Synchronous Control/Asynchronous Dataflow (SCAD) Processor

***Abstract*—The processing units (PUs) of the Synchronous Control/Asynchronous Dataflow (SCAD) processor communicate directly through an on-chip network rather than through a centralized register file. Programs are sequential sequences of move instructions that send immediate operands, opcodes, and data transfer commands to the PUs. To allow asynchronous execution of the issued move instructions, the input and output ports of the PUs use special FIFO buffers. The following defines the set of PUs and move instructions of the SCAD processor.**

Overall Architecture

Processing Units

- `CU` is the control unit that reads the next move code instruction (see below) determined by the program counter `pc` from the program memory and sends it to the involved PUs (including the `LSU`)
- `LSU` is the load/store unit that loads values from or stores values to the data memory
- `PU[0]`,...,`PU[numPU-1]` are general-purpose PUs

Input Buffers (Target Addresses)

- `PU[p].inL` : left input of `PU[p]`
- `PU[p].inR` : right input of `PU[p]`
- `PU[p].opc` : opcode input of `PU[p]`
- `LSU.adr` : address input of the `LSU`
- `LSU.val` : value input of the `LSU`
- `LSU.opc` : opcode input of the `LSU`
- `CU.cnd` : branch condition input of the `CU`
- `CU.pc` : program counter input of the `CU`

Output Buffers (Source Addresses)

- `PU[p].outL` : left output of `PU[p]`
- `PU[p].outR` : right output of `PU[p]`
- `LSU.out` : output of the `LSU`
- integer numbers (immediate operands)
- opcodes (see below) for target `PU[p].opc` and `LSU.opc`

Two-Lane Buffers

Except for the opcode buffers `PU[p].opc` and `LSU.opc`, all buffers have two lanes, i.e., they store entries (adr, val) where both adr and val can be undefined (denoted as $\perp$). For input buffers, adr is the address of an output buffer, and for output buffers, adr is the address of an input buffer.

Move Code Instructions

SCAD's move code instructions `src -> tgt` trigger a data transfer from source `src` to target `tgt`. If `src` is an opcode, it can only be sent to either a `PU[p].opc` buffer or to `LSU.opc`, otherwise any other input buffer can be the target.

OpCodes for `PU[p]`

Arithmetic Operations:

- `AddN/AddZ` : unsigned/signed addition
- `SubN/SubZ` : unsigned/signed subtraction
- `MulN/MulZ` : unsigned/signed multiplication
- `DivN/DivZ` : unsigned/signed division
- `ModN/ModZ` : unsigned/signed remainder
- `EqqN/EqqZ` : unsigned/signed equality (both are same!)
- `NeqN/NeqZ` : unsigned/signed inequality
- `LesN/LesZ` : unsigned/signed less than comparison
- `LeqN/LeqZ` : unsigned/signed less than or equal comparison
- `Not` : boolean negation
- `And` : boolean conjunction
- `Or` : boolean disjunction
- `Xor` : boolean exclusive disjunction

Token Move Operations:

- `Copy` : copy the head of `PU[p].inL` to the tail of `PU[p].outL`
- `Dup` : copy the head of `PU[p].inL` to the tails of `PU[p].outL` and `PU[p].outR`
- `Swap` : copy the head of `PU[p].inL` to the tail of `PU[p].outR` and the head of `PU[p].inR` to the tail of `PU[p].outL`
- `Join` : consume dummy values from `PU[p].inL` and `PU[p].inR` and send a dummy value to `PU[p].outL`
- `Kill` : consume the head of `PU[p].inL`

OpCodes for `LSU`:

- `Load` : loading a value from data memory address `srcAdr`:

```
srcAdr -> LSU.adr
Load -> LSU.opc
```

- `Store` : storing a value from `srcAdr1` to data memory address `srcAdr2`:

```
srcAdr1 -> LSU.val
srcAdr2 -> LSU.adr
Load -> LSU.opc
```

Move Instructions for Control-Flow Branches:

- `pcOff -> CU.pc` implements an unconditional branch: The `CU` immediately updates the program counter by adding the offset `pc := pc+pcOff`.
- `tgtAdr -> CU.cnd` : conditional branches are implemented with two move instructions as follows:

```
srcAdr -> CU.cnd
pcOff -> CU.pc
```

After fetching `srcAdr -> CU.cnd`, the `CU` is stalled with the `pc` pointing to instruction `pcOff -> CU.pc` until a boolean value is transfered from `srcAdr` to `CU.cnd`. If the arriving value is true, the branch is taken in that the `CU` simply performs the unconditional branch `pcOff -> CU.pc`, otherwise, the `pc` is incremented to continue with the following instructions.

