

<p style="text-align: center;">CS8803 Summer 2009 Language and Compilers for Embedded Systems Exam #1 June 3, 2009</p>

Close books, close notes.

Calculator is allowed.

You are allowed to bring a one page (double-sided) cheat sheet (A4 paper).

This exam is given under the Georgia Tech Honor Code System. You must observe and sign the Honor Pledge: "I have neither given nor received aid on this exam." Your print name and signature below signifies your compliance with this honor code.

Name (Print): _____ **SOLUTION** _____

Signature: _____

1. _____ (35 pts)

2. _____ (35 pts)

3. _____ (30 pts)

Total (100 pts) _____

Answer your questions as **concise** as possible.

1. (35%) Please answer the questions as concise as possible.

1.1. (5%) What is the difference when we calculate Instruction Level Parallelism (ILP) and Instruction Per Cycle (IPC)?

IPC is calculated based on the configuration of a specific machine while ILP represents the property of a program and is independent of the underlying hardware. ILP assumes a unit-cycle operation, infinite resources and perfect frontend. In other words, ILP is the upper bound of attainable IPC.

1.2. (4%) What is a “Superpipelined” machine?

A superpipelined machine is simply a microprocessor with a deeper pipeline compared to its contemporary processors.

1.3. (4%) Explain two types of false dependency.

- **Write-after-Write (WAW) or output dependency**
- **Write-after-Read (WAR) or anti-dependency**

1.4. (6%) Give two major roadblocks of finding higher ILP.

- **Data Dependency**
 - o **RAW, WAW, WAR**
- **Control Dependency**
 - o **Branch**
 - **Conditional**
 - **Indirect**

- 1.5. (5%) Briefly explain dataflow execution model (or so-called dependence architecture in Rau and Fisher's article).

The compiler explicitly indicates the dependences that exist between operations and gives the information to the hardware. At run-time, the hardware schedules the execution of an instruction upon the availability of its input operands.

- 1.6. (6%) Give two reasons as to why a VLIW design could be a better option for an embedded processor than a superscalar machine?

A VLIW could simplify the hardware by delegating instruction scheduling to the compiler. As such, it could achieve similar performance without increasing the operating frequency or power consumption, leading to higher energy efficiency.

- 1.7. (5%) Explain the function of a "stop bit" in VLIW encoding.

A "stop bit" is used to delimit the end of the current instruction and separate dependent instructions. Instructions in-between stop bits belong to the same instruction group.

2. (35%) Instruction-Level Parallelism (ILP).

```

LDW   R1 = 0 (R10)
LDW   R2 = 4 (R10)
LDW   R3 = 8 (R10)
MPYL  R4 = R2, R3
MPYL  R5 = R4, R3
BEQ   R3, R4, LL1 ; branch if ==
LDW   R1 = 12 (R10)
ADD   R7 = R1, R2
BGT   R7, R3, LL3 ; branch if >
GOTO  LL2

LL1:
LDW   R2 = 28 (R10)
ADD   R3 = R1, R5
ADD   R4 = R2, R3
MPYL  R7 = R4, R1

LL2:
LDW   R1 = 16 (R10)
MPYL  R6 = R4, R4
ADD   R7 = R6, R1

LL3:
STW   0 (R11) = R4
STW   8 (R11) = R6
STW   12 (R11) = R7
    
```

