6.823 Computer System Architecture
Predication for VLIW Processors

http://csg.csail.mit.edu/6.823/

As we saw in Lecture 17, ***Predication*** is a technique to provide conditional execution without branches and control dependences. Predication allows associating each operation with a Boolean flag, called a predicate. If the predicate is true, the operation executes normally; if the predicate is false, the operation is treated as a no-op. To avoid control dependences, predicated operations are fetched and decoded as usual, but turned into a no-op if their predicate is false.

In this handout, we will explain how to add predication to VLIW processors by first describing their ISA, and then explaining how we implement predication on top.

Our VLIW Processor can execute 4 MIPS-like operations with a single instruction: two simple integer operations, one multiply or divide operation, and one memory operation, with the following properties:

- All functional units and load/store units are fully pipelined and latch their inputs.
- A simple integer operation has a latency of a single cycle, meaning the result of an operation issued at cycle X can be used by a dependent instruction in cycle X+1. Simple integer operations include standard ALU operations and calculating branch address and/or the target, but *exclude* multiplies and divides.
- A multiply or divide operation has a fixed latency of 2 cycles
- A load/store has a fixed latency of 3 cycles.

Thus, the following is the instruction encoding:

Simple Int Op	**Simple Int Op**	**Mul/Div Op**	**Memory Op**

We extend our VLIW architecture as follows. We add 32 1-bit predicate registers, `p0` to `p31`, stored in a predicate register file. We also change the encoding of the integer, multiply/divide, and load/store operations to allow them to be predicated on the value of one of these predicate registers.

We denote predicated operations in assembly by prefixing them with the predicate in parenthesis. For example:

`(p1)ADD rd, rs, rt`	`SUBI rt, rs, imm`	`MUL rd, rs`	`LW rs, 0(rt)`

denotes that the `ADD` operation is predicated on the value of predicate register `p1`: if `p1` is True (i.e., 1), the operation will execute as usual, and otherwise it will be turned into a no-op. Note that predicates are per-operation, so the `SUBI`, `MUL` and `LW` operations are not affected by the predicate even if they are part of the same VLIW instruction.

We also allow instructions to be predicated on the inverse of a predicate register. For example:

`(!p1)ADD rd, rs, rt`	`SUBI rt, rs, imm`	`MUL rd, rs`	`LW rs, 0(rt)`

denotes that this `ADD` operation is predicated on the inverse of the value of register `p1`: if `p1` is False (i.e., 0), the operation will execute as usual, and otherwise it will be turned into a no-op.

Finally, we add the following set of *simple integer* operations which set the predicate register:

- `SETPEQ pd, rs, rt`: Set PredReg[pd] to 1 if Reg[rs] == Reg[rt], or 0 otherwise
- `SETPGE pd, rs, rt`: Set PredReg[pd] to 1 if Reg[rs] >= Reg[rt], or 0 otherwise
- `SETPLT pd, rs, rt`: Set PredReg[pd] to 1 if Reg[rs] < Reg[rt], or 0 otherwise
- `SETPNE pd, rs, rt`: Set PredReg[pd] to 1 if Reg[rs] != Reg[rt], or 0 otherwise

Give example of if-else code with only the VLIW code and the predicated version.

With these changes, we can implement conditional execution without branches. For example, consider the following code:

```
if (x == 0)
     A = B;
     B = B + 1;
else
     A = A * A;
```

Assuming that registers `R1`, `R2`, and `R3` hold the values of `x`, `A`, and `B`, respectively, the following is the VLIW code with a conditional branch:

label	Simple Int Op	Simple Int Op	Mul/Div Op	Memory Op
	BNEZ R1, else			
if	ADDI R2, R3, 0	ADDI R3, R3, 1		
	J end			
else			MUL R2, R2, R2	
end				

With predication, we can rewrite the above assembly to execute on our VLIW machine without the branch. The newly introduced `SETPEQ` operation sets predicate register `p0` based on the if-condition, and the `ADDI` operations are predicated on `p0`, and the MUL operation is predication on the inverse of `p0`.

label	Simple Int Op	Simple Int Op	Mul/Div Op	Memory Op
	SETPEQ p0, R1, R0			
	(p0)ADDI R2, R3, 0	(p0)ADDI R3, R3, 1	(!p0)MUL R2, R2, R2	
end				