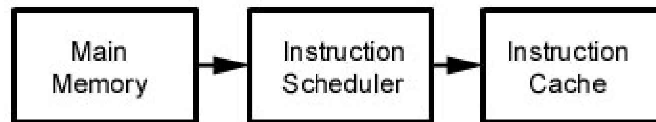
ARSe Instruction Scheduler: Milestone 3

Initial Logic & Algorithm Flow

ARSe Instruction Scheduler: Practical Uses

The ARSe instruction scheduler is intended to operate as a hardware based optimization for executable code. It is most common to find optimizations such as instruction scheduling in the compiler for a particular platform, but not all compilers have such optimizations. Our instruction scheduler will be a proof of concept for migrating these optimizations to hardware.

Because our instruction scheduler operates on small blocks of code, its proper use would not be in the instruction stream going to the processor. Instead, we would want to place the ARSe instruction scheduler between main memory and the instruction cache on the processor, as illustrated:



This is a similar configuration to what modern processors like Intel's Willamette and Transmeta's Crusoe do with a "trace cache," which caches decoded (or translated) copies of the instructions, and are available for the processor to retrieve at a later time. Like these trace caches, our chip is not involved with the interpretation of the instructions themselves. As a proof of concept we are using a simple instruction set that broadly allows many of the interesting cases for instructions scheduling.

Input Control

The chip will receive input serially. There are:

- eight pins used to receive instructions (one-at-a-time),
- one pin indicating when the chip is ready to read (which is turned on during startup and then set by the output controller after instructions are passed out of the chip),
- one pin indicating when the instruction source is writing to the chip.

When the chip is ready to read and the source is ready to write, the chip will:

1. write the instruction on the serial input to memory,
2. place the counter value into the Correct Order lookup table,
3. repeat 1 and 2 until the internal memory is filled.

The instruction source will hold the instruction on the pins long enough for it to be read and then go to the next one; it will also stop writing when the ready to read pin is off, which occurs when the memory is full. At this point, the chip begins optimizing the instruction block.

```
//When first turned on, everything resets so that chip will be ready to perform
ready_to_read = true;

if (ready_to_read && source_writing)
{
	for (i = 0; i < length_of_memory; i++)
	{
		memory[i] = instruction_pins;
		Order[i] = i;
	}
	ready_to_read = false;
}
```

Output Control

The chip will also transmit output serially. There are:

- eight pins used to send instructions (one-at-a-time),
- one pin indicating that the chip is outputting.

When the chip is ready to ready to write, the chip will:

1. set writing to true,
2. take the next instruction according to the Correct Order lookup table,
3. get this instruction from the instruction memory,
4. send it to output,
5. increment the counter,
6. repeat 2-5 until the last instruction is output,
7. set ready_to_read to true

The instruction output will be held on the pins for a defined length of time before going to the next one.

```
//When first turned on, everything resets so that chip will be ready to perform
writing = false;

//When the chip is ready to write after reordering…
writing = true;
for (i = 0; i < length_of_memory; i++)
{
        output_pins = memory[Order[i]];
}
writing = false;
ready_to_read = true;
```

ARSe Instruction Set

The ARSe instruction set is an 8-bit instruction set with the first two bits as the opcode and the remaining six bits used to encode source and sink registers, immediates, offsets, etc. Note that there are four registers ($r0-$r3) for usage. The set is composed of four types of commands:

- memop (01) - loads and stores - Loads take the value at a specified memory address and place it into a register, while stores take the value in a register and place it into a specified memory address.

Ex:	lw $r1, 0x7 →	0 1 0 1 1 1 1 0
	sw $r3, 0x4 →	0 1 1 1 1 0 0 1

- binop (11) - operations that access up to three registers (add $r1, $r2, $r3)

Ex:	add $r1, $r2, $r3 →	1 1 0 1 1 0 1 1
	sub $r3, $r0, $r2 →	1 1 1 1 0 0 1 0

- unop (10) - operations that access two registers (addi $r1, $r2, 2)

Ex:	addi $r1, $r2, 2 →	1 0 0 1 1 0 1 0
	subi $r2, $r1, 3 →	1 0 1 0 0 1 1 1

- jump (00) - A jump instruction skips portions of code and executes instructions beginning at the specified offset. This instruction is a general form for all jump instructions, potentially including conditional jump instructions, procedure calls, etcetera. For example, j 0xc jumps ahead 12 words, executes that statement and continues onward. In this instruction set, the statement to which j jumps must be jd, a jump destination.

Ex:	00000 j 0xc →	0 0 0 1 1 0 0 0
	00001 add $r1, $r2, $r3	1 1 0 1 1 0 1 1
	00010 addi $r1, $r2, 2	1 0 0 1 1 0 1 0
	…	
	01100 jd →	0 0 0 0 0 0 0 1
	01101 sub $r1, $r2, $r3	1 1 0 1 1 0 1 1
	01110 subi $r2, $r1, 3	1 0 1 0 0 1 1 1

Register values are not guaranteed after a jump, so the user should store any necessary values to memory before using the “j” instruction.

