

Sara MacAlpine
Robert Morton
Aamir Virani
14 Sep 2000

ARSe Instruction Scheduler

After the demise of the Mars Lander, it was found that the problem was rooted in inefficient instruction order. Thus, the valiant crew from WRC began researching a method to optimize the ARS extended instruction set (ARSe ©). This instruction set, which controls all spacecraft developed by WRC, includes loads, stores, unops (unary operations - uses one register), and binops (binary operations - uses up to three registers). Loads and stores have a latency of more than one clock cycle, which causes stalls when the instruction that follows is dependent on the completion of the load/store. Independent instructions can be moved into these latent positions to prevent stalls, which makes more efficient use of clock cycles. The potential target for our processor would be to reorder instructions in this manner before a CPU executes them.

The instructions are coded in eight bits. The first two are the opcode, which covers the four above-mentioned instructions. The source and destination registers each require two bits, allowing for four general-purpose registers. (This may change later.) Therefore, nineteen pins are used -- 10 pins for input and 9 for output. The chip will serially receive five to eight (8-bit) instructions, which will be buffered. After reordering, the (8-bit) instructions are then serially output in the same manner. Three additional pins are needed; one pin indicates that the chip is ready to read data, another is used by the user to indicate that data is being sent, and the final indicates that the reordered instructions are now being output.

Once the instructions are buffered, the chip should search in ascending order to find an instruction with latency, i.e., a load or a store. Once this is found, an independent instruction should be placed after the load/store if possible, eliminating the latency. To do this, we perform these steps:

1. Set up an array of dependency bits (one per instruction) initialized to 0.
2. Go through the subsequent instructions, determining whether each is dependent on the latent instruction. If so, the dependency bit is set to 1. Once a bit is set to 1, it cannot be set to 0.
3. After going through these instructions, the chip then checks if the instruction immediately following the latent instruction is dependent. If not, then the latency has been eliminated and the job is finished.
4. If it is dependent, it needs to check the lines following the instruction *after* the latent instruction and mark whether these are dependent on this instruction.
5. Then, the chip again checks whether the instruction immediately after this new checkpoint is dependent. If not, it is moved to the latency position and the job is done.
6. Otherwise, the cycle continues until a viable solution is found or the chip reaches the last of the input instructions.

This may not make much sense, so here is pseudocode that clarifies the algorithm.

```
//find latent instruction by checking opcode
y = Latent Instruction;
Found = 0;
while(!Found and y! = Last Instruction+1)
{
    trace = y+1;
    while(trace != Last Instruction+1)
    {
        If trace dependent on y
            Dependency[y]=1;
        trace++;
    }
    If Dependency[y+1]==0
        Found=1;
    Else
        y++;
}
//now move instruction
```

The following components will be on the chip: memory, counters, decoders, and comparators. An idea of how they will be used is in the attached block diagram.

Depending on space and time constraints, we may add the following:

1. Currently, the optimization may be considered top-to-bottom. By going through the input instructions backwards, bottom-to-top optimization may also be done.
2. Different latencies can be assigned to the load and store instructions, indicative of a more realistic pipeline.
3. Use a more complex instruction set, ARSe+, which includes a jump instruction, currently under research.