School of Electrical, Electronic & Computer Engineering

Automated Generation of Processor Architectures in Embedded Systems Design

Maxim Rykunov, Andrey Mokhov, Alex Yakovlev, Albert Koelmans

Technical Report Series

NCL-EECE-MSD-TR-2010-164

November 2010

Contact:

maxim.rykunov@ncl.ac.uk

andrey.mokhov@ncl.ak.uk

alex.yakovlev@ncl.ac.uk

albert.koelmans@ncl.ac.uk

Supported by EPSRC grants EP/C512812/1 and EP/G037809/1.

NCL-EECE-MSD-TR-2010-164

School of Electrical, Electronic & Computer Engineering,
Merz Court,
University of Newcastle upon Tyne,
Newcastle upon Tyne, NE1 7RU, UK

`http://async.org.uk/`

Maxim Rykunov, Andrey Mokhov, Alex Yakovlev, Albert Koelmans
Microelectronics System Design Group, Newcastle University, UK

Abstract

Automated design of processor architectures has traditionally been focused on the clocked pipeline organisation consisting of a fairly standard datapath and control logic. As the area of processor design automation is becoming increasingly inclusive of system paradigms that are heterogeneous in terms of timing, such as multiclock and asynchronous circuits, there is a need for appropriate models and associated synthesis algorithms.

This paper approaches the problem of designing control logic for a processor using Conditional Partial Order Graphs (CPOGs). The new method allows composing a large set of microarchitectural algorithms (instructions) into a compact relational form, which opens way for various transformation and optimisation procedures leading to an efficient implementation of control logic. In this paper we present a CPOG-based design methodology, demonstrate its application to synthesis of control logic and datapath for an asynchronous microprocessor. Our approach is compared with synchronous and asynchronous BALSA-based designs it terms of power and performance using Altera FPGA platform.

1 Introduction

Nowadays microprocessor designers face new challenges associated with the rapidly changing requirements for power consumption, high performance, area efficiency, etc., as well as demand for shorter time to market and greater productivity. To date, there exist approaches and associated EDA (Electronic Design Automation) support for automated generation of Application Specific Instruction set Processor (ASIP) [1, 2]. They show a general processor design environment based on micro-architectural description of instruction algorithms. For example, the PEAS-III system [3] allows the generation of a complete tool-suite and includes a compiler, assembler, linker and simulator; the MetaCore system [4] is a benchmark driven ASIP development system, based on a formal representation language (see also the Xtensa system [5], the EPICS system [6], etc.). While these methodologies have reached significant levels in automating the whole design process, they are limited to design of synchronous systems. However, modern processors are increasingly diversified in timing modes, which calls for new ways of handling concurrent and asynchronous interactions.

In principle, one could apply a recently developed desynchronisation method, where the synchronous design is converted automatically into a asynchronous one [7, 8, 9]. Desynchronisation has the advantage of minimal changes in the conventional design flow, and may be an appropriate way towards design reuse, i.e. obtaining new asynchronous implementations from existing synchronous designs. In the event of a rapidly evolving processor specification, a more "direct" way of incorporating new instructions and their microarchitectural algorithms will be needed.

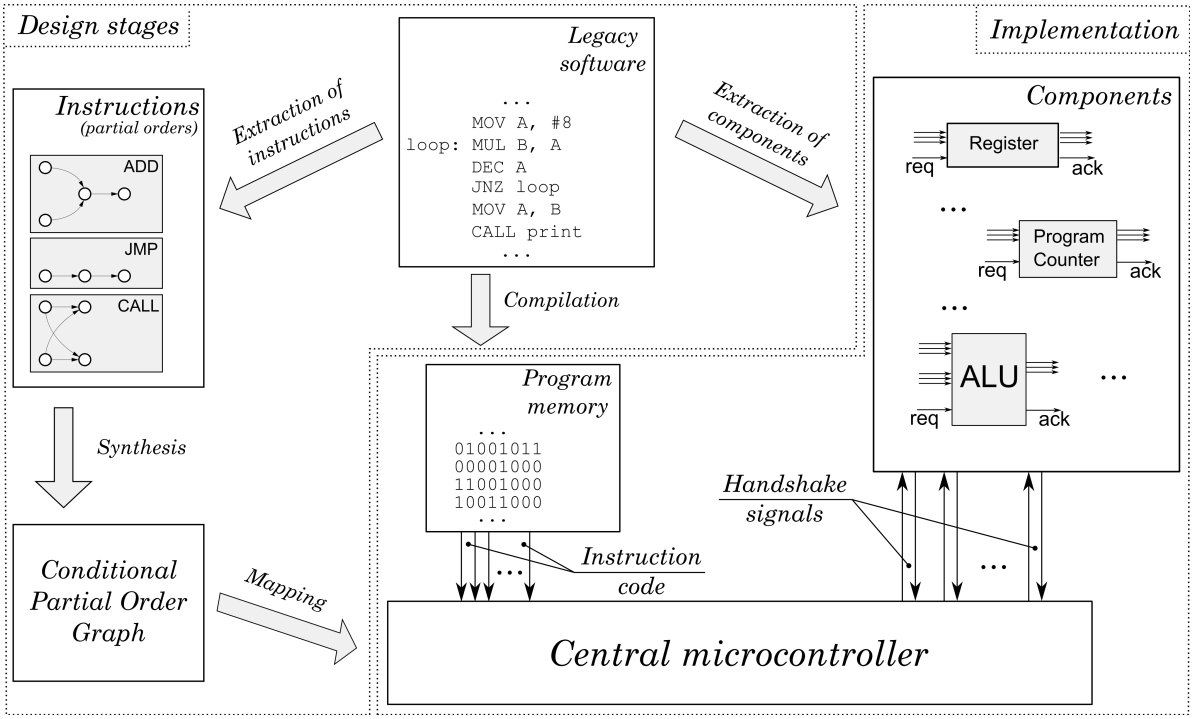


Figure 1: Conceptual view of the design process

There are direct methods in asynchronous design and they have been applied to CPUs. However they are known to lead to inefficient implementations, especially in terms of area and performance [10, 11, 12]. This is caused by the fact that multiple algorithmic procedures for a large set of instructions require explicit enumeration of all alternatives. Syntax-directed methods instantiate every branch as a hardware control thread, thereby leading to large overheads.

The key part in a microprocessor control specification is the description of instructions. Each instruction corresponds to a schedule of primitive actions such as data transfer, arithmetic operation, memory access, etc., which are performed by operational units (datapath components). The design of instruction sets for a particular combination of operational units and software requirements is a difficult task [2]. Moreover, it is worth mentioning that there is a significant difference between designing CPU control and other types of hardware. Namely, microprocessor control contains many algorithms, which are associated with a variety of instructions rather then a single algorithm, which is commonly present in random logic design.

