This paper gives the cycle-by-cycle execution account of the out-of-order implementation of the 16-bit Ridiculously Simple Computer (RiSC-16), a teaching ISA that is based on the Little Computer (LC-896) developed by Peter Chen at the University of Michigan.

1. RiSC-16 Instruction Set

The RiSC-16 is an 8-register, 16-bit computer. All addresses are shortword-addresses (i.e. address 0 corresponds to the first two bytes of main memory, address 1 corresponds to the second two bytes of main memory, etc.). Like the MIPS instruction-set architecture, by hardware convention, register 0 will always contain the value 0. The machine enforces this: reads to register 0 always return 0, irrespective of what has been written there. The RiSC-16 is very simple, but it is general enough to solve complex problems. There are three machine-code instruction formats and a total of 8 instructions. The instruction-set is given in the following table.

<table>
<thead>
<tr>
<th>Assembly-Code Format</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>lui regA, immed</td>
<td>R[regA] &lt;- immed &amp; 0xffc0</td>
</tr>
<tr>
<td>beq regA, regB, immed</td>
<td>if ( R[regA] == R[regB] ) { PC &lt;- PC + 1 + immed (if label, PC &lt;- label) }</td>
</tr>
<tr>
<td>jalr regA, regB</td>
<td>PC &lt;- R[regB], R[regA] &lt;- PC + 1</td>
</tr>
</tbody>
</table>

PSEUDO-INSTRUCTIONS:

| nop | do nothing |
| halt | stop machine & print state |
| lli regA, immed | R[regA] <- R[regA] + (immed & 0x3f) |
| movi regA, immed | R[regA] <- immed |
| .fill immed | initialized data with value immed |
| .space immed | zero-filled data array of size immed |

The instruction-set is described in more detail (including machine-code formats) in The RiSC-16 Instruction-Set Architecture. System calls, interrupts/exceptions, and the handling of interrupts and exceptions are described in more detail in the document RiSC-16 System Architecture. The out-of-order architecture is described in the document An Out-of-Order RiSC-16.
2. Example Operation

The following figures illustrate (in excruciating detail) the movement of instructions, data, and status information through the pipeline during the execution of a relatively simple piece of code. This is done to animate the design, hopefully giving a clear picture of what happens in the machine. The following code example is used:

```assembly
# main loop: loads a number and then a variable number of data items to subtract
# from the first. at end, saves result in "diff" memory location
lw   r1, r0, arg1
lw   r3, r0, count
loop: lw   r2, r4, arg2
      movi r7, sub
      jalr r7, r7
      addi r3, r3, -1
      beq r3, 0, exit
      addi r4, r4, 1
      beq r0, r0, loop
exit: sw   r1, r0, diff
      halt

# subtract function: operands in r1/r2, return address in r7. result -> r1
sub: nand r2, r2, r2
      addi r2, r2, 1
      add r1, r1, r2
      jalr r0, r7

# data: count is the # of items to subtract from arg1 (in this case, 1: arg2)
# diff is where the result is placed
count: .fill 1
arg1: .fill 9182
arg2: .fill 737
diff: .fill 0
```

The execution takes 29 cycles and illustrates many of the possible behaviors: ALU operations, memory operations, BEQ instructions predicted-taken and predicted-not-taken, BEQ instructions predicted correctly and incorrectly, JALR instructions (which use the branch-miss facility and behave like a mispredicted BEQ), instruction-enqueue of 0, 1, and 2 instructions, the filling up of the instruction queue thereby blocking enqueue and fetch, retirement of instructions, etc.

