TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

A Comparison of Exposed Datapath and Conventional Processor Architectures

MasterThesis

von

Alexander Schneiders

August 2, 2021

Technische Universität Kaiserslautern,
Department of Computer Science,
67653 Kaiserslautern,
Germany

Examiner: Prof. Dr. Klaus Schneider
Msc. Julius Roob

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit mit dem Thema "A Comparison of Exposed Datapath and Conventional Processor Architectures" selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Kaiserslautern, den 2.8.2021

Alexander Schneiders

Abstract

In this work we compare the current state of the Synchronous Control Asynchronous Dataflow (SCAD) architecture against three well established Von Neumann architectures – that are the pipelined architecture, the super scalar architecture and the VLIW architecture. All four architectures aim to exploit instruction level parallelism (ILP), each in their own manner. Furthermore we implement a simulator framework that allows a fair and comparable simulation of these architectures at a higher level than detailed simulators used in the industry. We explore the possibility of branch prediction in the SCAD architecture and conclude, that is not wise to implement this feature. We also explore the possibility to employ load-store buffers in the SCAD architecture and again conclude, that it is not wise to implement this feature.

Zusammenfassung

In dieser Arbeit vergleichen wir den aktuellen stand der Synchronous Control Asynchronous Dataflow (SCAD) Architekture gegen drei bekannte Von Neuman Architekturen – die Pipeline Architektur, die Super Scalare Architektur und die VLIW Architektur. Jeder dieser Architekturen nutzt Parallelität von Instruktionen auf eine andere Art und Weise aus. Desweiteren entwickeln wir ein Simulator Framework, das uns erlaubt diese Architekturen in einer fairen und vergleichbaren Art und Weise zu simulieren, auf einem höheren level als detailirte Simulatoren aus der Industrie. Wir erkunden die Möglichkleit von Sprungvorhersagen in der SCAD Architektur und schließen, dass es sicht nicht lohnt dies zu entwickeln. Außerdem erkunden wir die Möglichkleit “load-store buffer” in der SCAD Architektur zu verwendene, und schließen wieder, dass es sich nicht lohnte diese zu entwickeln.

Contents

List of Figures

1 Introduction

For a long time increased performance of processors was achieved by minimization of circuitry and increased clock speed. After hitting a wall with these techniques, further increase in performance can be achieved by executing multiple instructions at the same time. This is referred to as instruction level parallelism (ILP). In order to achieve that compiler and processor need to work together:

First of all a program needs to offer parallelism that can be exploited. If every instruction in a program depend on their predecessor instruction, then there is no parallelism that could be exploited by the processor architecture. Some program – like matrix multiplication, or daxpy – offer a lot of parallelism naturally. There are well known compiler techniques to increase the number independent instructions (and thereby parallelism) in a given program. These include unrolling of loops, modulo scheduling, trace scheduling, reordering of instructions and more. Every architecture that aims to exploit parallelism benefits from such compiler techniques.

Once a program offers a certain amount of parallelism, there are multiple strategies of exploiting it. Those include overlapping executing of instructions, identifying and bundling independent instructions together or executing instructions in data-flow order rather than in control-flow order.

Similar to the performance increase by clock speed, the performance increase by ILP has hit a wall. That is due to the fact, that most architectures are Von Neumann architectures who employ a central register file. This register file becomes a bottleneck, because all instruction that is processed in parallel need to access it. Therefore, if one wants to increase performance by exploiting ILP even further, we need to focus on non Von Neumann architectures that do not have a central register file. Non Von Neumann architectures without registers include so called data-flow processors and so called exposed datapath architectures. In a data-flow processor computation is not driven my control flow (e.g. a program counter), but rather by the availability of operands (data-flow). In an exposed datapath architecture, the compiler has detailed knowledge of the processors circuitry, functional-units and their connectivity. Here the compiler schedules the movement of data on the chip in great detail, while computation occurs as a side-effect of data transport. This means increased complexity for the compiler, but can result in simple circuitry. However both these architectures come with their disadvantages and suffer especially from anything control flow related – that is branches and memory accesses (since memory must always be accessed in program order to preserve the programs semantic).

The Synchronous Control Asynchronous Dataflow (SCAD) architecture [BJS16; BS16] is a processor architecture, that shares features of Von Neumann, dataflow and exposed datapath architectures.
In the SCAD machine we have a pool of processing units (PU), each of which has input buffers (to buffer operands) and output buffers (to buffer) results. A processing unit fires whenever operands are available in their input buffers and produces a result that is put into the output buffer. This way of firing is data-flow. The SCAD machine is programmed by move instructions that move data from an output buffer of one processing unit to the input buffer of another processing unit (e.g. the result of a previous operation becomes the operand of the next operation). This way of moving data on the chip a feature of exposed datapath architectures. A move-program for SCAD is a list of move instructions that are fetched by a central control unit with a program counter (CU) and registered at the input and output buffers of the processing unit. This type of control flow is a Von Neumann feature.

In this work we compare four architectures that all aim to exploit instruction level parallelism – three of which are well established Von Neumann architectures with register files, and the SCAD machine. These architectures are (1) the pipelined architecture [Wat16], (2) the super scalar or dynamic scheduled architecture [Tom67], (3) the very long instruction word (VLIW) [Fis83] architecture and (4) the Synchronous Control Asynchronous Dataflow architecture[BS17].

Pipelining, super scalar and VLIW are all Von Neumann architectures, in the sense that they have a central register file and a program counter. Hence they suffer from the described memory wall. Pipelining attempts to exploit instruction level parallelism by overlapping execution of instructions. That is the next instruction can start, before the previous one is finished. Super Scalar attempts to exploit instruction level parallelism, by identifying dependencies between instructions and dispatching them to functional units as soon as operands are available. That means it presents the same interface to the compiler as the pipelined architecture, but tries to execute instruction in dataflow order internally. In VLIW the compiler identifies independent instructions and bundles them into larger instructions. Then one large instruction (multiple normal instructions) can be executed at the same time without conflict.

Our goals in this work are the following:

1. Compare the different approaches of the different architectures and identify their essence.
2. Especially, compare the SCAD architecture to the three well established ones, and get a better understanding where the SCAD architecture currently stands.
3. Identify parallels of SCAD with the other three architectures. That is

which step in the cycle of the SCAD machine can be found in one of the other architectures.

4. Pipelining, Super Scalar and VLIW all can make use of speculative executing in the form of branch prediction. This has always been a problem for data flow processors. Therefore we explore the possibility of branch prediction in SCAD and compare it to the other three architectures.

5. Implement a simulator framework that can be used to simulate all of the above mentioned architectures. There are already many processor simulator, many of which simulate implementation details of actual processors used in the industry. That makes them largely incomparable and not suitable for our analysis. Therefore we implement a framework that simulates the architectures at a higher level and where the four architectures share as much code as possible, in order to allow for easier comparability and easier analysis.

In chapter 2 (Preliminaries) we introduce each architecture in detail and explain how they work. That includes their modules, their execution cycle and notable an overview of their compiler tool-chain. In chapter 3 (Simulator Framework) we introduce our implementation of the simulator and give reasons why it was designed in that particular way. We also show how to use it and how execution traces can be analyzed and visualized. In chapter 4 (Analysis of the Architectures) we compare the different processor architectures and identify their similarities and differences. We identify the defining parameters and bottlenecks of each architecture. Special attention is given to the issue of branch prediction. We make the case that branch prediction is SCAD is not possible in the same way that it is in the other three architectures. We explain how branch prediction can be implemented, but why it will not speed up the executing significantly. Furthermore we give an outline of alternative techniques that can be used to overcome the disadvantage of no branch prediction. In chapter 5 (Conclusion) we summarize our work and give an outlook to what compilation approach should be explored to make the SCAD machine more competitive with the conventional established architectures.

1.1 Related Work

Several data-flow and exposed datapath architectures have been developed in the past. Those include most notably the *transporttriggeredarchitecture* (TTA) [Cor99], Raw [Lee+98], Trips [Bur+04], Flexcore [Thu+09] or Wavescalar [Swa+07]. The transport triggered architecture is programmed via move instructions and the actual computation occurs as a side-effect of data movement [Cor99]. However TTA has registers – instead if buffers like SCAD – at each processing unit.

After the SCAD machine was introduced several attempts at code generation have been made, including scheduling instructions with SAT solvers and SMT solvers.