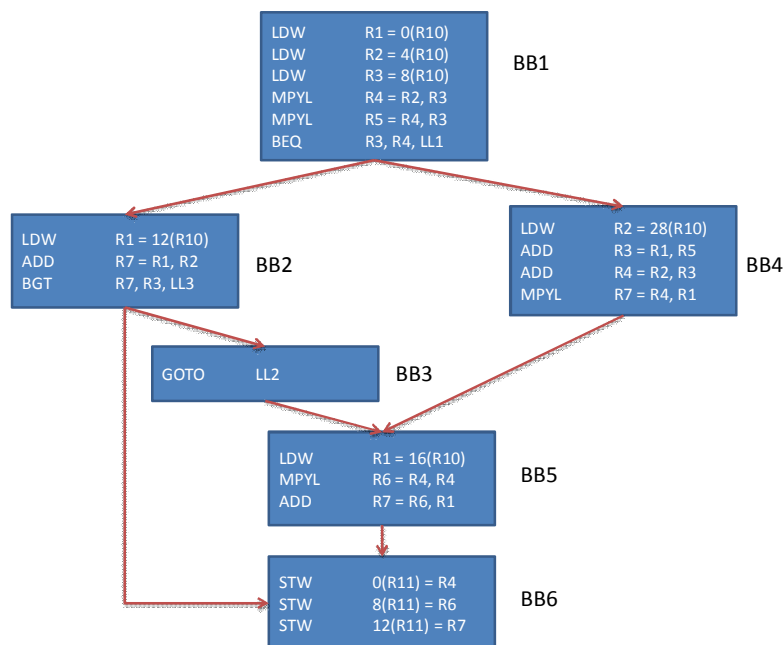
Instruction Format
=====

OP A = B, C
A: Destination
B and C: Source operands

LDW A = offset(B)
A: Destination
B: Base address

STW offset(B) = A
A: Source operand
B: Base address

2.1. (5%) How many basic blocks are in the code above? Please draw the complete control-flow graph (CFG).



There are 6 basic blocks.

```

LDW R1 = 0 (R10)
LDW R2 = 4 (R10)
LDW R3 = 8 (R10)
MPYL R4 = R2, R3
MPYL R5 = R4, R3
BEQ R3, R4, LL1 ; branch if ==
LDW R1 = 12 (R10)
ADD R7 = R1, R2
BGT R7, R3, LL3 ; branch if >
GOTO LL2
LL1:
LDW R2 = 28 (R10)
ADD R3 = R1, R5
ADD R4 = R2, R3
MPYL R7 = R4, R1
LL2:
LDW R1 = 16 (R10)
MPYL R6 = R4, R4
ADD R7 = R6, R1
LL3:
STW 0 (R11) = R4
STW 8 (R11) = R6
STW 12 (R11) = R7

```

Instruction Format
=====

OP A = B, C
A: Destination
B and C: Source operands

LDW A = offset(B)
A: Destination
B: Base address

STW offset(B) = A
A: Source operand
B: Base address

2.2. (10%) What is the ILP if we relax the restriction with a perfect branch predictor? Given the conditional branch instruction is always taken. (The code above is the same, duplicated here for your convenience.)

1	LDW R1 = 0 (R10)	LDW R2 = 4 (R10)	LDW R3 = 8 (R10)
2	MPYL R4 = R2, R3		
3	MPYL R5 = R4, R3	BEQ R3, R4, LL1	LDW R2 = 28 (R10)
4	ADD R3 = R1, R5		
5	ADD R4 = R2, R3		
6	MPYL R7 = R4, R1	MPYL R6 = R4, R4	STW 0 (R11) = R4
7	LDW R1 = 16 (R10)	STW 8 (R11) = R6	
8	ADD R7 = R6, R1		
9	STW 12 (R11) = R7		

ILP = 16/9 = 1.78

```

LDW R1 = 0 (R10)
LDW R2 = 4 (R10)
LDW R3 = 8 (R10)
MPYL R4 = R2, R3
MPYL R5 = R4, R3
BEQ R3, R4, LL1 ; branch if ==
LDW R1 = 12 (R10)
ADD R7 = R1, R2
BGT R7, R3, LL3 ; branch if >
GOTO LL2
LL1:
LDW R2 = 28 (R10)
ADD R3 = R1, R5
ADD R4 = R2, R3
MPYL R7 = R4, R1
LL2:
LDW R1 = 16 (R10)
MPYL R6 = R4, R4
ADD R7 = R6, R1
LL3:
STW 0 (R11) = R4
STW 8 (R11) = R6
STW 12 (R11) = R7

```

Instruction Format
=====

OP A = B, C
A: Destination
B and C: Source operands

LDW A = offset(B)
A: Destination
B: Base address

STW offset(B) = A
A: Source operand
B: Base address

2.3. (10%) Continuing from the assumption of the last problem, if you can perform register renaming to remove false dependencies, (1) what is the ILP? (2) what is the largest issuable number of instructions in one cycle? (The code above is the same as before.)