The state of the machine at the start of each cycle is shown in Figures 1–28. Dark lines indicate movement of data (which is latched at the end of the cycle and is visible in machine state on next cycle). The top bit of the instruction ID indicates the result bus to watch: memory bus vs. ALU bus. For instance, a LW enqueued in slot 3 will tag the register file with id 13 rather than 03; this notifies other instructions not to latch the results of the LW’s add-immediate operation that simply generates the target address. Opcode values are prefixed with “a” indicating ALU instructions, “b” indicating branch operations, or “m” for memory operations. Non-obvious fields of the IQ entry (not all are fields; some are just signals): Valid, Done, Out, Branch-taken, Memory-issuable, Address-generated, Issuable (to ALU), Slot-number, and X = kill the instruction. The figures start on cycle 2; during cycle 1 we fetched instructions at pc=0000 and pc=0001 (two LW instructions) into the two first fetch buffers (A and B) and incremented the program counter by 2.
During execution cycle 2, we enqueue the first two instructions into the slots indicated by the tail pointer (labeled “T” at the side of the instruction queue); this includes reading from the register file: the register value RV, its valid bit V, and its source SC. During this phase, the targets of the two LW instructions (r1 and r3) are tagged “invalid” and their SC fields directed to the two LW instructions. Note the top bits of these IDs are “1”, indicating that the final result will come from the memory bus, not an ALU bus. We also fetch the next two instructions (an LW and a LUI—a MOVI is replaced by the assembler with a LUI+ADDI pair) into the alternate fetch buffers (C and D). The program counter is incremented by two.
During cycle 3, the first two instructions issue address-generate operations to the ALUs; we enqueue the second two instructions; and we fetch the third pair of instructions: an ADDI and a JALR. The program counter is incremented by two. During the enqueue phase, the targets of the two instructions (LW -> r2, LUI -> r7) are set appropriately: the SC field for r2 will become “11” and the SC field for r7 will become “03”, indicating that LUI’s result will be on an ALU bus. Note that the v/src fields in each of the two issuing LW instructions will become invalid and refer to the LW instruction itself. This allows the Tomasulo-style logic to be used to forward the results of an address-generation back into the memory instruction’s IQ entry.
During cycle 4, the results of the two address-generate operations are placed on the two ALU-result busses and feed their results to IQ slots 0 and 1. The second pair of instructions issue to ALUs: the LW in slot 2 sends an address-generate operation and the LUI in slot 3 sends a LUI. The third pair of instructions (ADDI+JALR) is enqueued in slots 4 and 5 and sets register-source values appropriately: because both target r7, the SC field for r7 becomes the ID of the latter of the two instructions—that of the JALR, which is enqueued into slot 5.
During cycle 5, the target addresses for the first two LW instructions are compared, and the “M” bits for each are set appropriately, indicating whether the memory operation can be issued to the memory queue. The LW/LUI pair is executed and the results placed on the ALU busses (LUI will be marked “done” in its IQ entry). The ADDI instruction in IQ slot 4 can issue, even though its register operand is tagged invalid in the IQ slot, because its source ID matches that on ALU-1 result bus. There are two IQ slots open; two instructions are enqueued. Two more are fetched.
During cycle 6, the LW instruction in iq0 is issued to the memory queue (its M tag is 1). On the following cycle, we will see this reflected in the memq’s entries. The JALR in iq5 is issued to an ALU because its register operand is available on ALU-0 result bus. The instruction queue is full (no entries marked “invalid”), and therefore the enqueue mechanism is stalled. Normally, this would not stall the fetch mechanism (later cycles will illustrate this) ... normally, another two instructions would be fetched into the alternate fetch buffers. However, there is a predicted-taken branch in fetchbuf B (the backwards branch `beq r0,r0, loop`), so the program counter is redirected during this cycle. Fetch will commence down the predicted path on the following cycle.
During cycle 7, the second LW instruction is issued to the memory queue (note that both were “ready” on the previous cycle, but we can only issue one per cycle). The “status” of the previous memory operation is “1” which indicates that it is in mid-request (once the status is “3” the operation is complete). The JALR issued on the previous cycle is on the ALU-0 result bus, and it has set the BRANCHMISS valid-bit high, indicating a change in control flow. This stalls both fetch and enqueue and invalidates the fetchbuf entries. The program counter will be redirected, using the valid produced by the JALR instruction. Instructions to be stomped on are tagged “1” in the “X” column: iq6 and iq7; those following the JALR. Those not stomped can still issue (iq1).
At the beginning of cycle 8, we see that the stomped instructions are now marked “invalid” in the instruction queue, and the tail pointer has been reset appropriately. The fetch buffers have been marked “invalid.” Several instructions (iq3, iq4, and iq5) are “done” and thus ready to commit, but are held up by the three LW instructions at the head of the queue. The program counter has been rest appropriately (it has the value of the branchmiss status from the previous cycle). Most importantly, the contents of the register file reflect the correct machine state: on the previous cycle the \texttt{addi} instruction in iq6 targeted r3, which had “06” as its source. Now, register r3 has the previous source of r3 listed: the LW in iq1. During this cycle, another LW is issued to the memory.
queue. Two instructions are fetched. Very little else happens because most of the enqueued instructions are done.