2 Preliminaries

In this chapter we introduce the four architectures, first the Pipelined Architecture, then Super Scalar / Dynamic Scheduling, then VLIW and finally Synchronous Control Asynchronous Dataflow (SCAD). For each architecture we introduce the idea of exploiting ILP, give their modules, their execution cycle and sketch their compilation and code generation tool chain.

2.1 Pipelined Architecture

In a pipelined architecture instructions are executed in program order, but in an overlapping fashion – meaning, the next instruction can start while the previous one has not yet finished. That way multiple instructions are being worked on at the same time, hence exploiting ILP. This is achieved by splitting the execution cycle into multiple stages. Then in each cycle an instruction moves from one stage to the next. The external view (interface to the compiler) is largely a conventional Von Neumann architecture with a program counter and a central register file. It is worth noting, that instructions are not reordered, but instead executed in program order / control flow order. Parallelism is only achieved by overlapping execution. Hence this architecture is the easiest one and in some sense it can be found nested into the other architectures.

Conventionally the execution cycle is split into five stages [Wat16]. These are Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access (MEM) and finally Write Back (WB) and can be seen in figure 2.1. In the following we explain the function of each stage, explain how dependencies (conflicts) are handled and sketch the code generation approach.

1. Instruction Fetch (IF)
   This stage keeps track of the program counter (PC) and fetches the next instruction. If the pipeline must stall – due to a branch – the program counter is set to $\perp$ and updated when known. If this stage fetches a jump instruction it can update the PC immediately. If it fetches a branch instruction it must either stall or perform branch prediction.

2. Instruction Decode (ID)
   In this stage the instruction is decoded into its components. That is opcode and operands. If an operand is not an immediate operand (constant value) it is fetched from the central register file. That means this stage must wait until the register file is updated, or fetch the future register value from a later stage.

3. Execute (EX) In this stage any computation happens. That means this stage contains functional units like arithmetic and logic units (ALUs) and floating point units (FPUs). Not only the computation required by the opcode is done here, but also evaluation of conditions and memory offset computations.

4. Memory Access (MEM) All memory access – that is load and store operations – are performed in this stage. As mentioned earlier memory access must always be performed in program order / in control flow order, to preserve the programs semantics. This is ensured by performing all memory operations in one stage towards the end of the pipeline. Having this stage late in the pipeline means, that instruction which are under speculation, can advance until here before stalling.

5. Write Back (WB) In the final stage the result of computations and load operations is written back to the central register file. Again, similar to memory access, all write operations on the register file must be performed in program order. Having all write back operations in one stage towards the end of the pipeline makes this possible. Similar to memory access, instruction which are under speculation can advance until here before stalling.

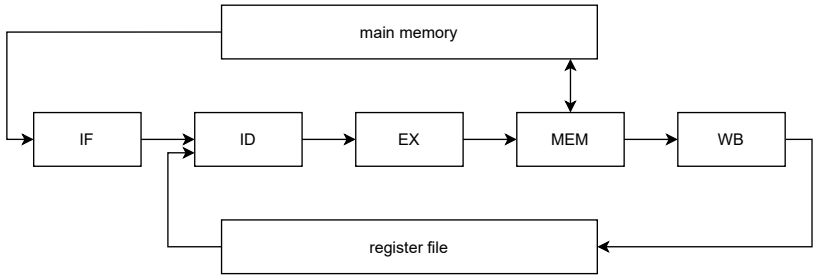

Figure 2.1: *The stages of the pipeline with register file and memory.*

2.1.1 Conflicts

If an instruction depends on a previous instruction – e.g. if the result of the first instruction is an operand of the next – they can not be executed in parallel. That is also true for the overlapping nature of the pipelined architecture. In that case the pipeline must stall by inserting a no-operation (NOP) into the stream of instructions and ensure that the first one finishes before the second one is executed. In the pipelined architecture these are called conflicts. We have four types of conflicts: read-after-write (RAW), write-after-write (WAW), write-after-read (WAR), read-after-read (RAR). In the following we explain each conflict between two instructions I1 and I2 where I1 comes before I2.

- Read after Write (RAW)
  If I1 writes to a register and I2 reads from it, then I2 must wait until the result of I1 is ready. That means I2 must wait in the ID stage, until I1 reaches the WB stage. Branch instructions can be seen as a special case of RAW, since the PC is just another register and the instructions updates PC at the execution stage. With the use of forwarding where the result of I1 is forwarded to I2 immediately once available (instead of waiting to reach WB) these conflict can be solved with only one NOP.

- Write after Write (WAW)
  If both I1 and I2 write to the same register, we must ensure that I1 does not overwrite I2. In the pipeline this is not a problem, since all write back operations occur at the end in program order. However we keep this conflict in mind for the super scalar and the VLIW case.

- Write after Read (WAR)
  If I1 reads a reigster and I2 writes the same register, we have to ensure that I1 reads the value before it is overwritten. In the pipelined case this is again no problem, since all instructions are executed in program order and I1 reads in stage ID while I2 writes in stage WB. However this conflict becomes importan for VLIW when bundling instructions.

- Read after Read (RAR)
  If I1 and I2 both read the same register there is no problem. This is case is only included for completeness, but poses no conflict.

As mentioned conflicts in a pipelined architecture can be resolved by inserting NOPs. This can be done by the compiler, however it is usually done by the processor during execution. This reomces the need for the compiler to be aware of the specific pipeline layout.

2.1.2 Code Generation

Code generation for a pipelined architecture – if ignoring NOPs as mentioned above – is the same as for any RISC instruction set and is well understood. Variables need to be mapped to registers in such a way, that variables which are alive at the same time, are mapped to different reigsters. This is achieved by graph-coloring heuristics.

2.2 Super Scalar Architecture

The Super Scalar architecture is a dynamically scheduled processor architecture that implements a Von Neumann architecture with registers. It uses the well known Tomasulo algorithm [Tom67] for out-of-order execution. It is dynamically scheduled, because the functional unit that will execute a given instruction, and the order of execution of instructions is determined to some degree dynamically at runtime. That is achieved by inserting instructions that should be executed into the reservation station, until their operands are all ready. Once that is the case instructions are dispatched to a pool of functional units. This is where execution can occur out of control flow order. Once execution of an instruction is complete the result is inserted into the reorder buffer. This buffer orders all results back into control flow order, before they are written to registers or memory. It is needed, since update of registers and memory must occur in control flow order, or else one would be introducing weak memory. In the following we explain each component of the architecture, sketch its code generation and discuss its limitations.

2.2.1 Reservation Station

After being fetched and decoded instructions are inserted into the reservation station where they await being dispatched to a pool of functional units. For that matter the reservation station is a table of entries, each storing

1. the operation that should be executed i.e. the opcode
2. the operands if they are available
3. otherwise, a reference to another reservation stage entry that will eventually produce the operands (in the form of the index of that other entry)

The entry

$$\texttt{[+ ; (}\perp\texttt{,2) ; (4,}\perp\texttt{)]}$$

means, that 1) an addition should be executed, 2) the first operand is not yet available ($\perp$), but will be produced eventually by the instruction in the reservation station at index 2 and 3) the second operand is available and its value is 4. Furthermore a flag $inUse$ is needed to signify whether or not a certain entry is free. Since entries must remain in the reservation station until no other component has a reference to it, an additional *dispatched* flag is needed to signify that an instruction has been dispatched, but the entry is still in use. If we were to remove entries from the reservation station on dispatch, the same slot could be filled again by a different instruction which could also be dispatched and might "overtake" the first one in execution. Under these conditions the index if an entry in the reservation station would not suffice as unique identifier to reorder and write-back results.

2.2.2 Reorder Buffer

Results from the executed stage can be produced out of order. Therefore they must be reordered into control flow order, before write back to memory or registers can occur. This is done by the reorder buffer. It is FIFO buffer, meaning all entries in the reorder buffer will occur in program order. Whenever an entry is inserted into the reservation station, a fitting entry is enqueued to the reorder buffer. That means entries in the reservation station an the reorder buffer occur in pairs. An entry in the reorder buffer stores a reference to its partner entry in the reservation station. That is used to insert results from the execution stage into their appropriate reorder buffer entry. Furthermore an entry in the reorder buffer stores where the results should be written. That can be a register, the program counter – in case of a branch or jump – or a memory location. Once the head of the reorder buffer is ready – its result has be inserted – it can be dequeued and the result can be written back. Besides the reference to the reservation station and depending on the type of instruction entries in the reorder buffer contain the following fields

- Register instructions:
  For instructions that write to a register – arithmetic, bit-wise, boolean or comparison instructions – the entry contains the registers index, that should be updated and a field where the results will be inserted (initially $\perp$).