1	LDW R1 = 0 (R10)	LDW R2 = 4 (R10)	LDW R3 = 8 (R10)	LDW T2 = 28 (R10)	LDW T1 = 16 (R10)
2	MPYL R4 = R2, R3				
3	MPYL R5 = R4, R3	BEQ R3, R4, LL1			
4	ADD T3 = R1, R5				
5	ADD T4 = T2, T3				
6	MPYL R7 = T4, R1	MPYL R6 = T4, T4	STW 0 (R11) = T4		
7	ADD R7 = R6, T1	STW 8 (R11) = R6			
8	STW 12 (R11) = R7				

$$\text{ILP} = 16 / 8 = 2$$

Largest issuable number of instructions in one cycle = 5

```

LDW   R1 = 0 (R10)
LDW   R2 = 4 (R10)
LDW   R3 = 8 (R10)
MPYL  R4 = R2, R3
MPYL  R5 = R4, R3
BEQ   R3, R4, LL1 ; branch if ==
LDW   R1 = 12 (R10)
ADD   R7 = R1, R2
BGT   R7, R3, LL3 ; branch if >
GOTO  LL2

LL1:
LDW   R2 = 28 (R10)
ADD   R3 = R1, R5
ADD   R4 = R2, R3
MPYL  R7 = R4, R1

LL2:
LDW   R1 = 16 (R10)
MPYL  R6 = R4, R4
ADD   R7 = R6, R1

LL3:
STW   0 (R11) = R4
STW   8 (R11) = R6
STW   12 (R11) = R7

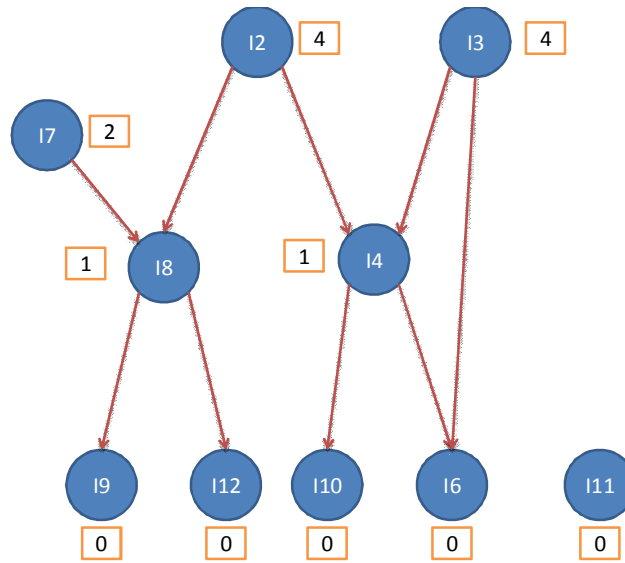
```

Instruction Latency	
=====	
LDW	2 cycles
MPYL	3 cycles
BEQ/GT	1 cycle
ADD	1 cycle
GOTO	1 cycle
STW	1 cycle

2.4. (10%) Keep the same instructions above (i.e., no renaming), and assume the “taken” probabilities of BEQ and BGT are 5% and 95%, respectively. Please perform trace scheduling of the most frequently taken path. Then, based on “List scheduling” algorithm using latency numbers shown in the above, show the final instruction sequence in the trace. What you also need to do is to “eliminate the dead code” but you don’t need to worry about the fix-up or recovery code for the not-taken paths. Dead code is referred to those instructions whose destination register results were never used by any subsequent instruction. Note that, branches are not dead code.

In the figure of the Prob 2.1, BB1->BB2->BB6 will be a trace.

Original trace	After dead code elimination
I1: LDW R1 = 0 (R10)	I2: LDW R2 = 4 (R10)
I2: LDW R2 = 4 (R10)	I3: LDW R3 = 8 (R10)
I3: LDW R3 = 8 (R10)	I4: MPYL R4 = R2, R3
I4: MPYL R4 = R2, R3	I6: BEQ R3, R4, LL1
I5: MPYL R5 = R4, R3	I7: LDW R1 = 12 (R10)
I6: BEQ R3, R4, LL1	I8: ADD R7 = R1, R2
I7: LDW R1 = 12 (R10)	I9: BGT R7, R3, LL3
I8: ADD R7 = R1, R2	I10: STW 0 (R11) = R4
I9: BGT R7, R3, LL3	I11: STW 8 (R11) = R6
I10: STW 0 (R11) = R4	I12: STW 12 (R11) = R7
I11: STW 8 (R11) = R6	
I12: STW 12 (R11) = R7	



Schedule = {(I2, I3), I7, (I4, I8), (I6, I9, I10, I11, I12)}

3. (30%) Given the following code and their encoding method on the right.