The goal of this paper is to show that the CPOG-based design methodology can help to significantly improve the design space exploration. Moreover our flow has the ability for a quick and easy retargeting of the synthesised processor and control logic. We also want to demonstrate the high potential of our approach in terms of performance and power consumption characteristics in comparison to existing synchronous and asynchronous design methodologies. Comparisons were done by running the same benchmark program on processors synthesised using different approaches.

The main contribution of this work is a new design flow for automated synthesis of CPU control as a composition of scheduled algorithms of CPU instructions (see Figure 1). It is particularly important to consider this flow in the context of diversifying requirements, such as high performance, low power consumption, high security, etc. This flow is based on a new model: Conditional Partial Order Graphs (CPOG). CPOGs facilitate systematic transformation of a superposition of microarchitectural control algorithms into a netlist for the central control logic of a CPU.

The rest of this paper is organised as follows. In Section 2 we discuss the motivation for a model to describe the microarchitectural level of CPU design and other goals in achieving an effective design flow for synthesis of control circuits for microprocessors. Section 3 provides a quick overview of a new

model for system specification and synthesis – CPOG. Section 4 outlines the whole proposed design flow. Section 5 illustrates the application of the flow to the synthesis of control for Intel's 8051. In Section 6 we discuss our approach in terms of power and performance and compare it with synchronous and asynchronous BALSA-based designs. Finally we summarise the contribution of this paper and discuss the future work in Section 7.

2 Motivation for a new model

As processors become more diverse in terms of timing, the conventional FSM-oriented approach to designing their control logic starts to hit fundamental limitations. An FSM captures behaviour of a system using an explicit set of *states* and *transitions* between these states. There is a direct correspondence between the states of an FSM and those of the modelled system, which leads to problems in terms of concurrency specification. For example, to specify n concurrent events, the designer has to explicitly specify 2^n intermediate states, each of them corresponding to a state, where a subset of these n events have already happened. There are 2^n different subsets of a set of n events, hence the number of the required states [13] [14]. This brings us to the first requirement for a new model: it should be able to capture concurrency without explicit enumeration of all possible event interleavings.

Models like Petri Nets (PNs) [15] and Signal Transition Graphs (STGs) [16] are able to capture concurrency and choice at a very fine level, and produce more compact and faster control circuits than the methods based on syntax-directed translation from HDLs. However, they are built on the explicit listing of all the event traces and causal relations of a system, what limits their applicability to microcontrollers with a rather small state space. Processors may contain hundreds of encoded instructions, each a microprogram operating on a shared set of datapath components. It is practically impossible to specify such systems using PNs or STGs, as demonstrated in [13].

Another requirement for a new model is *expressiveness*: it should be expressive enough to cover a wide range of solutions for different optimisation criteria, thus enabling a prompt retargeting of the synthesised processor for varying and often conflicting optimisation goals. For example, we can retarget a CPU architecture in order to:

- minimise power consumption – this can be achieved by reducing the width of buses between datapath components and/or by using low-power components; this inevitably affects the performance of the system and necessitates rescheduling of the execution of the instructions.
- maximise performance – this can be achieved by increasing the number of functional units, their bit-width and therefore their performance; the instructions are to be rescheduled in a maximally concurrent way.
- optimise the architecture for a particular software application – this may involve complete re-encoding of the instructions, taking into account their statistical properties.
- etc.

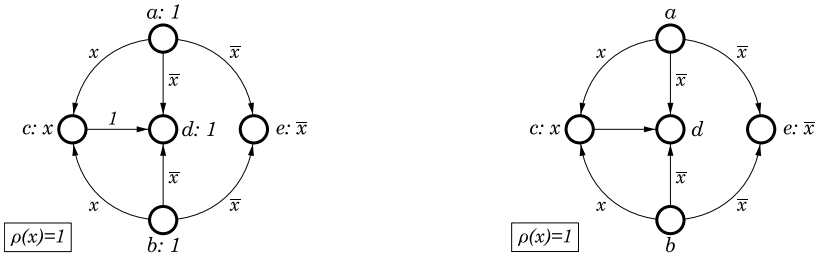
The Conditional Partial Order Graph model introduced recently [14, 17] satisfies all the requirements and provides a formal framework for specification, verification and synthesis of processor microarchitectures. In this paper we present a processor design methodology built around this model. The next section briefly introduces the CPOG model.

3 CPOG essentials

Conditional Partial Order Graph [13, 14] is a quintuple $H(V,\ E,\ X,\ \rho,\ \phi)$, where V is a finite set of *vertices*, $E \subseteq V \times V$ is a set of *arcs* between them, and X is a finite set of *operational variables*. An

opcode is an assignment $(x_1,\ x_2,\ \ldots,\ x_{|X|}) \in \{0,\ 1\}^{|X|}$ of these variables; X can be assigned only those opcodes which satisfy the *restriction function* ρ of the graph, i.e. $\rho(x_1,\ x_2,\ \ldots,\ x_{|X|}) = 1$. Function ϕ assigns a Boolean *condition* $\phi(z)$ to every vertex and arc $z \in V \cup E$ of the graph.

In the context of processor microarchitecture we interpret vertices V as data path components whose order of execution is determined by an instruction code provided by operational variables in X. The order is captured in the precedence relation E which is conditional (every vertex or arc $z \in V \cup E$ has an associated condition $\phi(z)$). The restriction function ρ effectively lists all the allowed instruction codes.



(a) Full notation (b) Simplified notation

Figure 2: Graphical representation of CPOGs

Figure 2(a) shows an example of a CPOG containing $|V| = 5$ vertices and $|E| = 7$ arcs. There is a single operational variable x; the restriction function is $\rho(x) = 1$, hence both opcodes $x = 0$ and $x = 1$ are allowed. Vertices $\{a,\ b,\ d\}$ have constant $\phi = 1$ conditions and are called *unconditional*, while vertices $\{c,\ e\}$ are *conditional* and have conditions $\phi(c) = x$ and $\phi(e) = \overline{x}$ respectively. Arcs also fall into two classes: *unconditional* (arc $(c,\ d)$) and *conditional* (all the rest). As CPOGs tend to have many unconditional vertices and arcs we use a simplified notation in which conditions equal to 1 are not depicted in the graph. This is demonstrated in Figure 2(b).