- Branch instruction:
  For branch instructions the entry contains the two possible next program counters – then-pc, else-pc – and a field where the condition will be inserted (initially $\perp$). In case branch prediction is in use it also contains a field with the prediction that was made. If this type of entry is dequeued from the head of the reorder buffer the new PC is set according to condition. We discuss how branch prediction works more in depth later when comparing the architectures in chapter 4.

- Load/Store instruction:
  If the instruction was a load instruction the entry contains the register index that should be written to, and a field of the address. The address field is initially $\perp$ and is produced by the execution stage. If the instruction was a store instruction the entry contains the address field similar to the load case. Furthermore the value that should be stored is given in the same tuple representation as in the reservation station where `(x,`$\perp$`)` indicates that the value x is available, while `(`$\perp$`,y)` indicates that the value is not yet available, but will be produced by the instruction with reservation index y. Note that store entries in the reorder buffer might collect two results from the execution stage before they are ready.

2.2.3 Load/Store

Once an entry is dequeued from the reorder buffer memory and register update can occur. Performing memory access whenever a load/store entry is ready at the head of the reorder buffer would be correct, but not efficient. Since memory access is known to usually take longer time than other operations this would cause following entries to be dequeued and processed later. Instead ready load/store entries from the reorder buffer are dequeued and enqueued to the load/store buffer. This second buffer is also a FIFO buffer and serves in some sense as an extension of the reorder buffer. Its entries are only load/store instructions, have all their fields filled, are in program order and not under speculation, since they have been dequeued from the reorder buffer. Therefore we can perform memory access easily from the head of the load store buffer. With further optimization it is possible for load/store instructions to finish early without reaching the head of the load/store buffer:

- If a load instruction for address x is enqueued and there is a store instruction for address x present in the buffer, we can forward the value from there without performing the load instruction. This can be done easily since the buffer can contain only once store instruction for address x (see second optimization).
- Similar, if a store instruction to address x is enqueued and there is a store instruction for address x present, the existing one can be overwritten with the new one. Hence only the later of the two stores is performed. This is correct, since all load instructions that might occur between the two stores must have finished early with the above optimization.

Note that load instructions can now finish out of order and also write back to registers our of order. That is still correct, since the reference into the reservation station is always used to match whether or not a value should be updated (see register file).

2.2.4 Register File

The register file stores not only the current values of registers, but also which instruction will update a register and whether or not a register is currently up to date. This is done in a similar way as in the reservation station using tuples `(x,⊥)`, `(⊥,x)`. For example the register file

```
[(1,⊥); (⊥,2); ...]
```

means that register $r0$ is up to date and contains value 1, while register $r1$ is not up to date and will be updated by the instruction with reservation index 2. Note that this is the same representation as in the reservation station, hence decoding an instruction can simply lookup the tuple in the same way the value from a simple register file would be accessed.

When an instruction that will update a register is decoded and inserted into the reservation station index x the corresponding register is set to `(⊥,x)`. When a result is written back to the register file – either from the head of the reorder buffer, or from a load instruction – the value is only updated if the reservation index matches. That allows for our of order updating of the register file.

2.2.5 Code Generation

Code generation is not specific to the super scalar architecture, but only uses the instruction set. Code generated this way will execute on any implementation of that instruction set, whether it is a singly-cycle processor, pipelined, super scalar. Since dispatching instructions to the a free functional unit in the available pool of functional units happens dynamically at runtime, there is no need for the compiler to decide which instruction will be executed on which functional unit. In fact it does not need to be aware of the number of available functional units (opposite to code generation for the SCAD machine). This simplifies the process. The main challenge is mapping variables to registers. If there are more variable alive a certain time than available registers, some variables will have to be mapped to memory, which is known as spilling. Graph coloring heuristics are successfully used to map variables to registers while minimizing spilling.

2.2.6 Bottlenecks

The ability to dynamically dispatch – and by that reorder – instructions, the capability for branch prediction and the use of load/store buffers are the major advantages of the super scalar architecture. A known bottleneck restricting the number of instructions that can be executed in parallel however results from the centralized nature of the reservation station and the register file. The centralized nature of the reservation station restricts the number of instructions that can be dispatched in one cycle. The centralized nature of the reorder buffer restricts the number instructions that can take from the execution stage and sorted into the reorder buffer. The centralized nature of the register file restricts the number of values that can be written back on one cycle. Those bottlenecks are the ones that register-less architectures like SCAD aim to overcome.

2.3 VLIW Architecture

The Very Long Instruction Word (VLIW) architecture exploits ILP by bundling multiple independent instructions into one big instruction [Fis83]. All instructions in one bundle can then be executed in parallel. In that way the compiler implicitly reorders the instructions according to their data dependencies and can achieve some data-flow order. The parameter that is limiting parallelism is here the size of a bundle – that is the number of instructions in one large word. A naive implementation can simply have as many simple processor cores as the bundle size. These cores might also be of a pipelined nature, however that would increase the potential for conflicts and conflict detection and resolving (stalling) would have to be done across multiple pipelines.

When the compiler identifies independent instructions that can be executed in parallel, it is not enough to look at data dependencies (one instructions requires the result of a previous one). One must also account for instructions, that write to the same register. If we allow multiple such instructions in one bundle then only one of them (whichever comes last in program order) is allowed to actually update the register. Furthermore consider the following program:

```
I1 : r0 <- a + b
I2 : r1 <- c + r0
I3 : r0 <- d + e
I4 : r1 <- f + r0
```

Instructions `I1` and `I3` are independent in the sense that `I3` does not require the value computed by `I1`. But if we put them in the same bundle for VLIW then only the result of `I3` can be written back into `r0`, while the result of `I1` would need to be forwarded to the next bundle. This quickly becomes to complex and hence it is better to not allow instructions with the same target register in one bundle. Therefore "independent" in the VLIW sense is stronger than "independent" in the pipelined or super scalar sense. This also means that each bundle can contain at most one control flow instruction (jump or branch). That becomes clear when the PC is viewed as just another register. Jumps and branches write to PC and hence multiple jumps or branches are always in conflict.

2.3.1 Compilation and Code Generation

Since the notion of independence is stronger than just data-flow, it can be hard for a compiler to find enough independent instructions that can be bundled. Hence the VLIW architecture benefits especially from source code transformations that increase ILP like loop unrolling, modulo scheduling and trace scheduling. Then the compiler identifies independent instructions and bundles them – which is related to the famous bin backing problem. Doing so instructions are implicitly reordered, hence their order of execution might no longer be the program order, but rather closer to data-flow order. In contrast

to the super scalar architecture this scheduling happens entirely at compile time (static scheduling) greatly reducing the circuitry complexity.

2.3.2 Versions / Variations

Many VLIW implementations are not "pure" VLIW, where a simple RISC instruction set is taken and bundled, but additionally use predication. That means a bundle of instructions can be predicated with a condition. The instructions in the bundle are only executed if the condition holds true, and not otherwise. This way branches can be avoided. In a simple RISC architecture only single instructions can be predicated, so many predicated instructions would be needed to represent one branch. Hence this feature is usually left out, in order to keep the instruction set clean – it would double otherwise. In VLIW however one single flag can be used to predicate an entire bundle of multiple instructions, which makes this architecture very suitable for predication.

A VLIW processor can be implemented by simply using multiple simple processor cores. In order to waste less chip space it is also possible for the different pipelines to share the same pool of functional units.

2.4 SCAD Architecture

The SCAD machine – Synchronous Control Asynchronous Dataflow [Bha20; BS17; BJS16; BS16] – a non Von Neumann processor architecture, in the sense that it does not have a central register file. Instead it has a pool of processing units, that have input and output buffers where variables are stored. Processing units can consume values from their input buffers and produce values for their output buffers, similar to data-flow process networks. The machine is programmed by move-instructions, moving values from output buffers to input buffers. The computation occurs whenever processing units find inputs ready. Move instructions are issued in program order (synchronous control), while the actual movement of data and by extension the computation occurs whenever values are ready (asynchronous data-flow).

Like most non Von Neumann machines, that do not use registers, the SCAD machine aims to eliminate the already mentioned register bottleneck. Furthermore it is in some sense an exposed datapath architectures, since the compiler needs to be aware of all available processing units, and needs to move data between them. The production / consumption nature of processing units makes it in some sense a data-flow processor. At the same time it remains some control flow features, like a central program counter, used to issue move instructions in program order. A possible use case of the SCAD machine is on accelerator cards. When it is not feasible to compile an entire program to an FPGA bitstream, due to size limitations, the SCAD machine might offer an intermediate solution, where the program is compiled to move instructions

while a SCAD machine with a certain number of processing units is compiled to an FPGA bitstream.