<pre> LDW R1 = 0 (R10) LDW R2 = 4 (R10) LL0: ADD R3 = R1, R2 SUB R4 = R1, R2 BEQ R1, R2, LL1 LDW R1 = 8 (R10) ADD R7 = R1, R2 LL1: LDW R2 = 28 (R10) BEQ R1, R2, LL1 LL2: STW 0 (R10) = R7 LDW R7 = 0 (R10) BEQ R1, R2, LL0 </pre>	<p>Encoding Format</p> <hr/> <p>Memory and Branch Classes (R_{temp} is destination for LDW and source for STW)</p> <table border="1" style="width: 100%; border-collapse: collapse; text-align: center;"> <tr> <td style="width: 6%;">opcode</td> <td style="width: 6%;">R_{temp/left}</td> <td style="width: 6%;">R_{base/right}</td> <td style="width: 15%;">PC-Relative offset</td> </tr> <tr> <td>31</td> <td>26 25</td> <td>21 20</td> <td>16 15</td> </tr> <tr> <td colspan="4" style="text-align: right;">0</td> </tr> </table> <p>ALU Class</p> <table border="1" style="width: 100%; border-collapse: collapse; text-align: center;"> <tr> <td style="width: 6%;">opcode1</td> <td style="width: 6%;">R_{src_left}</td> <td style="width: 6%;">R_{src_right}</td> <td style="width: 6%;">R_{dest}</td> <td style="width: 6%;">Shift_size</td> <td style="width: 6%;">opcode2</td> </tr> <tr> <td>31</td> <td>26 25</td> <td>21 20</td> <td>16 15</td> <td>11</td> <td>5</td> </tr> <tr> <td colspan="6" style="text-align: right;">0</td> </tr> </table>	opcode	R _{temp/left}	R _{base/right}	PC-Relative offset	31	26 25	21 20	16 15	0				opcode1	R _{src_left}	R _{src_right}	R _{dest}	Shift_size	opcode2	31	26 25	21 20	16 15	11	5	0					
opcode	R _{temp/left}	R _{base/right}	PC-Relative offset																												
31	26 25	21 20	16 15																												
0																															
opcode1	R _{src_left}	R _{src_right}	R _{dest}	Shift_size	opcode2																										
31	26 25	21 20	16 15	11	5																										
0																															

3.1. (15%) We are interested in a compression method which split a 32-bit instruction into two parts: left 16-bit and right 16-bit, similar to CodePack. Note that, ADD and SUB has the same opcode1 but differ in opcode2. Shift_size is '00000' for non-shift instruction. Also note that, the branch target is encoded as a relative offset. Based on the encoding format above, please show the "unique encoding patterns" for the left and right 16-bit, respectively.

Let's symbolize the opcode for each instruction as below:

Instruction	OpCode1	OpCode2
LDW	LLLLLL	
STW	TTTTTT	
ADD	AAAAAA	AAAAAA
SUB	AAAAAA	SSSSSS
BEQ	BBBBBB	

Assembly Code	Machine Code	
	Upper Half (Left)	Lower Half (Right)
LDW R1 = 0 (R10)	LLLLLL 00001 01010 (P3)	0000000000000000 (P1)
LDW R2 = 4 (R10)	LLLLLL 00010 01010 (P4)	0000000000000001 (P4)
ADD R3 = R1, R2	AAAAAA 00001 00010 (P1)	00011 00000 AAAAAA (P5)
SUB R4 = R1, R2	AAAAAA 00001 00010 (P1)	00100 00000 SSSSSS (P6)
BEQ R1, R2, LL1	BBBBBB 00001 00010 (P2)	0000000000000011 (P2)
LDW R1 = 8 (R10)	LLLLLL 00001 01010 (P3)	0000000000000010 (P3)
ADD R7 = R1, R2	AAAAAA 00001 00010 (P1)	00111 00000 AAAAAA (P7)
LDW R2 = 28 (R10)	LLLLLL 00010 01010 (P4)	0000000000000111 (P8)
BEQ R1, R2, LL1	BBBBBB 00001 00010 (P2)	1111111111111111 (P9)
STW 0 (R10) = R7	TTTTTT 00111 01010 (P5)	0000000000000000 (P1)
LDW R7 = 0 (R10)	LLLLLL 00111 01010 (P6)	0000000000000000 (P1)
BEQ R1, R2, LL0	BBBBBB 00001 00010 (P2)	1111111111110111 (P10)