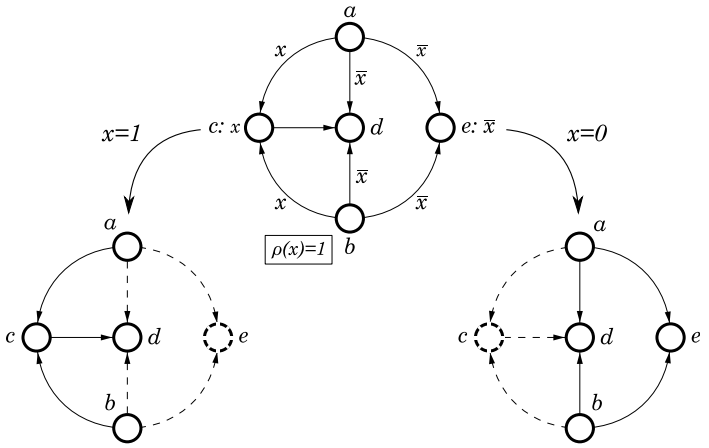


Figure 3: Multiple CPOG projections

The purpose of conditions ϕ is to 'switch off' some vertices and/or arcs in the graph according to the given opcode. This makes CPOGs capable of specifying multiple partial orders or instructions (a partial order is a form of behavioural description of an instruction). Figure 3 shows a graph and its two *projections*. The leftmost projection is obtained by keeping only those vertices and arcs whose conditions evaluate to 1 after substitution of the operational variable x with 1. Hence, vertex e disappears, because its condition evaluates to 0: $\phi(e) = \overline{x} = \overline{1} = 0$. Arcs $\{(a,\ d),\ (a,\ e),\ (b,\ d),\ (b,\ e)\}$ disappear for the same reason. The rightmost projection is obtained in the same way, with the only difference that variable x is set to 0. Note also that although the condition of arc $(c,\ d)$ evaluates to 1 (in fact it is constant 1), the arc is still excluded from the resultant graph because one of the vertices it connects (vertex c) is excluded, and obviously an arc cannot appear in a graph without one of its vertices. Each of the obtained

projections can be treated as a specification of a particular behavioural scenario of the modelled system. Potentially, a CPOG $H(V,\ E,\ X,\ \rho,\ \phi)$ can specify an exponential number of different partial orders of events in V according to one of $2^{|X|}$ different possible opcodes.

To conclude, a CPOG is a structure to represent a set of encoded partial orders in a compact form. The synthesis and optimisation methods presented in [14, 18] provide a way to obtain such a representation given a set of partial orders and their opcodes. For example, the CPOG in Figure 3 can be synthesised automatically from the two partial orders below it and the corresponding opcodes $x = 1$ and $x = 0$.

4 Proposed design flow

Figure 1 shows an overview of the process that can be used to generate control logic of a CPU.

Specification of such a complex system as a processor usually starts at the architectural level [19]. This helps designers to deal with the system complexity by structural abstraction: the whole system is divided into several subsystems in a such way that each of them can be designed individually, thus significantly simplifying the whole design flow and reducing the solution search space. As stated in the introduction, the next particularly important step is refining the architecture to the level of scenarios or instructions, each of which corresponds to a schedule of primitive actions. In this aspect, the whole design flow can be separated into 3 subsections: system specification, synthesis and physical implementation.

4.1 System specification

First of all, we need to define the set of instructions (behavioural scenarios) and datapath components of the microprocessor. There are several ways to do this:

- Extraction from legacy software

Components and instructions can be extracted from some legacy software (could be written in C/C++, Assembly language, etc.).

- Using Architecture Description Languages (ADLs)

The ADL specification is used to help in performing different design automation tasks, e.g. hardware generation and functional verification of processors. There are several categories of ADLs, based on the nature of the available information, such as: structural ADLs (e.g. MIMOLA [20], UDL/I [21]), behavioural ADLs (e.g. nML [22], ISDL [23]), mixed ADLs (e.g. LISA [24], EXPRESSION [25]) and partial ADLs (e.g. AIDL [26]).

- Other instruction specifications

There can be some other sources, such as a list of instructions obtained from a microprocessor specification (e.g. a CPU manual). This type of specification is not formal and therefore should be processed manually.

In order to distinguish scenarios, they need to be encoded with Boolean vectors. These encodings can be assigned arbitrarily or can be provided as part of system specification. Importantly, the size and latency of the final microcontroller circuit depends significantly on the chosen encoding of the scenarios. Section 5 presents an example of a binary encoding scheme for a given set of scenarios. However, other types of encoding can also be used, such as one hot, matrix, balanced and others [17, 13].

4.2 Synthesis

Once the behavioural scenarios and datapath components of the system are specified and encoded, it is possible to synthesise a CPOG which contains all of them by using algebraic techniques described in [13, 14]. At this stage the obtained CPOG can be viewed as an intermediate model for a compact representation of the system behaviour, which will be further mapped onto a gate-level circuit. Therefore

it is not necessary for the designer to be familiar with the CPOG-based theory in order to use the proposed design flow.