In the following we describe all components of the SCAD machine, how move code is generated and what some known issues and bottlenecks are.

2.4.1 Move Code

A move instruction is always of the form "`src -> dst`", meaning data should be moved from a source to a destination. A source can be an output buffer of a processing unit, or an immediate value. Destination can be an input buffer, or `discard` if a value should be dropped. The instruction `pu0@out -> pu1@in0` means a value from the output buffer of processing unit 0 should be moved to the input buffer with index 0 of the processing unit number 1. The instruction `16 -> pu2@in1` means the immediate value 16 should be moved to input buffer number 1 of processing unit number 2. For better readability input buffers can be named i.e. an input buffer used for opcodes would be named `opc` instead of `in0`. Then the instruction `(+,1) -> pu0@opc` means the immediate value (+,1) should be moved to the input buffer of processing unit 1 named `opc`. The universal processing unit described below makes use of such an opcode buffer.

2.4.2 Processing Units

The actual computation occurs in the processing units. The correspond to the execution stage of the other architectures. In this work we differ between functional units (FU) and processing units (PU). With functional units we mean units that perform a certain computation like arithmetic and logic units (ALUs), floating point units (FPUs) or similar. With processing units we mean the components of the SCAD machine, which consist of a functional unit combined with buffers and a firing rule. Processing units in SCAD can implement any functionality with n inputs and m outputs. A processing unit has n input buffers and m output buffers which are FIFO in nature. When values are available the head of each input buffer and the output buffers have enough space to store the result, operands from the input buffers are consumed and the processing unit fires. The produced result is then placed into the output buffers. Usually we use so called universal processing units, that implement different operations and where one input buffer represents an opcode. However it is also possible to have specialized processing units for certain operations. That way the SCAD machine can be adjusted to specific applications.

The universal processing units – that were implemented in previous simulators – implemented all available opcodes. Now we implement different types of processing units that offer different operations with different latencies in order to mirror the available functional units in other architectures like the super scalar one (ALU, FPU, ...). In 2.2 the execution of a processing unit is shown. In the first step one operand is still missing. The opcode signifies,

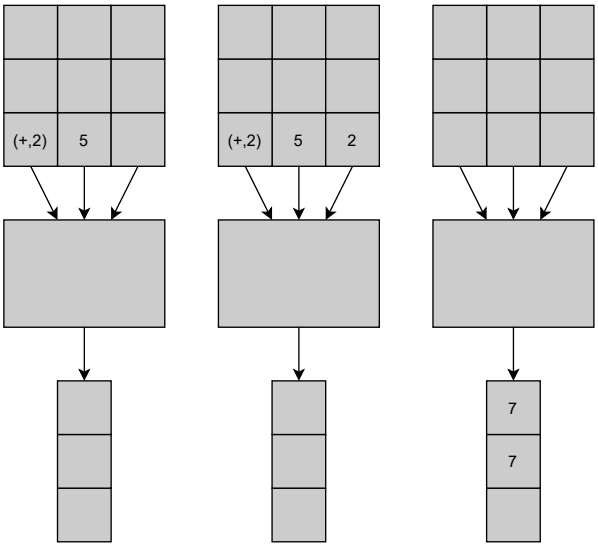

Figure 2.2: *Exampel execution of a processing unit in SCAD.*

that an addition should be performed, and that two copies of the result are required. In the next step the operand arrives and the head of the input buffer is ready to be dequeued. The last step shows the PU after firing.

2.4.3 Input Buffer

Input buffers are not just simple FIFO buffers (as mentioned above), but also perform matching of operands. Input buffer entries are tuples of the form (val, src) where val is the actual value that is stored, and src is the buffer address where val came from. $(\perp, \perp)$ signifies an empty entry. $(val, \perp)$, where the value is present, but the source is not, means that his value is an immediate value, i.e. registering the move instruction `4 -> pu0@in0` will enqueue $(4, \perp)$ to the input buffer, meaning it is placed in the first empty slot closest to the head. An entry of the form $(\perp, src)$ where the source is present but the value is not signifies that the value still has to arrive from src. Once it arrives it will be inserted into this entry. For example registering the instruction `pu0@out -> pu1@in1` enqueues $(\perp, pu0@out)$ in pu1@in1. Meaning this entry is placed in the first free slot closest to the head.

2.4.4 Output Buffer

Similarly output buffers are not just simple FIFO buffers, but also perform matching of values with their intended target address. Output buffer entries are tuples of the form (val, tgt) where val is the actual value that is stored, and tgt is the buffer address where val should be send to. $(\perp, \perp)$ signifies an empty entry. $(val, \perp)$, where the value is present, but the source is not, means that this value was already produced by the respective processing unit, but is not yet know where it should be send to. An entry of the form $(\perp, tgt)$ where the target address is present but the value is not, signifies that the value

still has to be produced by the respective processing unit. Once it becomes available it will be inserted into this entry.

Consider the instruction `pu0@out -> pu1@in1` being registered at the output buffer of PU0. If there is an entry of the form $(val, \perp)$ in the output buffer, then the entry of that form closest to the head is chosen and the target address is inserted, yielding $(val, pu1@in1)$. If such an entry is not present, then the first free slot closest to the head is chosen and $(\perp, pu1@in1)$ is inserted.

2.4.5 Control Unit

The control unit (CU) is a special unit, that stores a program counter and is responsible for fetching move instructions and registering them at the input and output buffers of processing units. That is done via the move instruction bus (MIB) over which the control unit can broadcast to all input and output buffers. Similarly to the VLIW architecture the control unit in SCAD can fetch multiple instructions – that do not interfere in the sense that they are registered at different buffers – and register all of them in parallel. Recall that a classical RISC instruction (two operands, one result) corresponds to three move code instructions (move first operand, move second operand, move opcode). This increase in code size – not considering computational overhead that we will see later – already motivates why this is necessary in order to achieve any competitive runtime.

The control unit handles control flow by storing a program counter. If PC equals $\perp$ it means the control unit is stalling and no instructions are fetched. Jumps and branches are handled in the following way. There is an input buffer *jmp* for jump targets and input buffers $then - PC, else - PC$ and *cond* for branches. If a value is available at the head of *jmp* it is consumed and PC is updated accordingly. For now jumps are only implemented with immediate operands. Allowing any PU to send a value to `cu@jmp` can be used to implement jump tables of function pointer semantics. In that case, if an entry of the form $(\perp, src)$ is enqueued to *jmp* the CU must stall and wait. Branches are implemented similarly. When a value is enqueued to $then - PC$ the CU sets PC to $\perp$ (stalling) and enqueues the old PC + 1 to $else - PC$. As soon as *cond* has a value ready at its head, the tuple $(then - PC, else - PC, cond)$ is consumed by the CU and the new PC is set accordingly to either $then - PC$ or $else - PC$.

2.4.6 Load/Store Unit

Memory access in the SCAD machine is performed by the load-store unit (LSU). It is a special processing unit with input and output buffers. Instead of computing values it can access memory. It has three input buffers *opcode*, *address* and *value*. The opcodes can be either *load* or *store*. Similarly to other PUs the LSU fires when operands are available. If the opcode is *load* only input buffers *opcode* and *address* are consumed. The result will be placed in the LSU's output buffer. If the opcode is *store*, then *address* and *value* are

both needed and consumed. No result is stored in the output buffer. Similarly to the universal processing unit the LSU opcode has an additional constant, that signifies how many copies should be produced – i.e. $(load, 3)$ means that 3 copies of the loaded value will be placed into the output buffer. For store operation the constant can be ignored.

2.4.7 Data Transport Network

The Data Transport Network (DTN) connects all output buffers (n) to all input buffers (m) and is a n to m routing network. It is responsible for transporting values from output buffers to input buffers. If the head of an output buffer with address src is ready – and of the form (val, tgt) – then it can be dequeued and dispatched into the DTN. The DTN will route (src, val, tgt) to the input buffer with address tgt. There it is guaranteed to find an entry of the form $(\perp, src)$ with the value missing. The value is then inserted into the first such entry closest to the head. It is guaranteed to find such an entry, since the move instruction `src -> tgt` was registered in both – the tgt buffer as well as the src buffer – in the same cycle.

Since it is possible that multiple output buffers want to send a value with the same target address, the DTN needs an arbiter to choose one at a time.