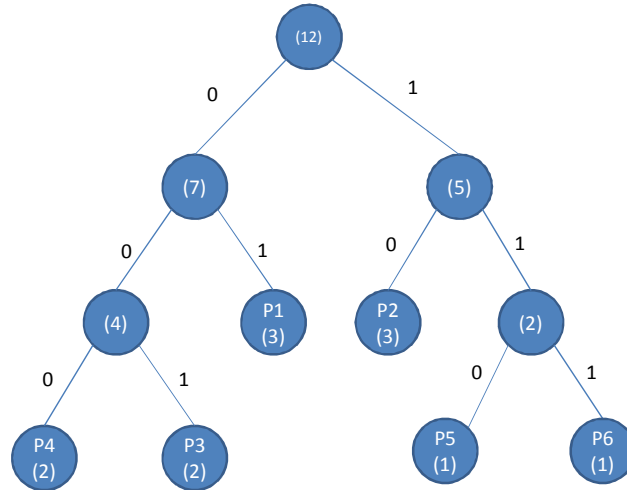
Pattern (Left 16 bits)	Count
P1: AAAAAA 00001 00010	3
P2: BBBBBB 00001 00010	3
P3: LLLLLL 00001 01010	2
P4: LLLLLL 00010 01010	2
P5: TTTTTT 00111 01010	1
P6: LLLLLL 00111 01010	1

Pattern (Right 16bits)	Count
P1: 0000000000000000	3
P2: 0000000000000011	1
P3: 0000000000000010	1
P4: 0000000000000001	1
P5: 0001100000AAAAAA	1
P6: 0010000000SSSSSS	1
P7: 0011100000AAAAAA	1
P8: 0000000000000111	1
P9: 1111111111111111	1
P10: 1111111111110111	1

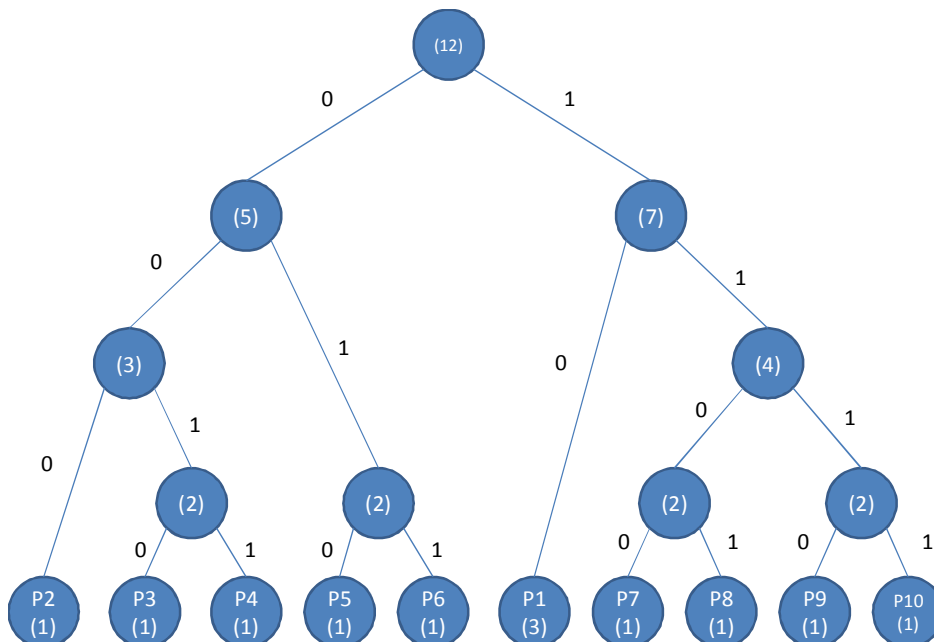
<Unique Encoding Patterns>

3.2. (15%) Using Huffman coding method discussed in class, encode the left and right separately. Also discuss the compression rates (i.e., new code size / old code size)

Huffman Coding (Left)



Huffman Coding (Right)



Pattern (Left 16 bits)	Code
P1: AAAAAA 00001 00010	01
P2:BBBBBB 00001 00010	10
P3: LLLLLL 00001 01010	001
P4: LLLLLL 00010 01010	000
P5: TTTTTT 00111 01010	110
P6: LLLLLL 00111 01010	111

Pattern (Right 16bits)	Code
P1: 0000000000000000	10
P2: 0000000000000011	000
P3: 0000000000000010	0010
P4: 0000000000000001	0011
P5: 0001100000AAAAAA	010
P6: 0010000000SSSSSS	011
P7: 0011100000AAAAAA	1100
P8: 0000000000000111	1101
P9: 1111111111111110	1110
P10: 1111111111110110	1111

Old Code Size = $12 \times 4 = 48$ Bytes = 384 bits

New Code Size = Left 16bits size + Right 16bits size = (30bits) + (39bits) = 69 bits

Compression Ratio = $69 / 384 = 0.1797$