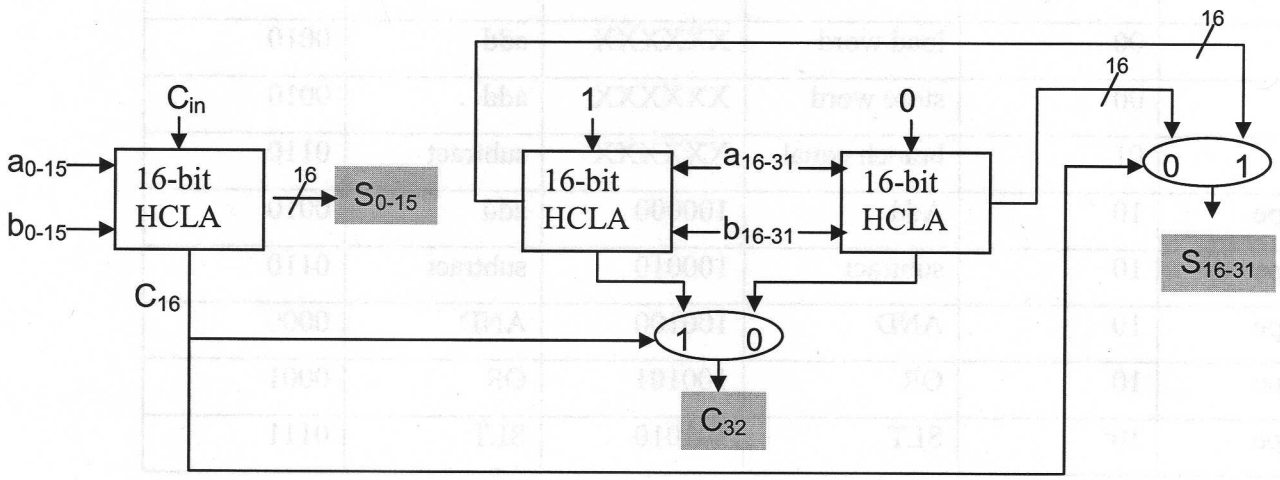
Fan In	Delay
1	T
2	T
3	2T
4	3T
5 or more	5T

So a 2-input AND gate would have delay T and a 4-input OR gate would have delay 3T. Further assume that mux's have delay 4T.

We are going to slightly modify the design of the full adder from what we assumed in class. We will still use two two-input xor's to implement the Sum logic:

Sum = Cin ^ (A ^ B)

We will create a 32-bit adder out of these full adders. We will use the 4-bit CLA that we covered in class as the basic building block of this design, and we will use it (as we did in class) to make 16-bit hierarchical CLAs (HCLA). But instead of connecting these in series to make a 32-bit adder, we will use carry select to speed up the 32-bit adder. The design will look as follows:

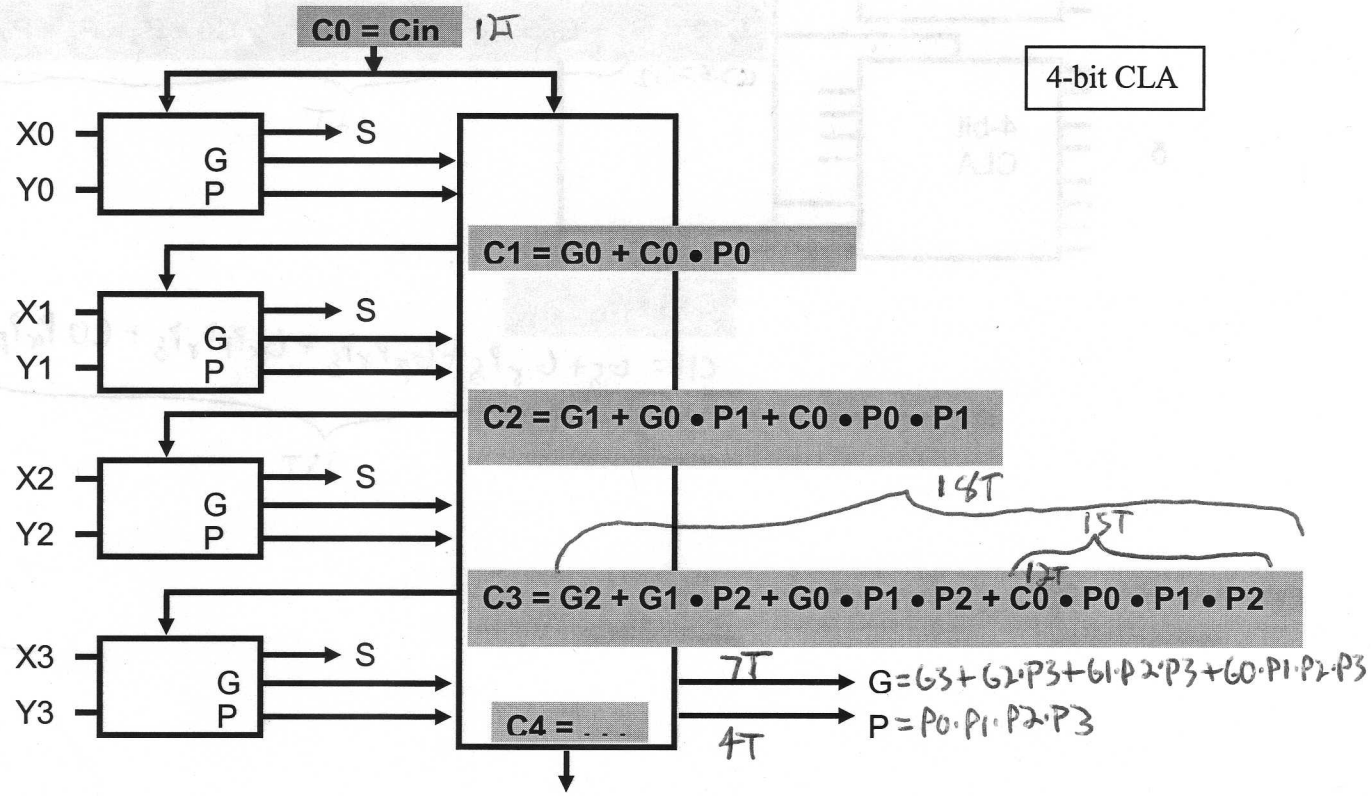


Your task is to find the maximal delay of this design – i.e. find the delays of $S_{0\text{-}15}$, C_{32}, and $S_{16\text{-}31}$ – the maximal delay of these three outputs will be the maximal delay of the design. Fill in the values in the table on the following page to receive full credit (and to help with possible partial credit).