2.4.8 Code Generation

Generating move code for the SCAD architecture turns out to be a difficult problem that is not yet fully solved. Operations need to be mapped to PUs – and thereby variables are implicitly mapped to the output buffers, where they can be found.

The three Von Neumann architectures detailed above all have a central register file, from which values can be read any number of times. Due to the production and consumption nature of the SCAD machine, the code generator needs to know how many copies for each variable will be needed and produce the right amount. That is especially challenging considering control flow: i.e. two branches must below the same split point must consume the same number of copies for each variable. Similarly the right number of copies must be provided in loops. These problems are known as balancing and loop bounding and can be solved using well known source code transformations from like SSA and SSI form [Sch18].

Again the three Von Neumann architectures detailed above all have a central register file, which allows random access to each register at any given point in time. Due to the FIFO nature of buffers in the SCAD machine, it is not possible to access any variable that is needed. In fact there are situations when the required variable is not at the head of its output buffer. Then another variable must move out of the way, to allow access. This is known as a buffer-interference between the two variables and can not always be avoided. In our current move code generators, this adds significant computational overhead.

In the early stages of move code generation for the SCAD machine SAT-solver and SMT-solvers [BS16] were used to find the minimum number of processing units necessary to execute a program without such overhead. However realistic scenarios are of a resource constrained nature, where the pool of available PUs is given, and one must find a mapping that exploits ILP while minimizing overhead at the same time. The move code generator we use here achieves this naively by moving values out of the way and re-enqueuing them (duplicating them), until the next required value is at the head of its output buffer.

3 Simulator Framework

We implemented a simulator framework that allows us to simulate multiple processor architectures in F# for the MiniC and Abacus languages. MiniC and Abacus are a minimalistic C-like language and a minimalistic RISC assembler respectively, that are developed by the Embedded Systems Group in Kaiserslautern and used for research and teaching purposes [BSS15].

In order to compare the architectures and analyze the current state of the SCAD machine and its code generation, the simulators need the following properties:

- Cycle accurate and comparable:
  The running time of the simulators should be comparable in the sense, that given similar code, they take similar time. Furthermore we need to ensure that both do similar amount of work in one cycle, in order to ensure a fair comparison.
- Logging and visualizing:
  In order to analyze why the processors perform in a certain way we need log information like the work load on functional units, the time spent waiting for values from the DTN, when and for what reason stalling occurs and similar data.
- Configurable:
  All features important for our comparison should be easily configurable. That includes the number and kind of functional units, size of buffers and reservation station, whether or not to use branch prediction and many more.
- Extendable:
  Since there exits other variations of the SCAD machine, and other architectures against which we could compare SCAD, we want our simulator framework to be easily extendable, in the sense that we can take existing code shared between the simulators and combine it with little effort to form a new simulator for a different architecture.

Due to the objectives above we decided against using available super scalar simulators for real world languages and architectures, but instead developed a super scalar simulator for Abacus alongside a SCAD simulator. For that matter the namespace `MiniC.ProcSim` contains all code shared by all simulators, which includes functional units, branch predictors, memory and caches.

The actual simulators can be found in `MiniC.ProcSim.SCAD`, `MiniC.ProcSim.Abacus3O`, ... respectively. All functionality that can be

shared can be found in the common namespace, while only functionality specific to the architectures is in their respective namespaces. That will make it relatively easy to add additional architectures or different versions of SCAD in the future.

The simulators are run using

```
let sim = MiniC.ProcSim.Abacus3O.Simulator.Create prog config
let sim = MiniC.ProcSim.SCAD.Simulator.Create prog config


let stats = sim.Execute (Some 1000) (Some myCallBack)
```

where `prog` is the program to be executed – move code or Abacus code respectively – and `config` is a set of JSON strings containing the configuration. The first parameter to `Execute` specifies the maximal number of cycles that should be simulated. If `None` is supplied the simulation runs until the program terminates. The second parameter is an optional callback function that will be called in each cycle. It can be used to i.e. log the current states for visualization. Execution returns an array of statistical data (`stats`) i.e. which functional unit was working at which cycle, or whether or not the processor is stalling and for what reason.

As mentioned earlier we differ between functional units (FU) and processing unit (PU) where FUs are merely units that can perform a certain set of operations, while PUs are SCAD units consisting of an FU and buffers. This is also reflected in the simulator design, as can be seen in the following code sample:

```
type PU = {
  opc    :  InputBuffer<int*int>
  inputs :  InputBuffer<uint32>[]
  output :  OutputBuffer<uint32>
  fu     :  MiniC.ProcSim.FuncUnit<int>
}
```

As seen above a PU consists of buffers and an internal FU. The type for FUs is the same type that is used for the pool of FUs in the super scalar case, thus ensuring that both simulator perform their computation in the same way (since it runs the same code). This design pattern is used as often as possible in the simulator framework to ensure that only those features where the architectures differ are different code – giving us good comparability.

3.1 Configuration

The simulators are configurable via JSON files. Similar to the code design above they share as much of the configuration as possible. For example the configuration for the types of available FUs and what operations with what latencies they provide can be reused for all simulators. To that end we allow to give multiple configuration files when running the simulators. The program will combine those together to a complete configuration (overwriting earlier configurations with later ones if necessary). That way we can share the same configuration for the pool of FUs, memory, caches, branch predictors and more in one file, while having an additional configuration file with the simulator specific parameters for each one respectively.

The following is an example of a configuration for a pool of available functions units. It defines two types of units with their set of operations and latencies. Then defines how many of which type are present in the pool. This configuration can be used for both, the super scalar as well as the SCAD simulator.

```
"fuTypes" : [
    {   "name" : "ALU_TYPE1",
        "lanes" : [
            {   "latency" : 1,
                "ops" : [ "==B", "!=B", "!B", "&B", "|B",
                    "==N", "==Z", "!=N", "!=Z", "<=N", "<=Z",
                    "<N", "<Z" ]  },
            {   "latency" : 2,
                "ops" : [ "+NF", "+ZF", "-NF", "-ZF",
                "+N", "+Z", "-N", "-Z" ]  }
        ]   },
    {   "name" : "ALU_TYPE2",
        "lanes" : [
            {   "latency" : 4,
                "ops" : [ "*NF", "*ZF" ]  },
            {   "latency" : 8,
                "ops" : [ "/NF", "/ZF" ]  }
        ]  }
],
"fuPool" : [
    { "type" : "ALU_TYPE1", "number" : 4 },
    { "type" : "ALU_TYPE2", "number" : 2 }
]
```

3.2 Command Line Interface And Workflow

For each simulator we have one executable with a command line interface. Consider the following command.

```
./scadSim moveLabel myCode.mov 10000 config1.json config2.json
```

This calls the simulator for the SCAD architecture. The first parameter specifies the input format of the program. The input program can be in MiniC, Cmd-Code, Cmd-Code with labels, move code or move code with labels. The next parameter is the program to be executed, and then the maximum number of cycles that should be simulated. Afterwards one can list as many configuration files as needed.

In case the input program is specified as MiniC or Cmd-Program the tool will perform compilation and code generation and make those files available for further editing and use.

3.2.1 Execution Trace

Running the simulators produces an execution trace, which is is – similar to the config files – saved in JSON format. The execution trace with first list the configuration that was used (combined from multiple configuration input files), then the program that was used and finally a sequence of all steps with the entire machine state for each step.

This trace file can be used to query information about the performance e.g. how often the machine stalls and for what reason, how many FUs are firing at any given point. It can also be used to visualize the execution.

3.2.2 Visualization

We implemented a visualization, that is a web-page – or simply an HTML file that can be viewed in any browser – which can be used to observe the execution step by step. The interface allows to load an execution trace (JSON file) and then uses the information in that trace to show the program, the machine in its configuration and the state in each step. It allows running the simulation as animation, or stepping through it one cycle it a time.

4 Comparison Of The Architectures

In this chapter we compare the four processor architectures and analyze their differences and similarities. To that end we explore each machines capability for our of order execution and speculative execution and identify their limiting factors.

First we explore how much of the work is done in the compiler (statically) and how much is done during execution in the processor (dynamically) for each architecture. Then we compare the architectures pairwise in order to point out important details in their differences and similarities. After that we look at each machines capability for out of order execution and speculative execution. Here we are especially interested to see how the SCAD machines compares to the three well established Von Neumann architectures. We make the case that speculative execution by means of branch prediction is possible to some extend in SCAD, but not advisable. This serves as motivation to explore alternative means of speculative execution and dealing with control flow in general for the SCAD machine. Finally we use our simulator framework to compare the performance of the SCAD machines current code generator against the Super Scalar machine which again motivates which compilation techniques for SCAD should be explored in the future.