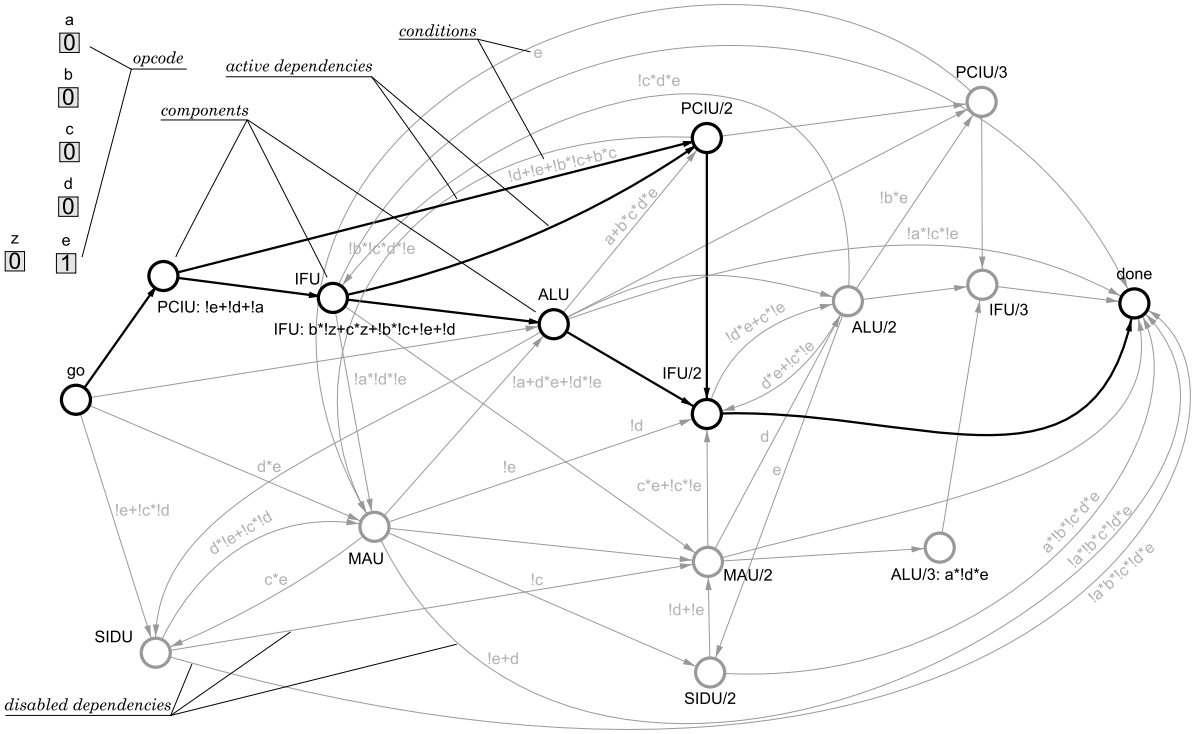


Figure 4: Synthesised CPOG

It is important to mention the flexibility of the CPOG-based methodology, in the sense that CPOG synthesis is possible not only from partial orders, it can also be a mixture of partial orders and CPOGs. This allows a complete reuse of the existing CPOG specifications without their decomposition into separate partial orders, so that one can add a new instruction into the specification of a system and avoid its complete resynthesis.

There are several possible optimisation techniques at this stage, which can reduce the size of a given CPOG using logic minimisation and by exploiting the structural graph properties. It is particularly important that the size and latency of the obtained microcontroller strongly correlates with the complexity of the original graph, therefore it is crucial to optimise the CPOG specification as far as possible by the application of CPOG optimisation techniques [13, 18].

4.3 Physical implementation

The final stage of the our automated generation of control logic is mapping the CPOG representation into a set of logic gates. As soon as the CPOG specification of a system is synthesised it can be mapped onto Boolean equations in order to produce a physical implementation (gate-level netlist) of the specified microcontroller. The size of the resulting Boolean equations is linear in the size of the CPOG. The details of translating of CPOGs into Boolean equations can be found in [13, 14]. Finally, we can translate them to VHDL, Verilog or other HDL (Hardware Description Language) and/or input these equations into the technology mapping and Place and Route (P&R) tools. See the example in the next Section.

Along with hardware mapping we have to perform software mapping, i.e. the compilation of the program code from a given legacy software. The compilation result is stored in the program memory.

Our design process, defined in Figure 1, shows that the interface between control and operation components is based on a handshake protocol. This allows significant flexibility in developing or reusing the datapath of the controller. Due to the handshake protocol, the full power of partial orders can be exploited, because the timing of control events is not bounded to particular delay constraints. The

advantages of such an approach have been recently applied to the designs in [27, 28, 29]. All information about the number and type of the CPU components, such as registers, program counter, ALU (Arithmetic logic unit), etc., can also be extracted from the legacy software and later on added to the design.

In the next section we present an example of control logic synthesis.

5 Example of control logic synthesis

Regarding our design flow, first of all we need to specify the instructions of the microprocessor. We chose the instruction set from the core of the Intel 8051 microcontroller, as it is a well-known, popular CPU and still used in many applications even nowadays.

Extractions of components. Functional components, such as PCIU, ALU, IFU, etc. can be extracted from the given instruction set (see Figure 4), moreover each functional block can be use several times in one partial order, what is shown as PCIU/2, PCIU/3, IFU/2 etc.

Components	Description
PCIU	Program Counter Increment Unit
IFU	Instruction Fetch Unit
ALU	Arithmetic Logic Unit
MAU	Memory Access Unit
SIDU	Stack pointer Increment Decrement Unit

Extractions of partial order for instructions. The Intel 8051 microcontroller's instruction set contains 110 instructions. Many of them have the same partial order representation, so we group them into 20 classes. For instance, one of the classes corresponds to a group of ALU operations (namely, *ALU op. #data to Rn*). In this set of instructions one of the operands is a register and the other is an immediate constant (e.g. MOV A, #1 – move 1 to accumulator, ANL A, #1 – bitwise "AND" operation between accumulator and 1). Figure 5 shows the partial order of the actions for such an instruction. First, the constant has to be fetched into the Instruction Register (IR) – execution of components PCIU and IFU. Then ALU is executed concurrently with an increment of Program Counter (PC). Finally, it is possible to fetch the next instruction into IR.

Other instructions are described in the same way.

Encoding of partial orders. In order to distinguish between the synthesised partial orders, we need to encode them. As stated in the proposed flow, the chosen type of encoding has significant influence on the size and complexity of their CPOG composition. We used a binary encoding scheme. As we grouped all the instructions into 20 classes, we need at least 5 bits to encode them, i.e. assign opcodes {00000, 00001, ...10011}. Figure 5 shows that opcode 00000 was given to our example class. This encoding scheme has been chosen for the sake of simplicity; it might not be optimal in terms of area, performance or latency of the final microcontroller circuit. The decision as to which encoding has to be used in the final design is governed by the overall opimisation criteria, as pointed out in Section 2.

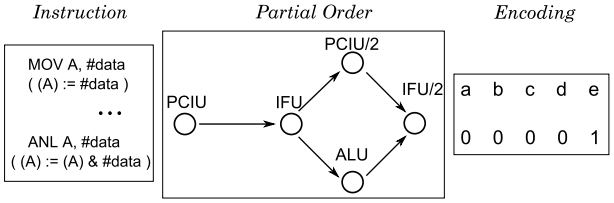


Figure 5: ALU op. #data to Rn

CPOG generation. Now, we have specified and encoded all the partial orders of the microcontroller. We can proceed to synthesis of a CPOG containing all of them. Figure 4 shows the obtained CPOG, which contains 15 vertexes and 46 arcs between them and with 5 opcode variables we obtained 225 literals. This result shows that it is a compact implementation and it would be impossible to describe this control circuit in such a compact form by using other techniques, such as FSMs. Most of the vertices and arcs have conditions, depending on which a pacticular partial order can be active or disabled. Our example partial order from Figure 5 is highlighted in Figure 4. Its elements are activated in accordance with the evaluation of the conditions on the operational code 00000.