Data Dependencies

Two members of the ARSe instruction set, load and store (the "memops"), have a two-cycle clock latency, which makes it necessary for some type of independent instruction to immediately follow them. Our goal is to locate suitable independent instructions from further along in the code and insert them after the loads/stores to eliminate the need for so many stalls. However, this becomes tricky when you consider the ARSe set's many data dependencies. Here is a list of each instruction, with those that are directly dependent on it listed underneath: (x is a don't care unless otherwise noted)

load sink, M1
- store sink, x
- store x, M1
- load sink, x
- unop sink, x, x * unop x, sink, x
- binop sink, x, x * binop x, sink, x * binop x, x, sink

store source, M1
- load source, x
- load x, M1
- store x, M1
- unop source, x, x
- binop source, x, x
- jump (when the store is not also the latent instruction we are seeking a follower for)

unop sink, source, x (where the x to the left is an immediate)
- store sink, x
- unop x, sink, x * unop source, x , x
- binop x, sink, x * binop x, x, sink * binop source, x, x
- load source, x

binop sink, source1, source2
- store sink, x
- load source1, x * load source2, x
- binop x, sink, x * binop x, x, sink * binop source1, x, x * binop source2, x, x
- unop x, sink, x * unop source1, x, x * unop source2, x, x

In addition to the above, in our processor, we will always mark anything we find following a jump destination instruction as dependent on it, since we don't want to change a jump destination's position. It would be impossible for us to then go back and change the jump offset if we did move the destination. Also, any time (jump offset + distance it would have to move up to fill latent spot) is greater than the maximum jump offset (5 bits) we would mark it as being dependent, or unmovable. If the movement of an instruction under a jump (causing it to move down) would overflow the offset in the negative sense, everything under the jump is marked as dependent. The main reason that jumps are so mobile in this instruction set is that register values are never guaranteed after a jump has occurred.

ARSe Instruction Scheduler: Algorithm Flow

The core algorithm may be summarily described with the following pseudo-code:

```
x=0;  // start with initial instruction
while (x < Last instruction) {

      if instruction[lookup[x]] != Latent instruction {
            x++;
            continue;   // resume searching for latent instructions
      }

      clear Dependency[] table;     // reset all dependency bits
      Found = 0;              // found an instruction to fill the latency?
      y=x;                    // start y at the latent instruction

      while (!Found and y < Last Instruction)
      {
            Jump_destination = 0;   // a jump destination has been found?
            Jump_source = 0;        // a jump source has been found?
            trace = y+1;
            while(trace <= Last Instruction)
            {
                  if instruction[lookup[trace]] dependent on y
                        Dependency[trace] |= 1;
                  if instruction[lookup[trace]] = jump destination
                        Jump_destination = 1;
                  if instruction[lookup[trace]] at edge of jump boundary
                        Jump_source = 1

                  // don't reorder instructions after a jump src/dest.
                  if Jump_destination = 1  or  jump_source = 1
                        Dependency[trace] |= 1;

                  trace++;
            }
            If Dependency[y+1]==0
                  Found=1;
            Else
                  y++;
      }

      // reorder the instructions
      if instruction[lookup[y+1]] is jump source {
            calculate new offset
            if offset overflows
                  continue;   // resume searching for latent instructions
      }
      temp_address = lookup[y+1];
      for (  ; y > x; y--) {
            lookup[y+1] = lookup[y];
      }
      lookup[y+1]=temp_address;

      x++;        // keep looking for latent instructions
}
```

An explanation is still required. First, the variables follow as such:

- x – is the head address that points to latent instructions.
- y – is the address used to locate movable instructions which have no dependencies that require them to stay put
- trace – is the address that is used to scan the instructions and mark a dependency bit indicating that they have some sort of dependency that prevents them from being moved.
- Dependency[] – is a table that contains one bit per instruction indicating whether or not it can be moved
- lookup[] – is the lookup table containing addresses of the reordered instructions
- instruction[] – is the table containing the original buffered instructions.
- Found – indicates when a movable instruction is found
- Jump_destination – indicates when a jump destination has been found during a particular trace.
- Jump_source - indicates when a jump source has been found during a particular trace.

The algorithm tracks through the buffered instructions trying to find any that have a latency hole that needs to be filled. When it finds one, it searches down for an instruction that can fill this hole. Since instructions can exist in such a fashion that reordering them will cause the code to behave differently, we have to mark each instruction on whether or not there is another instruction preventing it from being moved into the latency hole. This is the dependency sweep done with the address "trace," which uses the instruction at address "y" for reference in determining a dependency. See the section on dependency logic for a further explanation. After this dependency sweep, the first instruction that isn't marked with a dependency bit can be moved into the latency hole

A simple process of copying addresses down and then filling the hole made will complete the reorder for the latent instruction. Then we continue searching. Once finished, we are ready to output the reordered instructions by going through the reorder table. See the section on the output controller for more details.