4.1 Static Work and Dynamic Work

Processor architecture can be arranged in a hierarchy according to how much work is done at compile time (statically) and how much work is done during execution (dynamically).

1. The super scalar architecture and the pipelined architecture are at the top of this hierarchy, since the compiler needs very little knowledge about the internal implementation of the processor. For both cases the compiler needs to match variables onto registers and produce a linear stream if RISC instructions. Then during execution the pipelined architecture needs to identify conflicts and insert NOPs while the super scalar architecture needs to identify data dependencies and schedule instructions accordingly onto the pool of available functional units.

2. In the case of the VLIW architecture the compiler needs more detailed knowledge about the target processor and needs to perform more work. It needs to know how many pipelines there are (how many instructions per bundle) and then analyze data dependencies, reorder instructions and assembly them into bundles. In return the processor can be simpler,

meaning less circuitry besides the actual functional units. As mentioned a VLIW processor can be implemented by using multiple simple cores.

3. For the SCAD machine the compiler needs to have even more detailed knowledge about the particular target processor and needs to perform even more work. It needs to know not only the number and type of available processing units, but also the sizes of input and output buffers. Then it needs to map operations to processing units and resolve buffer interferences. Finally it also needs to assemble multiple move instructions into bundles in a similar fashion as the VLIW architecture. The work performed during execution – besides the actual computations in side the processing units – happens largely in the buffers (matching if values and addresses) and in the data transport network (routing of values). Since the computation in SCAD happens as a side effect of data transport and availability of operands, it is only expected that the most work and circuit complexity will be found in the data transport network and the input and output buffers.

4.2 Pipelined and Super Scalar

On first sight the pipelined architecture and the super scalar architecture seem very similar, in the sense that they expose the same interface to the compiler. Code that is compiled for a simple RISC instruction set can be executed on a simple single cycle CPU as well as on a pipelined or a super scalar implementation (if NOPs are inserted by the processor and not by the compiler). This transportability of binaries has led to a huge popularity of both architectures in real world applications. Furthermore the pipelined architecture can be seen as an extension of a single cycle CPU while the super scalar architecture can be seen as an extension of the pipelined architecture. However their are important differences in their paradigm execution.

The super scalar architecture can reorder instructions dynamically and execute them in data-flow order, while the pipelined architecture must always execute instruction in program order. That means the super scalar architecture can perform significantly better if a program offers a lot of parallelism.

On the other hand there are cases where the pipelined architecture can perform better than the super scalar architecture. That is the case with a linear sequence of instructions where each instruction depends on the predecessor. In the super scalar case only one instruction can be dispatched to the pool of functional units at a time. Then the result must be inserted into the reorder buffer, and from there can be forwarded to the register file as well as to the next instruction which is waiting on operands. In the pipelined case there is no reorder buffer and the result can be forwarded directly from the execution stage to the instruction decode stage. This constitutes a trade off between out of order capability and easier forwarding.

Both architectures can perform speculative execution by means of branch prediction. In the pipelined case instructions under speculation can proceed until the memory access or the write back stage (depending on whether it is a load/store or register operation). In the super scalar case instruction under speculation can proceed until the head of the reorder buffer. In that sense both architectures are similar again.

4.3 Super Scalar and VLIW

The super scalar architecture and the VLIW architecture can be seen as somewhat opposite in nature, while offering very similar features. The first schedules instructions dynamically during execution, resulting in an easier compilation at the price of increased circuitry. While the latter schedules instructions statically at compile time, resulting a more complex compiler and significantly less complex circuitry.

However both offer the possibility for out of order executing in a data-flow manner and speculative execution. Furthermore both suffer from the same bottleneck of a central register file.

4.4 Super Scalar and SCAD

The comparison between the super scalar architecture and the SCAD architecture is the most interesting for us, since the super scalar architecture is the most successful and most widely adopted standard. Therefore we want to explore how the current state of the SCAD architecture compares against it. Furthermore both architecture expose a control flow interface to the compiler side – meaning a linear sequence of instructions, a program counter and jump and branch operations – while aiming to execute instructions in data-flow order internally. Hence many similarities can be expected.

4.4.1 Reservation Stage and Input Buffers

Consider that the execution stage in the super scalar case (pool of functional units) is encapsulated by the reservation station on one side and the reorder buffer in the other side. Instructions wait in the reservation station until their operands become available. Then they wait in the reorder buffer until their results can be send back to the reservation station and become operands for the next instructions. Consider now, that the processing units in the SCAD case are encapsulated by input buffers on one side and output buffers on the other side. Instructions wait in the input buffers until their operands become available Then they wait in the output buffers until their results can be send back to input buffers and become operands for the next instructions.

It becomes clear that the reservation station corresponds to input buffers, while the reorder buffer corresponds to output buffers. Therefore a fair comparison will have to balance the size of the reservation station with the size of input buffers and output buffers. However there are important differences. In the SCAD machine as many instructions as there are processing units can be dispatched from input buffers to execution at one time. In the super scalar case the bandwidth between reservation station the pool of functional units is limited. Not only is the bandwidth limited by the number of register (which are not present in SCAD), but also by the complexity of considering the entire reservation station at once (instructions must be chosen from all slots of the reservation station). This constitutes an advantage of SCAD over the super scalar architecture. On the other hand the super scalar architecture can dispatched any instruction from the reservation stage to any available functional unit in the execution stage, while the SCAD machine can only dispatched instructions from the respective input buffers to the corresponding processing unit. Consider a situation where two instructions in an input buffer are ready to be executed. Only the first one can be dispatched, while the next one will be dispatched in the next cycle. In the super scalar case both instructions can be dispatched from the reservation station to two separate functional units. Consider furthermore a situation where the head of an input buffer is not ready, but an instructions higher up in the input buffer is ready. Here the instruction can not be dispatched, but must wait. In the super scalar case the instruction could be dispatched immediately. This is due to the FIFO nature of SCAD buffers and constitutes an advantage of the super scalar architecture over SCAD. Both these scenarios can be avoided by carefully scheduling operations to processing units during the compilation for SCAD. Again SCAD has the potential to perform better, but needs to perform significantly more work during compilation to achieve that.

In the extreme case of a single functional unit the super scalar machine has an obvious advantage over the SCAD machine. The super scalar machine can issue any instruction from the reservation station that is ready to the single functional unit. The SCAD machine however can not do that, and must therefore schedule accordingly.

4.4.2 Memory Access

In the super scalar architecture load and store operations that reach the head of the reorder buffer are enqueued to the load-store buffer, then the actual memory access proceeds from there. Similarly load and store operations in the SCAD machine are enqueued to the input buffers of the load-store unit. Both – the load-store buffer, as well as the LSU input buffers – work in a FIFO manner. However there are significant differences. The LSU input and output buffers in SCAD also perform matching of operands, similarly to the reservation stage and the reorder buffer in the super scalar case, and in that sense, are more complex than the load-store buffer. On the other hand the load-store buffer can reorder or even skip instructions which is not possible with the LSU. If two store operations with the same target address are present

in the load-store buffer, then the first one can be omitted and only the second one is kept. If first a store operation and after that a load operation with the same target address is present in the load-store buffer, the value can be forwarded without performing an actual memory access. Therefore it might be advisable to employ load-store buffers in the SCAD LSU. In the first case – where two store operations with the same address are present – a load store buffer inside the LSU could avoid a second memory access without knowledge of the SCAD machine. In the second case – where a store and load operation with the same address are present and the value could be forwarded – the LSU can not do so, since the loaded value must be enqueued to the head of its output buffer. Hence, if we want to make use of a fully functional load-store buffer inside of an LSU, we would need to change the input and output buffers of the corresponding LSU. One could match entries in the input and output buffer pairwise with an additional token. Then, when a value is loaded, it is placed into the slot with the corresponding token instead of enqueued to the head of the output buffer. In that sense the input output buffers would perform reordering of memory accesses that were internally performed out of order. In the super scalar case loaded values are forwarded to the reservation station as well as the register file. Therefore the performance increase gained by load-store buffers carries over. In SCAD however, loaded values are only placed in the output buffer and are forwarded to input buffers only later. Dispatching loaded values from the output buffer to the DTN must still occur in FIFO manner. Hence the increase in performance that could be gained by employing load-store buffers inside of an LSU only carries over to the corresponding output buffer of that LSU, but not any further.

