SCAD Architecture Project Paper

Julius Roob, ...
University of Kaiserslautern, Embedded Systems Group
julius@juliusroob.de, ...

DRAFT   February 16, 2016

Contents

Todo list

1 Instruction Set Architecture

Aside from the mandatory move instruction, our Instruction Set Architecture (ISA) needs to have instructions for loading immediate values, jumping to fixed addresses and branching. An overview of those instructions is given in Figure 1.

instruction	**semantics**
`move src, dest`	Move data from an output buffer to an input buffer by sending this instruction to the corresponding functional units. Data move will asynchronous
`jump address`	Set PC to address.
`immediate data`	Place data into output buffer of control unit.
`branch address`	A no-op when the first data in the input buffer is a 0, jump to address otherwise. Will wait for data to arrive when there is none.

Figure 1: SCAD Architecture Instructions

2 Move Instruction Bus

2.1 2-Phase Commit

To take into account both stalls of source and destination functional units, the control unit sends move instructions in two phases, both of which are indicated by a rising edge of the ”valid” flag. First, with phase low, the functional units only check whether there is space in the corresponding input and output buffers. When stalls are asserted, the control unit waits some time until retrying. When no functional unit stalls, the phase being high signals a ”write”.

Either the text, the diagram and/or the implementation need to be adapted (not in sync).

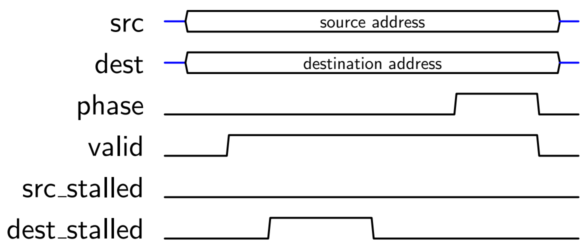

Figure 2: 2-Phase Commit on Move Instruction Bus with Destination Stalling

3 Control Unit and Data Network

To keep the MIB simple, we will have the control unit send immediate values, and receive branch conditions through the data network. While broadcasting immediate values through either the MIB or a separate bus may be the faster alternative, having the control unit send them through the data network keeps the architecture cleaner.

better formulation please

4 Future Work

Having duplication as a seperate functional unit may cause it to be a significant bottleneck. One feasible solution to this is the extension of the move instruction by an additional ”non-destructive” move, where data sent is kept in the output buffer.

at least two more ideas

Appendices

A Memory Access and Branch

Basic example for memory access and branching:

```
// Basic function:
// *result = *op1 == *op2 ? 33 : 27;

// Load operands from memory
immediate <op1_address> // op1_address into the control unit output
move ctrl.o0, load.i0
immediate <op2_address>
move ctrl.o0, load.i0

// Send result destination to "store" function unit
immediate <result_address>
move ctrl.o0, store.i0

// Send parameters to compare unit
move load.o0, cmp.i0
move load.o0, cmp.i1

move cmp.o0, ctrl.i0 //move to control unit input for branch
branch yes // branch to yes if control unit input != 0
no:
  immediate 27
  jump both
yes:
  immediate 33
both:
  move ctrl.o0, store.i1 // move to data input of the store unit
```

B Fibonacci

```
// WORK IN PROGRESS: might be right - but could just as well be wrong

setup:
  immediate 0
  move ctrl.o0 duplication.i0

  immediate N
  move ctrl.o0 add.i0
  move duplicaton.o0, add.i1

  immediate 1
  immediate 0
  move ctrl.o0 add.i0
  move ctrl.o0 add.i1

loop:
  // Loop invariant: * output buffer of add contains: (n-i)
  //                                                  then fib(i)
  //                 * output buffer of duplication contains: fib(i-1)

  immediate -1
  move ctrl.o0, add.i1

  move duplication.o0, add.i1
  // add.i1: -1, fib(i-1)
  // add.o0: (n-i), fib(i)

  move add.o0, duplication.i0
  // add.i1: -1, fib(i-1)
  // add.o0: fib(i)
  // duplication.o0: (n-i), (n-i)

  move duplication.o0 ctrl.i0
  // add.i1: -1, fib(i-1)
  // add.o0: fib(i)
  // duplication.o0: (n-i)
  // ctrl.i0: (n-i)

  move duplication.o0 add.i0
  // add.i1: fib(i-1)
  // add.o0: fib(i), (n-i-1)
  // ctrl.i0: (n-i)

  move add.o0, duplication.i0
  // add.i1: fib(i-1)
  // add.o0: (n-i-1)
  // duplication.o0: fib(i), fib(i)
  // ctrl.i0: (n-i)

  move duplication.o0, add.i0
  // add.o0: (n-i-1), fib(i+1)
  // duplication.o0: fib(i)
  // ctrl.i0: (n-i)

  // takes ctrl.i0 as branch condition
  branch loop

finished:
  // TODO: write values to result place in ram?
```