Mapping. We can translate the obtained CPOG into Boolean equations using algebraic techniques [13, 18]. Then these equations are imported into Altera's Quartus II tool for technology mapping into FPGA. The mapping result as a Register Transfer Level (RTL) netlist is shown in Figure 6. As a result our control circuit placed on the chip EP1S10F780I6 from the Altera Stratix family contains 106 logic elements, which is about 1% of the total chip area.

The synthesised control circuit was simulated using the same EDA tool. The results confirmed the correct implementation of the handshake protocol between the controller and data path components, and the expected orders of component activation according to the instructions.

The future work intends to validate the new CPU design flow by implementing both a control and datapath for the 8051 microprocessor in hardware, first as FPGA and as ASIC later on. This will include experiments with different types of operational units (e.g. bundled data, dual-rail), and different encodings to best fit the given program benchmarks and hardware requirements.

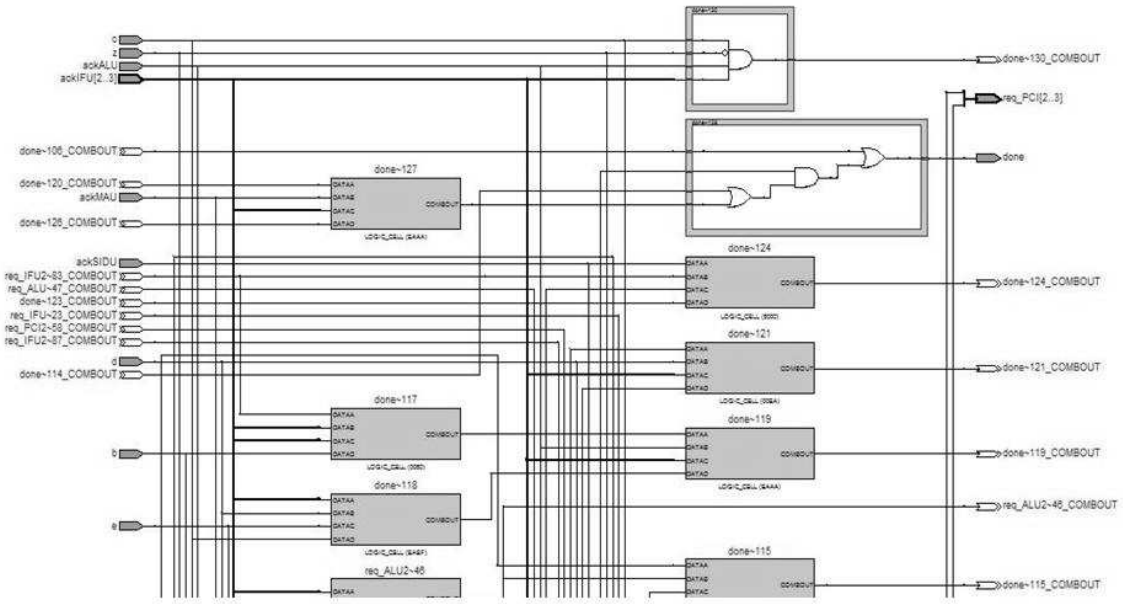


Figure 6: Synthesis of circuit implementation

6 Benchmark and Comparison results

To compare the presented methodology with existing approaches we decided to specify and synthesise a simplified processor with a minimal instruction set capable of executing the following simple program:

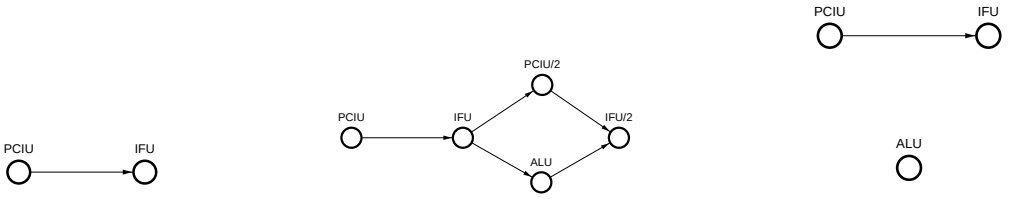
Algorithm 1 Benchmark program

```
Lab1:  MOV A, #FF
Lab2:  DEC A
       JNZ Lab2
       JMP Lab1
```

The processor should contain 3 operational units (PCIU, IFU, ALU) and should support execution of 5 instructions (NOP, MOV, DEC, JNZ, JMP). The units perform the following primitive actions: PCIU increments the program counter (PC), IFU fetches the next instruction or operand into instruction register (IR), and ALU can copy data between registers and subtract one register from another. The instructions execute the following sequences of these primitive actions:

- NOP (*No operation*) instruction does nothing except incrementing the program counter (PCIU) and fetching the next instruction (IFU) as shown in Figure 7(a).
- MOV A, #FF (*loads constant value #data into accumulator A*). The constant is given immediately after the instruction opcode (so called *immediate addressing mode*). Figure 7(b) shows the partial order of actions for this operation. At first, the constant has to be fetched into the IR (actions PCIU and IFU). Then an ALU operation is performed (copying the constant to the accumulator) concurrently with another increment of the PC. Finally, it is possible to fetch the next instruction opcode into the IR.
- DEC decrements the accumulator concurrently with the next instruction fetch, as shown in Figure 7(c).
- JNZ #123 is a conditional branching operation (*Jump to #123 if Not Zero*): if the accumulator contains zero the program continues with the next instruction, otherwise the specified address is copied to the PC (similarly to MOV) – see Figure 7(d). At first, PC is incremented so that it points to the constant (the branch address). Then if the accumulator is zero (signified with variable z) the PC is incremented again and the next instruction is fetched (the upper branch of the conditional partial order). If the accumulator is not zero then the address is fetched into the IR and copied into the PC (the lower branch IFU$\rightarrow$ALU), followed by fetching the next instruction.
- JMP #123 performs an unconditional jump operation (*Jump to #123*): the specified address, where the program should continue its execution, is given immediately after the instruction opcode. Figure 7(e) shows the partial order of actions for this operation. At first, the address has to be fetched into the IR (actions PCIU and IFU). Then the ALU operation is performed – the address is copied into the PC, followed by fetching the next instruction from this address.

The five instructions can be overlaid into the single CPOG shown in Figure 8. The instructions are given sequential 3-bit opcodes 000, 001, 010, 011, 100 for simplicity.