Program Execution on SCAD

The execution of move code programs is divided into three phases that overlap in the hardware implementation:

Instruction Issue

The `CU` fetches move instructions from program memory using the program counter `pc`. Opcodes are immediately appended to the tail of the buffer `PU[p].opc` or `LSU.opc`. Similarly, immediate operands, i.e., integer constants, are appended with a dummy address to the tail of a two-lane input buffer. All other move instructions `src -> tgt` instruct the PU owning the source buffer to send a value to the target buffer. If both the source and target buffers have space, the entry $(src, \perp)$ is added to the tail of the target buffer `tgt` and the entry $(tgt, \perp)$ is added to the tail of the source buffer `src`. However, if these buffers contain an entry $(\perp, val)$ with a valid value but an invalid address, then the headmost such entry is chosen to add the corresponding address (`src` or `tgt`).

The `CU` is stalled when it fetches the first move instruction `srcAdr -> CU.cnd` of a conditional branch to wait for the arrival of the value of the condition evaluation. During this time, the program counter points to the unconditional branch `pcOff -> CU.pc` which is the second move instruction of a conditional branch.

Typically, the `CU` uses of superscalarity, i.e., it fetches and issues many move instructions within a cycle.

Firing Processing Units

All PUs watch their input buffers for available inputs and fire as soon as the inputs are available and there is space in the output buffers for the results. Inputs are available if the head entries of the required buffers have valid values (addresses must then also be valid), and these entries are then consumed when the PU fires. The PU will then add the result value to the headmost entry $(tgt, \perp)$ with a valid target address if such an entry exists, and otherwise adds a new entry $(\perp, val)$ with an invalid target address (since the move instruction to transfer this value has not yet been fetched).

Data Transfers

In parallel to instruction issuing and PU firing, the output buffers `PU[p].outL`, `PU[p]`, and `PU[p].outR` send their completed head values through the on-chip network: If (tgt, val) is the completed head of such an output buffer, the value val is sent from that output buffer to the input buffer tgt. This is possible because a move instruction `src -> tgt` was previously registered in both buffers, and therefore there is an entry $(src, \perp)$ in the input buffer tgt waiting for this value.

The dataflow transfer network is capable of handling many data transfers in each cycle. Ideally, it is a non-blocking network that can implement any partial permutation on the addresses so that data transfers are only blocked if they refer to the same target buffer.

Motivation of SCAD

The code generation of move code programs has different goals than traditional code generators: While traditional code generators aim to minimize the use of registers shared by the processing units, move code programs aim to maximize the use of instruction-level parallelism (ILP) of the programs. For example, SCAD code generators will favor breadth-first traversal of expression trees over depth-first traversal, since it is known that the former maximizes ILP, but requires more local memory to store intermediate result values, while the latter minimizes local memory requirements but sacrifices ILP.

RISC processors have a rather limited number of shared registers, so code generators for RISC machines are forced to minimize the use of these registers and sacrifice the ILP in this way. Unlike RISC processors, the chip size of SCAD architectures grows only quasi-linearly with the size of the buffers. To achieve this, special non-blocking on-chip networks have been developed whose size grows only quasi-linearly.

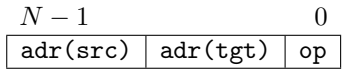
The different goals of code generation and the consumer-producer model of computation of SCAD suggest the use of dataflow graphs as internal compiler representation (see [1]). Therefore, the code generator requires a node and buffer ordering as explained in [2] to allocate PUs for the nodes of the dataflow graph, and to schedule the move instructions corresponding with the edges of the dataflow graph.

Machine Encoding of Move Instructions

To encode of move instructions as bitvectors, all PUs are given addresses: `adr(CU):=0`, `adr(LSU):=1`, and `adr(PU[p]):=p+2`. Based on this, the buffers are given the following addresses:

- `adr(CU.cnd):= 0`
- `adr(CU.pc):= 1`
- `adr(LSU.out):= 2`
- `adr(LSU.adr):= 2`
- `adr(LSU.val):= 3`
- `adr(LSU.opc):= 1`
- `adr(PU[p].inL):= adr(PU[p].outL):= 2*adr(PU[p])`
- `adr(PU[p].inR):= adr(PU[p].outR):= 2*adr(PU[p])+1`
- `adr(PU[p].opc):= adr(PU[p])`

Machine words have the following format with a 2-bit code `op` which is `11` for `src -> CU.pc`, `10` for `Load -> LSU.opc` and `Store -> LSU.opc`, `01` for `n -> tgt` with immediate operand `n`, and `00` otherwise:



References

[1] Schneider, K. Translating structured sequential programs to dataflow graphs. In *Formal Methods and Models for Codesign (MEMOCODE)* (Beijing, China, 2021), I. Saha and L. Zhang, Eds., ACM, pp. 66–77.

[2] Schneider, K., and Bhagyanath, A. Consistency constraints for mapping dataflow graphs to hybrid dataflow/von Neumann architectures. *Transactions on Embedded Computing Systems (TECS) 22*, 5 (2023), 81:1–81:25.