6.823 Computer System Architecture

Handout #9: Execute Data and Write Effective Address extensions

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

Part A: Execute Data Instruction

One day, Ben Bitdiddle started an EDSACjr-based company. Ben wanted to leverage the speed of read-only memory and avoid the inherent hazards of the Princeton architecture, so he went with a Harvard architecture. Unfortunately, Ben's system didn't have any index registers, so he couldn't write self-modifying code. That meant there were a large number of programs he couldn't implement anymore. Ben decided to add an instruction to solve this problem. He called his new instruction `EXD` , for execute data. The `EXD` instruction treats the contents of the accumulator as a new instruction and executes whatever that instruction may be. If the accumulator does not contain a valid instruction, then `EXD` falls back on the processor's fault handling for bad instructions (which you needn't worry about).

For example, from Handout #1 the instruction `ADD 6` (which adds the contents of memory at address six to the accumulator) is encoded as: 0000 1000 0000 0110.

Therefore if the contents of the accumulator are 0000 1000 0000 0110, the `EXD` instruction will interpret the accumulator as an `ADD 6` instruction, and add the contents of memory at address six to the accumulator (now interpreted as the *number:* 0000 1000 0000 0110 = 2054). So if memory at address six holds the value one, then the accumulator will become 0000 1000 0000 0111. (Which can be interpreted either as the instruction `ADD 7` or the number 2055.)

To simplify writing assembly code, Ben Bitdiddle also augments the EDSACjr's instruction set with a load instruction, `LD n`. This load simply places the value in memory address n into the accumulator: `ACC` ← `Mem[n]`. LD is encoded as 01011 *n;* that is, the opcode is 01011.

Part B: Write Effective Address Extensions

You've noticed that many programs execute code similar to the following during loops:

```
LD  R1, 4(R2)
ADD R2, R2, 4
```

Or:

```
ST  R1, 4(R2)
ADD R2, R2, 4
```

You want to optimize your architecture for this common case. You are going to do so by adding "write effective address" variants of the load and store instructions, `LDWA` and `STWA`. The semantics of these instructions are that they will perform the normal memory operation (`LD` or `ST`) and then write the effective address in the register that indexed into memory (*not* the register whose contents are read/written to memory). Specifically these instructions do the following:

```
LDWA rs, rt, Imm:
     rs ← Memory[(rt) + Imm]
     rt ← (rt) + Imm

STWA rs, rt, Imm:
     Memory[(rt) + Imm] ← (rs)
     rt ← (rt) + Imm
```

These extensions allow us to rewrite the previous examples as:

```
LDWA R1, R2, 4
```

And:

```
STWA R1, R2, 4
```