During cycle 9, the first of the three LW instructions becomes ready in the memory queue; its result will be sent on the memory result bus on the following cycle. The two instructions fetched on the previous cycle (the NAND and ADDI at the top of the sub subroutine) are enqueued and the next two subroutine instructions (ADD and JALR) are fetched. The states of the memory queue entries are incremented by one.
During cycle 10, the result of the first LW instruction is seen on the memory result bus. The result is latched at the end of the cycle, when the instruction will be tagged “done”. No instructions are issued to functional units because the two potential instructions are dependent on the LW instruction in iq2 (its result will become available in two cycles). Because the IQ is full, enqueue is stalled. Because there are no predicted-taken branches (i.e. backwards branches) in the fetch buffers, instruction fetch is not stalled; the two instructions following the subroutine are fetched (they are actually data, but they will be discarded when the JALR at the end of the subroutine takes effect).
During cycle 11, the first LW instruction commits its result to the register file. On the following cycle, its IQ slot will be tagged as invalid, marking it available for a new instruction. The result for the second LW instruction is seen on the memory result bus. Instructions in iq6 and iq7 are stalled waiting on the third LW instruction. Enqueue is stalled waiting for an available IQ entry. Fetch is stalled waiting for an available fetch buffer.
During cycle 12, the second LW instruction commits. The slot opened up by the LW instruction (slot iq0) is the enqueue-target for one of the instructions in the fetchbufs (the ADD instruction in fetchbuf C). The result for the third LW instruction is seen on the memory result bus. This will enable the waiting instructions to issue to functional units on the following cycle. Fetch is still stalled because there are no empty fetch buffers.
During cycle 13, two instructions commit, opening up two more slots in the instruction queue. The NAND instruction in iq6 that was waiting on the LW in iq2 issues to a functional unit. The IQ slot opened up by the LW instruction in iq1, which committed on the previous cycle, is filled by the instruction in fetchbuf D: the JALR instruction that marks the end of the subroutine.
During cycle 14, two more instructions commit
Figure 14: EXECUTION CYCLE 15

![Diagram of execution cycle 15 with labels and dataflows.](image)
Figure 15: EXECUTION CYCLE 16
Figure 16: EXECUTION CYCLE 17

PC: 0006

INSTRUCTION
MEMORY

BRANCH
BACK:

FETCH
BUFFERS:

v PC

on FB v INST PC

0 0016

-> A: 0 0000 0114

-> B: 0 0000 0115

-- C: 0 02e1 0012

-- D: 0 0000 0013

INSTRUCTION
QUEUE:

HT IQ V D O B M A I S X OP rT EX RSLT ARG0 ARG1 v sc ARG2 v sc PC

H. 0: 1 1 0 0 0 1 0 0 0 a0 01 00 20fd 0002 23de 1 10 fd1f 1 07 000e
.. 1: 1 1 0 0 0 1 0 0 0 b7 00 00 0010 0000 0006 1 05 000f 1 00 000f
.. 2: 0 0 0 0 0 0 0 0 0 a0 00 00 0000 0001 0000 1 00 20fd 1 00 0010
.. 3: 0 0 0 0 0 0 0 0 0 a1 00 00 0000 fdfe 0006 1 05 0000 1 00 0011
.. 4: 0 0 0 0 0 0 0 0 0 a0 00 00 0000 ffe1 0000 1 00 0012
.. 5: 0 0 0 0 0 0 0 0 0 a0 00 00 0000 0000 0000 1 00 0000 1 00 0013
.. 6: 0 1 0 0 0 1 0 0 0 a2 02 00 fd1e 0002 02e1 1 12 02e1 1 12 000c
.. 7: 0 1 0 0 0 1 0 0 0 a1 02 00 fd1f 0001 fd1e 1 06 0000 1 00 000d

MEMORY
QUEUE:

S ADDR DATA OP ID

0 0011 0000 m5 10
0 0010 0000 m5 11
0 0012 0000 m5 12

DATA
MEMORY

v RSLT ID EX

0 02e1 12 0

MEMORY
RESULT
BUS

ALU-0
INPUT
REGISTERS:

v ARG0 ARG1 ARG2 OP B ID PC

0 0002 23de fd1f a0 0 00 000e

ALU-0 RESULT
BUS

ALU-0 BRANCHMISS

v RSLT ID EX

0 20fd 00 0

ALU-0
RESULT
BUS

v ID PC

0 00 0011

v RSLT ID EX

0 0010 01 0

ALU-1
RESULT
BUS

ALU-1 BRANCHMISS

v ID PC

0 01 0006

v RSLT ID EX

0 0010 01 0

ALU-1
RESULT
BUS

v ID PC

0 01 0006

v RSLT ID EX

0 0010 01 0

ALU-1
RESULT
BUS

v ID PC

0 01 0006

v RSLT ID EX

0 0010 01 0

ALU-1
RESULT
BUS

v ID PC

0 01 0006

v RSLT ID EX

0 0010 01 0

ALU-1
RESULT
BUS

v ID PC

0 01 0006
Figure 17: EXECUTION CYCLE 18
Figure 18: EXECUTION CYCLE 19
Figure 19: EXECUTION CYCLE 20
Figure 20: EXECUTION CYCLE 21
Figure 21: EXECUTION CYCLE 22
Figure 22: EXECUTION CYCLE 23
Figure 23: EXECUTION CYCLE 24
Figure 24: EXECUTION CYCLE 25
Figure 25: EXECUTION CYCLE 26
Figure 26: EXECUTION CYCLE 27
Figure 27: EXECUTION CYCLE 28
Figure 28: EXECUTION CYCLE 29