4.5 Out of Order Capability

All four architectures attempt to increase performance by exploiting parallelism. However only the super scalar, the VLIW and the SCAD machine to so by executing instruction out of order – that is in a different order than specified in the program. The pipelined architecture always executes every instruction in program order.

The super scalar machine can execute instruction in data-flow order since instructions are dispatched to the execution stage as soon as their operands become available. However this is limited by the size of the reservation station, the bandwidth between reservation station and the execution stage and the number of available functional units. The number functional units does not pose a bottlenecks intrinsic to the architecture. That means it can be increases without affecting the the architectures interface or without significant increase in complexity. These functional units need to be fed by the reservation stage. This is where the architecture has an intrinsic bottleneck. As mentioned earlier it suffers from the same bottleneck that all Von Neumann architectures share (the number of registers). Furthermore as many slots of the reservation station need to be considered when dispatching instructions as the size of the

bandwidth. Therefore an increase in bandwidth poses a significant increase in complexity.

The VLIW architecture can execute instructions out of order (and even in data-flow order) by reordering and scheduling them statically at compile time. However the size of one bundle poses an intrinsic bottleneck. The size can not be arbitrarily increased, since this would result (1) in many cores and (2) in situations where bundles can not be filled and have to be padded with NOPs.

Among the considered architectures, the SCAD architecture offers the greatest potential for out of order execution, since it does not suffer from the above mentioned bottlenecks. There is no central register file. Furthermore only the respective input buffers have to be considered when dispatching instructions and multiple PUs do not interfere with each other in this manner. However the DTN and the MIB both pose a bottleneck for out of order execution. With an increase in the number of processing units the DTN must also grow. As mentioned earlier the DTN performs a significant part of the SCAD machine's work. Therefore it is vital to find DTN implementations that scale with the number of processing units. Such implementations have been found and are seem promising to not becoming a bottleneck [Jai19]. Similarly an increased number of processing units can be explored only if the CU supplies enough move instructions. Therefore width of the MIB and the fetch width must be considered. As mentioned before we can bundle move instructions into larger words similar to the VLIW architecture.

4.6 Speculative Execution

Speculative execution is the idea of executing instruction prior to knowing if they should be executed at all. That occurs at branches, when it is not yet clear whether or not a branch should be taken. The most common technique of speculative execution is branch prediction. In the following we briefly explain how branch prediction works in the pipelined, super scalar and VLIW cases and afterwards explore the possibility of branch prediction in SCAD.

4.6.1 Branch Prediction

Speculative execution in the form of branch prediction is an important feature of modern super scalar processors. Instead of stalling whenever a branch is encountered a prediction – taken / not taken – is made and execution can continue. Once the branch condition becomes available and the prediction turns out to be correct, execution can continue normally. If the prediction turns out to be incorrect, all changes since the prediction need to be undone (rolled back). Therefore computations that occur under prediction can not be allowed to override registers or memory. This is achieved by allowing speculative execution to propagate only until the head of the reorder buffer. Whenever a prediction is made, an entry with the prediction is placed in the reorder buffer.

If that entry reaches the head of the buffer while the actual branch condition is still not available this entry can not be dequeued. When the condition becomes available it is compared to the prediction in the entry in the reorder buffer. If it matches the entry is no longer under speculation and clear the reorder buffer. If an entry with a mismatching prediction reaches the head of the reorder buffer, all changes must be rolled back. That is easily achieved by flushing the reorder buffer, the reservation station as well as the execution station and finally updating the program counter.

Similarly branch prediction can be achieved in the pipelined and VLIW case. One must only ensure that instructions under speculation are not written back to registers or memory. If wrong prediction is made, the entire pipelined can be flushed.

Control flow – and in particular stalling at branches – has always been a bottleneck for data flow architectures. That is also true for the SCAD machine. Therefore we want to explore the possibility of implementing branch prediction in SCAD. In the following we sketch how branch prediction can be implemented by means of an extended control unit. Then we point out the limitations and bottlenecks of this approach and make the case that branch prediction in SCAD – while possible to some extend – is not advisable. Finally we list alternative options, how the bottleneck of control flow can be overcome by the SCAD machine. Those will have to be explored in future work.

Branch Predication in SCAD

Recall the functionality of the control unit in SCAD. There is an input buffer *jmp* for jump targets. If a value is enqueued to that input buffer, PC can be immediately updated to that new value. Furthermore there are input buffers $then - PC$, $else - PC$ and *cond* that are used for branches. When branch instruction is fetched, the then-PC is enqueued, and else-PC is filled by the CU itself to current PC + 1. The condition buffer receives only the source address in the form `(srcAddr,`$\perp$`)`. Then PC is set to $\perp$ and execution stalls. When the condition eventually arrives via the DTN it is put into the condition buffer. Then the tuple of (then-PC, else-PC, cond) can be consumed by the CU and the program can be updated.

We now propose an extended CU 4.1 that has an additional buffer *spec* to store predictions for speculative execution. When a branch instruction is fetched everything described above occurs. But additionally a prediction is made and the result is put into the spec buffer. Then the CU peeks – not consumes – then-PC else-PC and spec and sets the new PC accordingly without stalling. When the condition eventually arrives the tuple of (then-PC, else-PC, spec, cond) can be consumed the CU. If condition and prediction match, execution can continue. If they mismatch, the prediction was wrong and the new PC needs to be set according to the condition. Furthermore all changes must be rolled back.

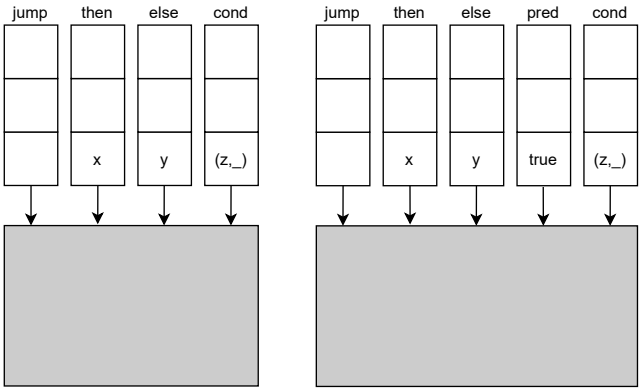

Figure 4.1: *A normal CU (left) and a CU with branch prediction (right). In both cases the condition will arrive from buffer z. On the left the CU is stalling. On the right it continues with branch x, since the speculation was "true".*

In order to roll back changes the CU must broadcast to all input and output buffers as well as all PUs whenever a prediction is made, and whenever the outcome of a prediction becomes known. A naive way of implementing that is to add an additional field to each data-value pair. The additional field $speculation - depth$ signifies if the entry is under speculation, and if so on which level. The size of the CU input buffers implicitly become the maximum speculation depth. A depth of zero means that an entry is not under speculation.

Whenever a prediction is made the current speculation depth is incremented and all further entries – that are registered by the CU at the input and output buffer via the move instruction bus – will receive the new speculation depth. When a prediction turns out to be correct the CU must broadcast along the MIB and all speculation depths are decremented (not further than zero). When a prediction turns out to be wrong the CU must broadcast again along the MIB and all entries with a speculation depth greater than zero discarded.

In a simulator this concept is easy to implement for the sake of testing and comparing. However this results in a significant overhead for real hardware. A different approach is to have pointers (the same number as the maximum speculation depth) for each input and output buffer, that point into the buffer, to where the speculation has been made. If no entries are under speculation, then all pointers are set to $\perp$. Once a speculation is made the next pointer in line is set to the tail of its respective buffer. If a prediction turns out to be correct the first pointer can be reset to $\perp$ and the next pointer becomes the head of the pointer FIFO. In that way the pointers also implement a ring buffer. If a prediction turns out to be wrong all entries above the pointer must

be discarded and then all pointers reset to $\perp$.

As mentioned before branch prediction is only feasible if the processors state can be rolled back. In Von Neumann architectures with a central register file this is achieved by propagating instructions under speculation only until write back to memory of registers takes place. Similarly entries in the SCAD machine that are under speculation can only propagate until a point where rolling back becomes impossible. That point turns out to be the head of input buffers. This is due to the fact, that values from the head of input buffers are consumed. If values are gone they can no longer be restored during a roll back.