(a) Partial order representation of NOP instruction

(b) Partial order representation of MOV instruction

(c) Partial order representation of DEC instruction

(d) Partial order representation of JNZ instruction

(e) Partial order representation of JMP instruction

Figure 7: Graphical representation of partial orders

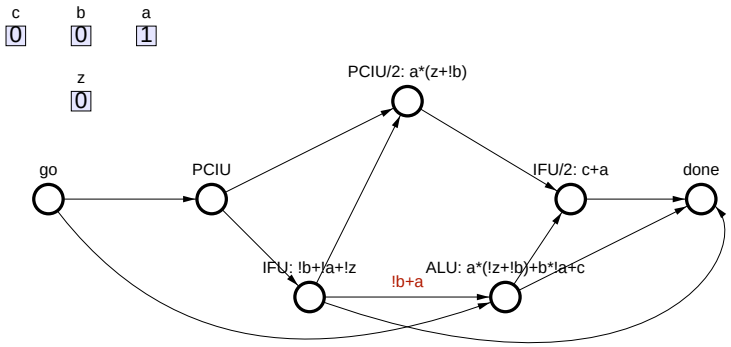


Figure 8: Graphical representation of CPOG

The block diagram, see Figure 9, represents the whole internal structure of our microprocessor: the control logic, which was synthesised with CPOG-based methodology and the datapath to support the instruction execution, which contains PCIU, IFU, ALU units and IR (the accumulator register is considered to be a part of the ALU). As the whole approach is asynchronous, all these components communicate using request and acknowledgement signals. Furthermore there are Data_In and Data_Out signals, which represent input and output data for the ALU.

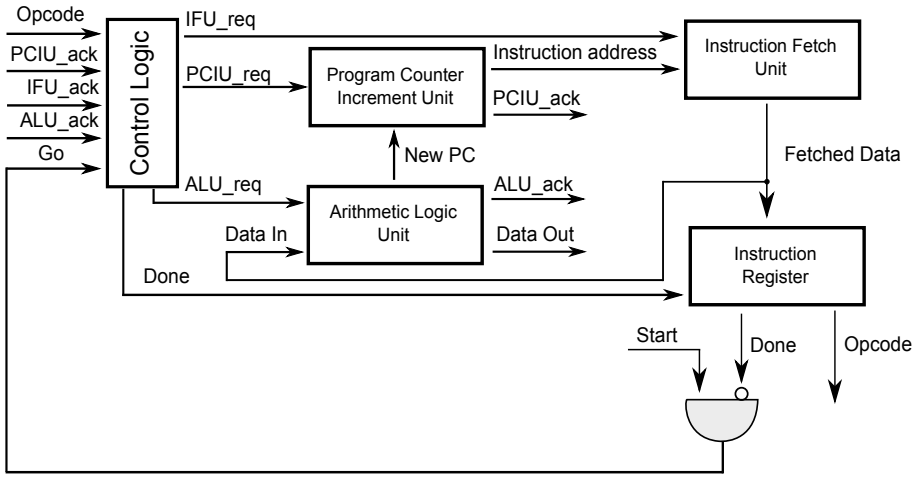


Figure 9: CPU block diagram

We simulated our design with the QuartusII tool and then placed and tested it on two different Altera FPGA platforms: Flex10K – EPF10K70RC240 and CycloneIII – EP3C16F484C6. Our approach was compared with a synchronous implementation of the same basic processor and an asynchronous Balsa implementation.

The synchronous implementation was synthesised using the QuartusII tool from VHDL code (see Appendix for full listing).

The asynchronous implementation was obtained using the Balsa language, and then compiled into a handshake circuit with the Balsa compiler. Each handshake component has a gate level implementation, thus Balsa can automatically generate them into a Verilog netlist for Xilinx or Altera synthesis tools. In addition, to reduce the area cost, four-phase bundle-data protocol was chosen instead of a dual-rail protocol.

In order to compare our design with the listed implementations, the same benchmark program (Algorithm 1) was executed on each of the designs on different FPGA boards. The performances and power consumptions are summarised in Table 1.

Type of processor	FPGA platform	Performance MIPS[1]	MIPS/W	Total power mW	Average Power Dynamic/Static mW	Area (logic elements)/% of total available on chip
Our custom CPU implementation	Flex10K	18	-	-	-	185/5%
	Cyclone III	31	405	74	2.5/52	198/1%
Synchronous implementation	Flex10K 18 MHz	8	-	-	-	144/4%
	Cyclone III 100 MHz	21	290	73	1.5/54	105/<1%
Asynchronous Balsa implementation	Flex10K	13	-	-	-	1548/42%
	Cyclone III	20	190	118	6.2/111	1510/10%

[1] MIPS - Million Instructions Per Second

Table 1: Performance and power consumption comparison with other implementations

From the results we can see that our approach has the best performance among all of the implementations: more than 38% (Flex10K) and 55% (Cyclone III) compared to Balsa implementation; regarding the synchronous approach – 125% (Flex10K) and 47% (Cyclone III). However power consumption and

the total amount of logic elements in synchronous design is better. The reason for this is that the data path in our implementation was designed manually and can be optimised futher, however the circuit implementation of the synchronous one was synthesised from VHDL code.

No power consumption was measured for the Flex10K FPGA platform, because Quartus II does not support such measurements for this board, but we can still compare the performance in terms of MIPS and area in terms of logic elements needed.

The maximum frequencies which can be obtained from the synchronous approach are 18 MHz for Flex10K and 100 MHz for Cyclone III.

The Balsa design shows also a high performance rate, however the total amount of needed logic elements and therefore power consumption is higher. This can happen because, as opposed to CPOG-based methodology, the Balsa tool control of instruction can be written as a "case statement":

```
case (IR as SSEMInst).Func of
JMP then MemoryREAD (); ZeroPC (); AddMDRToPC()
| LDN then MemoryREAD (); ACC_slave:=(MDR as word)
| SUB then ACC_slave:=(ACC - 1 as word)
| TEST then
if #ACC [31]
then IncrementPC () end
end;
```

See Appendix for complete code of Balsa implementation.

7 Conclusion and Future Work

We have presented a new design flow for specification and synthesis of processor control logic. We used a new formal model for the description of instruction sets, to deal with concurrency and prompt retargeting of the CPU.

The presented example of control logic synthesis shows the capability of proposed design flow and the potential benefits of CPOGs-based methodology.

Our future work will focus on implementation of complete implementation of asynchronous CPU Intel 8051, optimisation techniques for the current flow, such as implementing other types of instructions encodings and further development of self-timed pipelining within the represented methodology.

Acknowledgement

This work was supported by EPSRC grants EP/C512812/1 and EP/G037809/1.