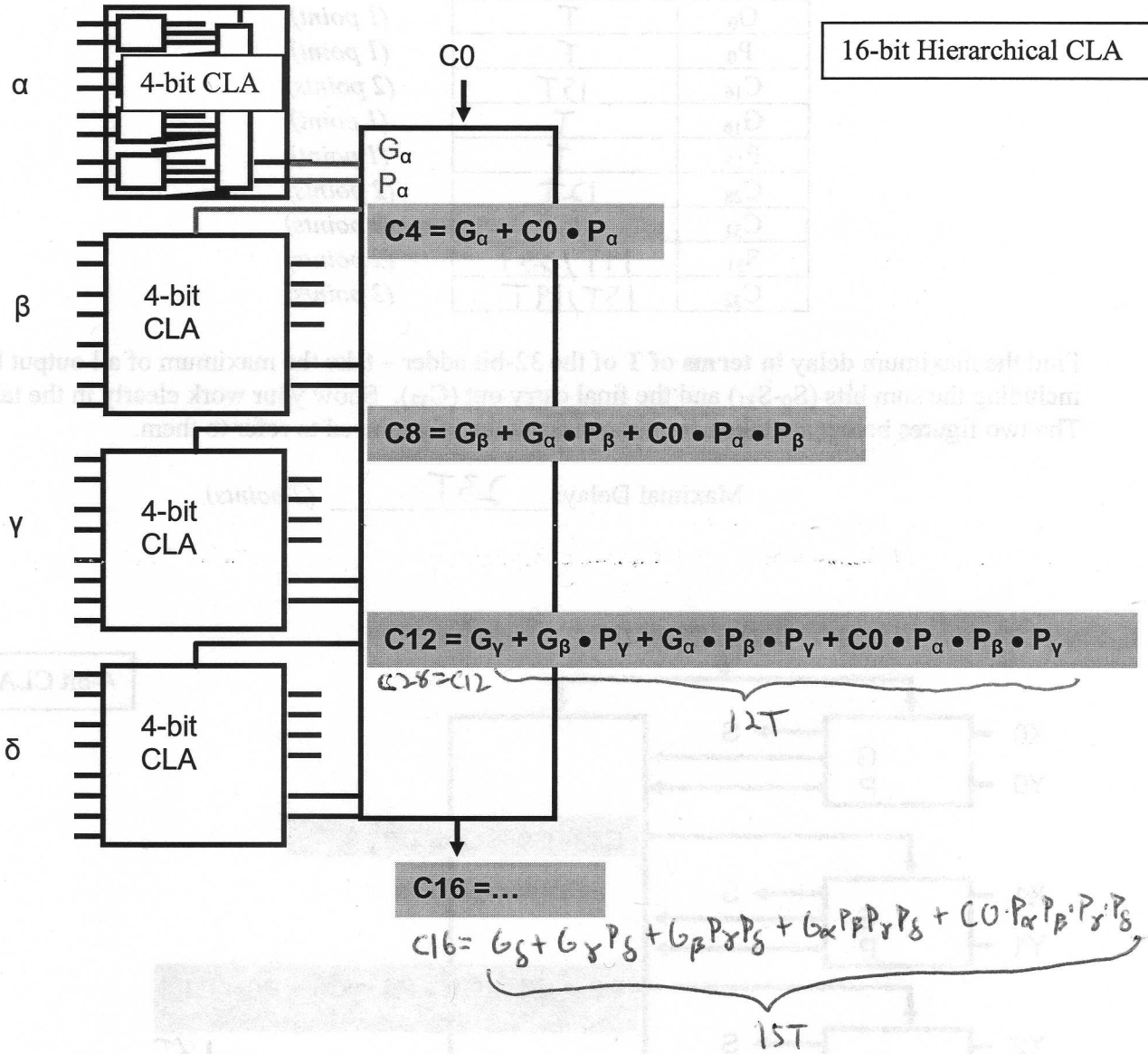
Output	Delay	
G_0	T	*(1 point)*
P_0	T	*(1 point)*
C_{16}	15T	*(2 points)*
G_{16}	T	*(1 point)*
P_{16}	T	*(1 point)*
C_{28}	12T	*(2 points)*
C_{31}	16T	*(2 points)*
S_{31}	19T/23T	*(2 points)*
C_{32}	15T/19T	*(2 points)*

Find the maximum delay **in terms of T** of the 32-bit adder – take the maximum of all output bits – including the sum bits (S_0-S_{31}) and the final carry out (C_{32}). Show your work clearly in the table above. The two figures below are taken from the class notes, if you need to refer to them.