Consider the scenario where two value with different speculation depth reach the head of an input buffer. If the processing unit were to consume them and one of them turns out to be wrong, while the other turns out to be correct, then correct one should have stayed in the input buffer to wait for a different partner. One might imagine a scenario where both values have the same speculation level. In that case they could be consumed by the processing unit and speculative execution can occur. Then the pointers described above would not only point into the buffers but also into the stages of the processing unit. However it is unreasonable to assume such a scenario can occur. Consider a simple branch with one basic block above the split point – that produces some values – then two successor basic blocks. If an operation in any of the branches uses a value from above the split point, then their speculation depth will not match (speculation depth of the opcode and of the value). It is only possible for all operands and the opcode to have the same speculation depth, if all of them come from withing the same basic block. That means they are either all immediate operands – in which case the computation could have been made by the compiler using constant folding – or one of them comes from a load operation (but memory access should not be performed speculatively). Therefore the case were all operands and opcode at the head of a input buffer have the same speculation depth greater than zero should not occur in a real program.

We now know that entries under speculation can only proceed until the head of input buffers and then must stall. That means the SCAD machine can not perform speculative execution by branch prediction, but only speculative registering of move instructions. This correspond to the insertion of instructions under speculation into the reservation station of a super scalar architecture. The super scalar architecture however can also dispatch them to the execution stage, even if the operands come from different speculation depths. Therefore it is only reasonable to revisit the idea of branch prediction in SCAD if it turns out that registering of move instructions via the MIB is a bottleneck over the actual computation.

4.7 Experiments

Using the simulator framework and the current move code generator we can run experiments to try and find out where SCAD currently stands. Unfortunately the current move code generator is in no way competitive and most experiments carry little to no information. However we want to highlight two experiments that show (1) how much overhead is added by the current move code generator and (2) that using branch prediction in SCAD gives indeed almost no increase in performance.

Size of Generated Move Code

The two move programs in figure 4.2 are both a simple loop that just counts down a value. The left one is generated, while the right one is written manually. It can be seen that even for a simple program with no interferences the move code generator produces 2.46 times as many instructions – most of which are duplications and discards.

Performance Increase with Branch Predication

As described earlier we have implemented branch prediction for the SCAD simulator, however concluded that it will not increase performance significantly. Comparing the increase in performance between super scalar und SCAD with and without branch prediction also shows this.

Figure 4.3 shows the performance for bubble-sort for an array of length 10. It can be seen that the performance increase in the super scalar case is quite significant, while it is almost non existant in the SCAD case.

```
   $16 -> pu0@in0
   DUPL,2 -> pu0@opc
   $0 -> pu1@in0
   DUPL,1 -> pu1@opc
   pu0@out0 -> pu0@in0
   DUPL,1 -> pu0@opc
   pu0@out0 -> pu1@in0
   pu1@out0 -> pu1@in1
   ==N,1 -> pu1@opc
   pu1@out0 -> cu@cond
   $0 -> cu@then
   pu0@out0 -> pu0@in0
   DUPL,1 -> pu0@opc
   $1 -> cu@jump
2: pu1@out0 -> pu0@in0
   DUPL,1 -> pu0@opc
1: pu0@out0 -> pu0@in0
   $1 -> pu0@in1
   -N,2 -> pu0@opc
   $0 -> pu1@in0
   DUPL,1 -> pu1@opc
   pu0@out0 -> pu1@in0
   DUPL,1 -> pu1@opc
   pu0@out0 -> pu0@in0
   pu1@out0 -> pu0@in1
   !=N,1 -> pu0@opc
   pu0@out0 -> cu@cond
   $2 -> cu@then
   pu1@out0 -> discard
   $3 -> cu@jump
0: pu0@out0 -> discard
3: $2147483647 -> cu@jump
```

```
   $16 -> pu0@in0
   DUPL,2 -> pu0@opc
0: pu0@out -> pu1@in0
   $0 -> pu1@in1
   ==N,1 -> pu1@opc
   pu1@out -> cu@cond
   $1 -> cu@then
   pu0@out -> pu0@in0
   $1 -> pu0@in1
   -N,2 -> pu0@opc
   $0 -> cu@jump
1: pu0@out -> discard
   $2147483647 -> cu@jump
```

Figure 4.2: *A simple loop, generated (left) and manually written (right)*

	no BP	with BP	speed up
super scalar	5,632	3,918	1.4375
SCAD	22,987	21,886	1.0503

Figure 4.3: *Performance of bubble-sort with and without branch prediction.*

5 Conclusion

We implemented a simulator framework to that allows us to analyze and compare the processor architectures – especially SCAD – on the level of detail that we need. We identified the core features and limiting parameters of each of the given processor architectures and compared them to the current state of the SCAD machine, gaining better understanding why it performs in the way that it does. Furthermore we explored the possibility of branch prediction and load store buffers for the SCAD architecture and concluded that these features won't be beneficial to SCAD. Instead it is necessary to explore code transformation techniques and better move code generation in order to make SCAD competitive. In the current state SCAD suffers from a lot of computational overhead and bad code generation. Future work will look at different compilation techniques e.g. generating move code from so called data process networks (DPNs) or similar.

Bibliography

[Bha20] A. Bhagyanath. “Code Generation for Synchronous Control Asynchronous Dataflow Architectures”. PhD. PhD thesis. Department of Computer Science, University of Kaiserslautern, Germany, Jan. 2020.

[BJS16] A. Bhagyanath, T. Jain, and K. Schneider. “Towards Code Generation for the Synchronous Control Asynchronous Dataflow (SCAD) Architectures”. In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*. Ed. by R. Wimmer. Freiburg, Germany: University of Freiburg, 2016, pp. 77–88.

[BS16] A. Bhagyanath and K. Schneider. “Optimal Compilation for Exposed Datapath Architectures with Buffered Processing Units by SAT Solvers”. In: *Formal Methods and Models for Codesign (MEMOCODE)*. Ed. by E. Leonard and K. Schneider. Kanpur, India: IEEE Computer Society, 2016, pp. 143–152.

[BS17] A. Bhagyanath and K. Schneider. “Exploring the Potential of Instruction-Level Parallelism of Exposed Datapath Architectures with Buffered Processing Units”. In: *Application of Concurrency to System Design (ACSD)*. Ed. by A. Legay and K. Schneider. Zaragoza, Spain: IEEE Computer Society, 2017, pp. 106–115.

[BSS15] N. Bhardwaj, M. Senftleben, and K. Schneider. “Abacus – A Processor Family for Education”. In: *Workshop on Embedded and Cyber-Physical Systems Education (WESE)*. Ed. by M. Törngren and M.E. Grimheden. New Delhi, India: ACM, 2015, 2:1–2:8.

[Bur+04] Doug Burger, Stephen W Keckler, Kathryn S McKinley, Mike Dahlin, Lizy K John, Calvin Lin, Charles R Moore, James Burrill, Robert G McDonald, and William Yoder. “Scaling to the End of Silicon with EDGE Architectures”. In: *Computer* 37.7 (2004), pp. 44–55.

[Cor99] Henk Corporaal. “TTAs: Missing the ILP complexity wall”. In: *Journal of Systems Architecture* 45.12-13 (1999), pp. 949–973.

[Fis83] Joseph A Fisher. “Very long instruction word architectures and the ELI-512”. In: *Proceedings of the 10th annual international symposium on Computer architecture*. 1983, pp. 140–150.

[Jai19] T. Jain. “Nonblocking On-Chip Interconnection Networks”. PhD. PhD thesis. Department of Computer Science, University of Kaiserslautern, Germany, July 2019.

[Lee+98] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. "Space-time scheduling of instruction-level parallelism on a raw machine". In: *ACM SIGOPS Operating Systems Review* 32.5 (1998), pp. 46–57.

[Sch18] A. Schneiders. "Using Static-Single-Information-Form for SCAD Code Generation". Bachelor. MA thesis. Department of Computer Science, University of Kaiserslautern, Germany, June 2018.

[Swa+07] Steven Swanson, Andrew Schwerin, Martha Mercaldi, Andrew Petersen, Andrew Putnam, Ken Michelson, Mark Oskin, and Susan J Eggers. "The wavescalar architecture". In: *ACM Transactions on Computer Systems (TOCS)* 25.2 (2007), pp. 1–54.

[Thu+09] Martin Thuresson, Magnus Själander, Magnus Björk, Lars Svensson, Per Larsson-Edefors, and Per Stenstrom. "FlexCore: Utilizing exposed datapath control for efficient computing". In: *Journal of Signal Processing Systems* 57.1 (2009), pp. 5–19.

[Tom67] R. M. Tomasulo. "An Efficient Algorithm for Exploiting Multiple Arithmetic Units". In: *IBM Journal of Research and Development* 11.1 (1967), pp. 25–33. DOI: 10.1147/rd.111.0025.

[Wat16] Andrew Shell Waterman. *Design of the RISC-V instruction set architecture*. University of California, Berkeley, 2016.