References

[1] Mishra and Kejariwal. Rapid exploration of pipelined processors through automatic generation of synthesizable rtl models. In *RSP '03: Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping (RSP'03)*, page 226, Washington, DC, USA, 2003. IEEE Computer Society.

[2] Scheffer and Grant Martin. *EDA for IC System Design, Verification, and Testing (Electronic Design Automation for Integrated Circuits Handbook)*. CRC Press, Inc., Boca Raton, FL, USA, 2006.

[3] Makiko Itoht and Shigeaki Higakit. Peas-iii: An asip design environment. In *ICCD '00: Proceedings of the 2000 IEEE International Conference on Computer Design*, page 430, Washington, DC, USA, 2000. IEEE Computer Society.

[4] Jin-Hyuk Yang and Chong-Min Kim. Metacore: an application specific dsp development system. In *DAC '98: Proceedings of the 35th annual Design Automation Conference*, pages 800–803, New York, NY, USA, 1998. ACM.

[5] Ricardo E. Gonzalez. Xtensa: A configurable and extensible processor. *IEEE Micro*, 20(2):60–70, 2000.

[6] R. Woudsma and R. A. M. Beltman. Epics, a flexible approach to embedded dsp cores. *Int'l Conference on Signal Processing Applications & Technology (ICSPAT)*, 651:506–511, Oct. 1994.

[7] Nikolaos Andrikos, Luciano Lavagno, Davide Pandini, and Christos P. Sotiriou. A fully-automated desynchronization flow for synchronous circuits. In *DAC '07: Proceedings of the 44th annual Design Automation Conference*, pages 982–985, New York, NY, USA, 2007. ACM.

[8] J. Cortadella, A. Kondratyev, L. Lavagno, K. Lwin, and C. Sotiriou. From synchronous to asynchronous: An automatic approach. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 21368, Washington, DC, USA, 2004. IEEE Computer Society.

[9] Jordi Cortadella, Alex Kondratyev, Senior Member, Luciano Lavagno, and Christos P. Sotiriou. Desynchronization: Synthesis of asynchronous circuits from synchronous specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25:2006, 1994.

[10] K. L. Chang and B. H. Gwee. A low-energy low-voltage asynchronous 8051 microcontroller cores. *Proc. ISCAS*, page 4, 2006.

[11] Hans van Gageldonk, Kees van Berkel, Ad Peeters, Daniel Baumann, Daniel Gloor, and Gerhard Stegmann. An asynchronous low-power 80c51 microcontroller. *Asynchronous Circuits and Systems, International Symposium on*, 0:0096, 1998.

[12] L. A. Plana and Riocreux. Spa–a secure amulet core for smartcard applications. *Microprocessors and Microsystems*, 27:15, 2003/10.

[13] Andrey Mokhov. *Conditional Partial Order Graphs.* PhD thesis, School of EECE, Newcastle University, 2009.

[14] Andrey Mokhov and Alexandre (Alex) Yakovlev. Conditional partial order graphs: Model, synthesis, and application. *IEEE Transactions on Computers*, 59:1480–1493, 2010.

[15] Tadao Murata. Petri Nets: Properties, Analysis and Applications. 77(4), 1989.

[16] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. *Logic synthesis of asynchronous controllers and interfaces.* Springer-Verlag, ISBN: 3-540-43152-7, 2002. InternalNote: submitted by: hr.

[17] Andrey Mokhov and Alex Yakovlev. Conditional partial order graphs and dynamically reconfigurable control synthesis. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 1142–1147, New York, NY, USA, 2008. ACM.

[18] Alex Yakovlev Andrey Mokhov, Arseniy Alekseyev. Automated synthesis of instruction codes in the context of micro-architecture design. In *In Proc. of 10th Int. Conf. on Applicatioon of Concurrency to System Design (ACSD 2010)*, 2010.

[19] J. Silc and B. Robic. A survey of new research directions in microprocessors. *Microprocessors and Microsystems*, pages 175–190, 2000.

[20] G. Zimmermann. The mimola design system: a computer aided digital processor design method. In *25 years of DAC: Papers on Twenty-five years of electronic design automation*, pages 525–530, New York, NY, USA, 1988. ACM.

[21] Tamio Hoshino. Udl/i version two: A new horizon of hdl standards. In *CHDL '93: Proceedings of the 11th IFIP WG10.2 International Conference sponsored by IFIP WG10.2 and in cooperation with IEEE COMPSOC on Computer Hardware Description Languages and their Applications*, pages 437–452, Amsterdam, The Netherlands, The Netherlands, 1993. North-Holland Publishing Co.

[22] A. Fauth and M. Freericks. Describing instruction set processors using nml. In *EDTC '95: Proceedings of the 1995 European conference on Design and Test*, page 503, Washington, DC, USA, 1995. IEEE Computer Society.

[23] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. Isdl: an instruction set description language for retargetability. In *DAC '97: Proceedings of the 34th annual Design Automation Conference*, pages 299–302, New York, NY, USA, 1997. ACM.

[24] Andreas Hoffmann, Oliver Schliebusch, Achim Nohl, Gunnar Braun, Oliver Wahlen, and Heinrich Meyr. A methodology for the design of application specific instruction set processors (asip) using the machine description language lisa. In *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 625–630, Piscataway, NJ, USA, 2001. IEEE Press.

[25] Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt, and Alex Nicolau. Expression: a language for architecture exploration through compiler/simulator retargetability. In *DATE '99: Proceedings of the conference on Design, automation and test in Europe*, page 100, New York, NY, USA, 1999. ACM.

[26] H. Morimoto and K. Yamazaki. Superscalar processor design with hardware description language aidl. In *2nd Asia Pacific Conference on Hardware Description*, volume 75, pages 51–58, Oct. 1994.

[27] Basit Riaz Sheikh and Rajit Manohar. An operand-optimized asynchronous ieee-754 double-precision floating-point adder. In *IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2010.

[28] V. Varshavsky (Ed.). Self-timed control of concurrent processes. *Kluver Academic Publishers*, 1990.

[29] Bah-Hwee Gwee; Chang; Yiqiong Shi; Chien-Chung Chua; Kwen-Siong Chong;. A low-voltage micropower asynchronous multiplier with shift add multiplication approach. *Circuits and Systems I: Regular Papers, IEEE Transactionson*, 56 Issue:7:1349 – 1359, July 2009.