Maximal Delay: 23T *(2 points)*



16-bit Hierarchical CLA



6. ***Multiadd MADness (20 points)***: You have noticed that your most commonly used application only requires 8 bits of precision for most operations, and that most of these operations are just add instructions. This motivates you to look at compressing your register storage to cleverly save on storage space, packing four 8 bit values into a single 32 bit register. So for example, register *$t0* may currently contain the following 32 bits:

01011101111000011001100100011011

But we will interpret these bits as four distinct 8-bit signed values, as such:

01011101	11100001	10011001	00011011

Note that we still read out all 32-bits from the register file, but that we will want to selectively use certain bits from the 32. To manipulate these compressed registers, you implement a new add instruction – the *multiadd* (*MAD*) – which will add two registers that have been compressed with four 8 bit values to produce an output to a register that will also store four 8 bit values. So for example, if register *$t0* and *$t1* currently contain the following bits:

$t0	01011101	11100001	10011001	00011011
$t1	00000000	10000000	00010001	01100001

And then we executed *mad $t2, $t1, $t0* we would see the following result in register *$t2*:

$t2	01011101	01100001	10101010	01111100

Notice a few things about this – first, this instruction has done four independent add operations – each of which uses 8 bits. Second, the sum of 11100001 and 10000000 overflows the 8 bits of storage that we have. Note that this overflow does not impact the sum of 01011101 and 00000000. You may assume that overflows signals are handled without you worrying about them (we'll just OR all overflow flags together and let the OS sort it out – but you don't have to worry about doing that on the datapath or control).

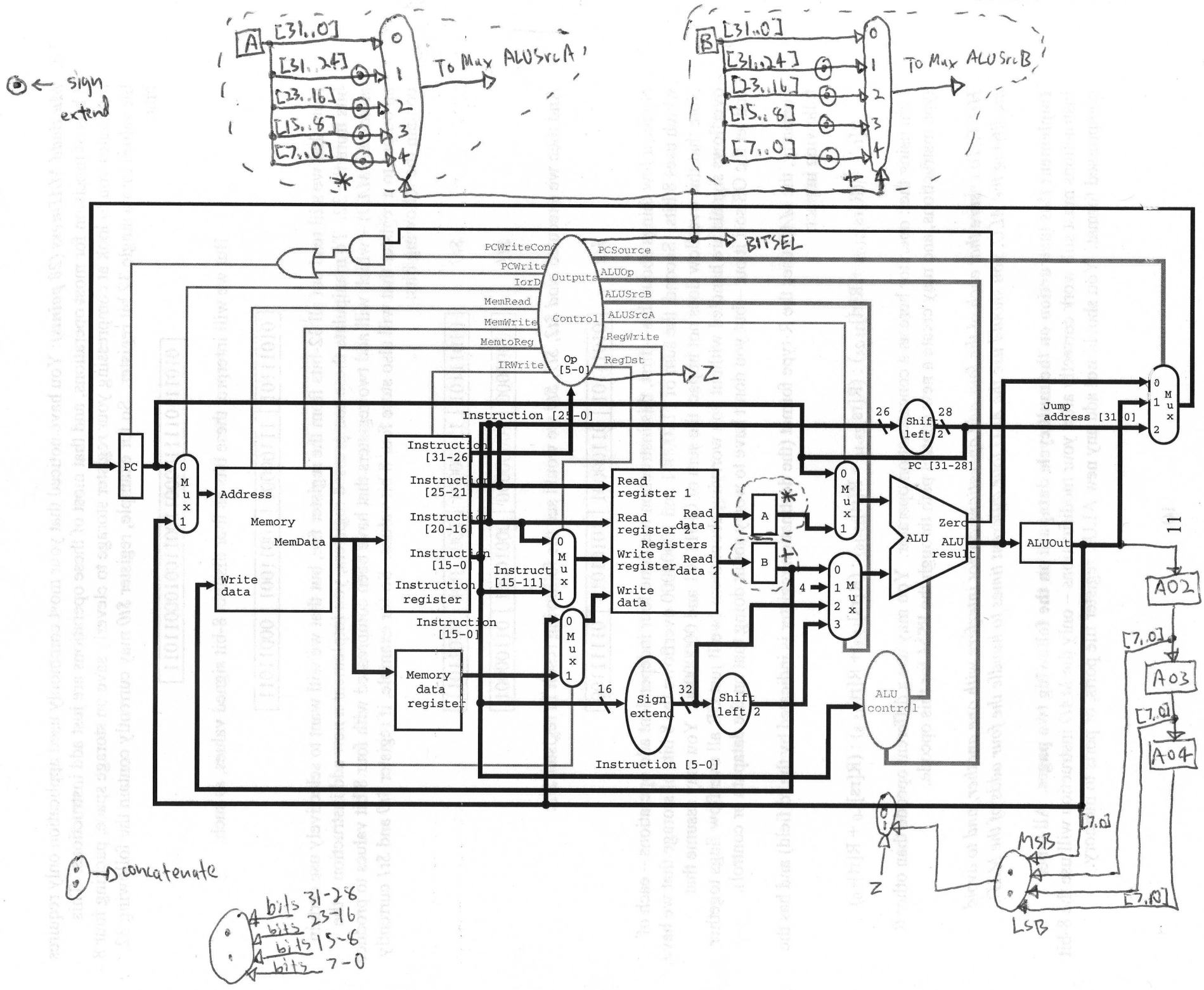
Formally, the *MAD* uses the R-type format (the destination register is indicated by the rd field) and has the following impact:

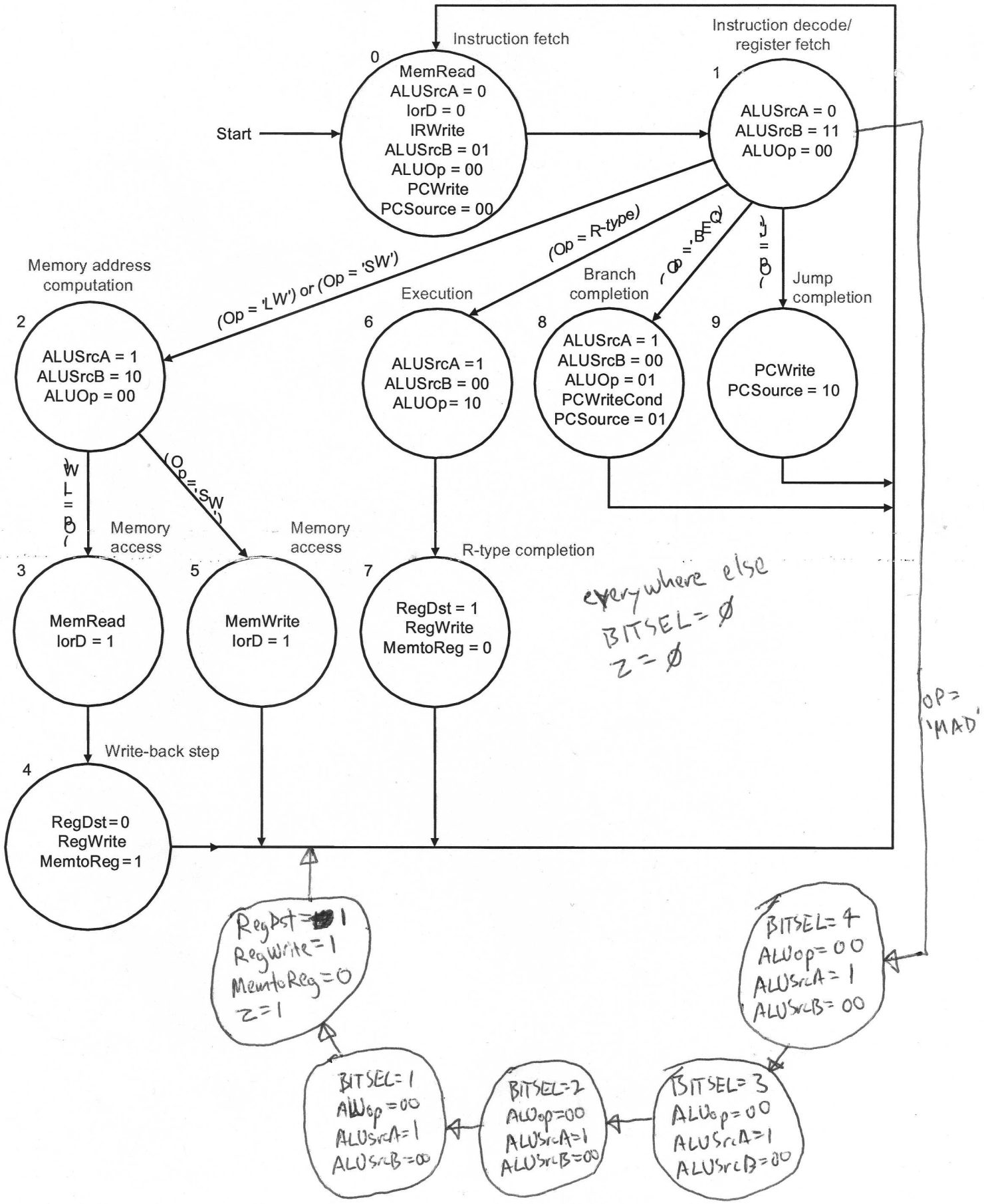
$$R[rd] = (R[rs]_{31\text{-}24} + R[rt]_{31\text{-}24}) : (R[rs]_{23\text{-}16} + R[rt]_{23\text{-}16}) : (R[rs]_{15\text{-}8} + R[rt]_{15\text{-}8}) : (R[rs]_{7\text{-}0} + R[rt]_{7\text{-}0})$$