Appendix

The synchronous implementation:

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
LIBRARY lpm;
USE lpm.lpm_components.ALL;
ENTITY SCOMP IS
PORT( clock, reset : IN STD_LOGIC;
program_counter_out : OUT STD_LOGIC_VECTOR( 7 DOWNTO 0 );
register_AC_out : OUT STD_LOGIC_VECTOR(15 DOWNTO 0 );
memory_data_register_out : OUT STD_LOGIC_VECTOR(15 DOWNTO 0 ));
END SCOMP;
ARCHITECTURE a OF scomp IS
TYPE STATE_TYPE IS ( reset_pc, fetch, decode, execute_dec, execute_load, execute_nop, execute_jnz,
execute_jump );
SIGNAL state: STATE_TYPE;
SIGNAL instruction_register, memory_data_register : STD_LOGIC_VECTOR(15 DOWNTO 0 );
```

```
SIGNAL register_AC : STD_LOGIC_VECTOR(15 DOWNTO 0 );
SIGNAL program_counter : STD_LOGIC_VECTOR( 7 DOWNTO 0 );
SIGNAL memory_address_register : STD_LOGIC_VECTOR( 7 DOWNTO 0 );
SIGNAL memory_write : STD_LOGIC;
BEGIN
-- Use LPM function for computer's memory (256 16-bit words)
memory: lpm_ram_dq
GENERIC MAP (
lpm_widthad => 8,
lpm_outdata => "UNREGISTERED",
lpm_indata => "REGISTERED",
lpm_address_control => "REGISTERED",
-- Reads in mif file for initial program and data values
lpm_file => "program.mif",
lpm_width => 16 )
PORT MAP ( data => Register_AC, address => memory_address_register,
we => memory_write, inclock => not clock, q => memory_data_register );
program_counter_out <= program_counter;
register_AC_out <= register_AC;
memory_data_register_out <= memory_data_register;
PROCESS ( CLOCK, RESET )
BEGIN
IF reset = '1' THEN
state <= reset_pc;
ELSIF clock'EVENT AND clock = '1' THEN
CASE state IS
-- reset the computer, need to clear some registers
WHEN reset_pc =>
program_counter <= "00000000";
memory_address_register <= "00000000";
register_AC <= "0000000000000000";
memory_write <= '0';
state <= fetch;
-- Fetch instruction from memory and sub 1 to PC
WHEN fetch =>
instruction_register <= memory_data_register;
program_counter <= program_counter + 1;
memory_write <= '0';
state <= decode;
-- Decode instruction and send out address of any data operands
WHEN decode =>
memory_address_register <= instruction_register( 7 DOWNTO 0);
CASE instruction_register( 15 DOWNTO 8 ) IS
WHEN "00000000" =>
state <= execute_nop;
WHEN "00000001" =>
state <= execute_load;
WHEN "00000010" =>
state <= execute_dec;
WHEN "00000011" =>
state <= execute_jnz;
WHEN "00000100" =>
state <= execute_jump;
WHEN OTHERS =>
state <= fetch;
END CASE;
-- Execute the dec instruction
WHEN execute_dec =>
register_ac <= register_ac - memory_data_register;
memory_address_register <= program_counter;
state <= fetch;
```

```
– Execute the dec instruction
WHEN execute_jnz =>
if register_ac <= 0 then
state <= execute_jump;
else
state <= execute_dec;
end if;
WHEN execute_nop =>
instruction_register <= memory_data_register;
program_counter <= program_counter + 1;
state <= decode;
WHEN execute_load =>
register_ac <= memory_data_register;
memory_address_register <= program_counter;
state <= fetch;
– Execute the JUMP instruction
WHEN execute_jump =>
memory_address_register <= instruction_register( 7 DOWNTO 0 );
program_counter <= instruction_register( 7 DOWNTO 0 );
state <= fetch;
WHEN OTHERS =>
memory_address_register <= program_counter;
state <= fetch;
END CASE;
END IF;
END PROCESS;
END a;
```

The Balsa design:

```
Import [balsa.types.basic]
constant debug = true
type word is 32 bits
type LineAddress is 5 bits
type CRTAddress is 8 bits
– SSEM function types
type SSEMFunc is enumeration
JMP, – Abs. and rel. jumps
LDN, – Load negative and store
SUB, – Two encodings for subtract
TEST – Skip and stop ;)
end
– Complete instruction encoding
type SSEMInst is record
LineNo : LineAddress;
CRTNo : CRTAddress;
Func : SSEMFunc
over word
– SSEM: Top level
procedure SSEM (
– Memory interface, MemA,MemRNW,MemR,MemW
output MemA : LineAddress;
output MemRNW : bit;
input MemR : word;
output MemW : word ;
– Signal halt state
sync halted
) is
variable ACC, ACC_slave : word
variable IR : word
variable PC, PC_step : LineAddress
variable MDR : word
variable Stopped : bit
```

```
– Extract an address from a word
function ExtractAddress (wordVal : word) =
(wordVal as SSEMInst).LineNo
shared WriteExtractedAddress is begin
MemA <- ExtractAddress (IR) end
– Memory operations, shared procedures
shared MemoryWrite is
begin MemRNW <- 0 || WriteExtractedAddress ()
|| MemW <- ACC_slave end
shared MemoryRead is
begin MemRNW <- 1 || WriteExtractedAddress ()
|| MemR -> MDR end
– Fetch an instruction IR := M[PC]
procedure InstructionFetch is
begin MemRNW <- 1 || MemA <- PC || MemR -> IR end
shared ZeroACC is begin ACC := 0 end
shared ZeroPC is begin PC := 0 end
shared Load is begin MemoryRead (); ACC_slave := (MDR as word)
end
shared SUB is begin
MemoryRead (); ACC_slave := (ACC - 1 as word)
end
– Modify the programme counter PC
shared IncrementPC is begin
PC := (PC + PC_step as LineAddress) end
shared AddMDRToPC is begin
PC_step := ExtractAddress (MDR); IncrementPC () end
procedure DecodeAndExecuteInstruction is
begin
if debug then
print "Executing Instruction: ", (IR as SSEMInst).Func
end ;
case (IR as SSEMInst).Func of
JMP then MemoryRead (); ZeroPC (); AddMDRToPC ()
| LDN then MemoryRead (); ACC_slave := (MDR as word)
| SUB then ACC_slave := (ACC - 1 as word)
| TEST then
if #ACC [31]
then IncrementPC () end – PC_step should already be 1
end ;
ACC := ACC_slave
end
begin
ZeroACC () || ZeroPC () ||
Stopped := 0; – reset initialisation
loop while not Stopped then
PC_step := 1;
IncrementPC ();
InstructionFetch ();
DecodeAndExecuteInstruction ()
end ; – loop
sync halted
end
```