I am using the : operator here as a concatenation operator. You may use a different opcode than other R type instructions, and may create a new control path through the FSM for this opcode.

HINT: To ensure that each of the four add operations do not interfere with one another, and to avoid modifying the ALU, you should use the ALU four different times to handle the four adds of the MAD.

Implement this instruction on the multicycle datapath on the following two pages. All other instructions must still work correctly after your modifications – only the *MAD* instruction will use the 8-bit compressed format. You should **not** add any new ALUs, register file ports, or ports to memory.


sign extend
A
[31..0]
[31..24]
[23..16]
[15..8]
[7..0]
To Mux ALUSrcA
B
To Mux ALUSrcB
BITSEL
PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Outputs
Control
Op [5-0]
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
Z
Instruction [25-0]
26
Shift left 2
28
PC [31-28]
Jump address [31 0]
PC
Address
Memory
MemData
Write data
Instruction [31-26]
Instruction [25-21]
Instruction [20-16]
Instruction [15-0]
Instruction register
Instruction [15-11]
Instruction [15-0]
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
A
B
4
ALU
Zero
ALU result
ALUOut
11
A02
A03
A04
[7..0]
Memory data register
16
Sign extend
32
Shift left 2
ALU control
Instruction [5-0]
Z
MSB
LSB
concatenate
bits 31-28
bits 23-16
bits 15-8
bits 7-0


Instruction fetch
Instruction decode/
register fetch
0
MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00
Start
1
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'JMP')
Memory address
computation
Execution
Branch
completion
Jump
completion
2
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
6
ALUSrcA =1
ALUSrcB = 00
ALUOp= 10
8
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01
9
PCWrite
PCSource = 10
(Op = 'LW')
(Op = 'SW')
Memory
access
Memory
access
R-type completion
3
MemRead
IorD = 1
5
MemWrite
IorD = 1
7
RegDst = 1
RegWrite
MemtoReg = 0
everywhere else
BITSEL = ∅
Z = ∅
OP = 'MAD'
Write-back step
4
RegDst = 0
RegWrite
MemtoReg = 1
RegDst = 1
RegWrite = 1
MemtoReg = 0
Z = 1
BITSEL = 1
ALUop = 00
ALUSrcA = 1
ALUSrcB = 00
BITSEL = 2
ALUop = 00
ALUSrcA = 1
ALUSrcB = 00
BITSEL = 3
ALUop = 00
ALUSrcA = 1
ALUSrcB = 00
BITSEL = 4
ALUop = 00
ALUSrcA = 1
ALUSrcB = 00