μSystems Research Group

School of Electrical and Electronic Engineering

Algebraic Specifications of ARM Cortex M0+ Instruction Set

Paulius Stankaitis

Technical Report Series

NCL-EEE-MICRO-TR-2014-192

Contact: paulius.stankaitis@ncl.ac.uk

Supported by EPSRC grant GR/XXXX

NCL-EEE-MICRO-TR-2014-192

µSystems Research Group

School of Electrical and Electronic Engineering

Merz Court

Newcastle University

Newcastle-upon-Tyne, NE1 7RU, UK

http://async.org.uk

Algebraic Specifications of ARM Cortex M0+ Instruction Set

Paulius Stankaitis

School of Electrical and Electronic Engineering

Newcastle University

A Dissertation submitted for the degree of

BEng (Hons) Electronic Engineering

May 2014

Abstract

Power consumption constraints became critical issue in microprocessor development. Active research is taking place to try and optimize this problem. This project tackles power constraint problem by further exploring fairly recently μSystem Research Group at Newcastle University introduced Instruction Set Architecture design flow, which is based on novel formalism called Conditional Partial Order Graphs. Study objective is to further explore this design flow by applying its methodology to ARM Cortex M0+ Instruction Set. Implementation of this design flow is split into four major parts and completed as follows: specification of instruction set, scenarios encoding, CPOG generation and mapping process. Throughout this project some interesting results were obtained, but highlight was very compact representation of ARM Instruction Set. Investigation of link between size of control logic and instruction set, also importance of encoding were analyzed and discussed. Project was concluded with discussion regarding its objectives, and whether they were accomplished or not.

Acknowledgements

I am particularly thankful to Dr Andrey Mokhov, who not only introduced me into this interesting field, but also enthusiastically guided and spent countless hours in discussions. Also express gratitude to people in μSystem Research Group, whose work indirectly contributed to this project. Finally, would like to thank to my supportive family and friends.

Table of Contents

Table of Figures & Tables

1 Introduction

1.1 Background and Motivation

Over the years aggressive transistor's scaling lead to immense microprocessor performance improvement, for instance microprocessor running frequency has increased 100 times more than theoretically predicted. However, power consumption of chip has been increasing and became crucial progress constraint [1]. Moreover, as in recent decade mobile electronics became more and more affordable to customers, devices operation time became vital. Though, battery life has not increased significantly [2], thus other power optimization approaches are being researched.

Reduction of power consumption through Instruction Set optimization approach has been known for a while. Nonetheless, it is still an active research area. In this field some important work has been done by µSystem Research Group at Newcastle University: introduction of formalism called, Conditional Partial Order Graph; new ISA design approach based on that model. Reduction of power limitation through ISA motivates to further explore this fairly recently introduced, though promising design approach.

1.2 Objectives

Project can be split into three major parts, where each had intermediate steps:

- Familiarize with µSystem Research Group introduced CPOG – based ISA design approach.
- Apply this methodology to ARM Cortex M0+ Instruction Set.
- Evaluate results.

1.2.1 Familiarization

This objective was mainly concerned with analysis of research papers produced by µSystem Research Group at Newcastle University and other relevant information e.g. ARM processors architecture.

1.2.2 Project Realization

After familiarization with key concepts of ISA design methodology and CPOG formalism project intends to apply this approach to ARM Cortex M0+ Instruction Set. Main aims of this part is to produce CPOG of Cortex M0+ IS and to compare SCENCO and ARM Encoded CPOGs

1.2.3 Project Evaluation

Perhaps, key objective of this project is interpretation of obtained results. Discussion and comparison between different encoding, control logic will be completed at this stage.

1.3 Thesis Overview

Chapter 2 explains some background information and key concepts, which is relevant to this project and is necessary to appropriately comprehend project. Moreover, it briefly overviews important research works, which highly affected this study.

Chapter 3 provides detailed methodology description, which is briefly introduced in Chapter 2 and used in this project. Procedures are structurally explained in their realization order.

Chapter 4 contains key results, which were obtained in this project. Results are presented structurally with respect to methodology described in Chapter 3.

Chapter 5 provides analysis and discussion of results and possible future work.

Chapter 6 summarizes and points out what objectives were met and what were not.

Appendices provided non key results.

2 Literature Review

2.1 Background

In 1971 Intel introduced, Intel 4004, a first microprocessor which revolutionized electronic. Complexity and performance of microprocessor has been dramatically increasing ever since. Nowadays, single chip contain over a billion transistors compared to 2108 Intel 4004 had, it is a remarkable achievement in only four decades. However, power efficiency became major issue, since consumers want their devices to work longer, run faster and be less expensive [3]. These desired parameters are highly constrained by power.

Numerous power efficient microprocessors design techniques are being developed as it is a very active research area. As discussed in [4] power optimization can be done on few design levels. However, this project is mainly concerned with low power component and architecture analysis through Instruction Set Architecture viewpoint.

2.2 Instruction Set Architecture

Instruction decoder is solid target for power efficiency improvement. Task of decoder is with given encoded binary code to enable particular microprocessor components in order to execute instruction. Therefore, size, latency and power consumption of decoder is highly affected by optimality of Instruction Set (IS). Nevertheless, identification of optimal instruction set is not an easy task involving many high-level decisions and computational calculations [5]. Moreover, plenty parameters of IS have to be considered by designers, but initial concerns are completeness, orthogonality and compatibility of instruction set [6]. General-purpose processors have common basic instructions, which are necessary for basic operations. Completeness measure has to assure that instruction set is sufficient to perform these basic operations. Richness of instruction set or orthogonality is particularly important parameter for power efficiency, as complex instructions increases the latency of the system and thus the energy dissipation (*Energy = Power x Time*).

Instruction format is another important feature which has to be considered. Instructions field consists of operational code (op-code) and operands, where operational code is unique binary signature used by decoder to differentiate between instructions. Decoder's critical parameters

like size and latency are highly affected by optimality of opcodes. Operands may be data (immediate operand) or address (addressed operand). Type and number of operands may vary within instruction as there are different functionality instructions.

According to instruction functionality they are categorized into 4 main groups: operate, memory access, control and miscellaneous. Operate instruction class mainly consist of arithmetic, logic and shift instructions and are found in majority of microprocessors. Operands of these instructions are normally user specified constants and source/destination registers. Control instructions class best example is branch instruction. Typically, instructions are executed sequentially in CPU. However, it is essential to have instruction, which execution would depend on previous instructions' outcome and would allow user to jump to specific instruction or execute sub-routine. Branch instruction has that functionality.

Instructions within memory-access class are used as name suggests transferring data from main memory to registers and vice versa. Operands in this class are usually used to specify memory address. Final class contains instructions, which do not fall in previous classes. Instructions may include interrupts, I/O instructions and processors state modes e.g. low power mode. Example of instruction set format and instruction fields is shown in Figure 1.

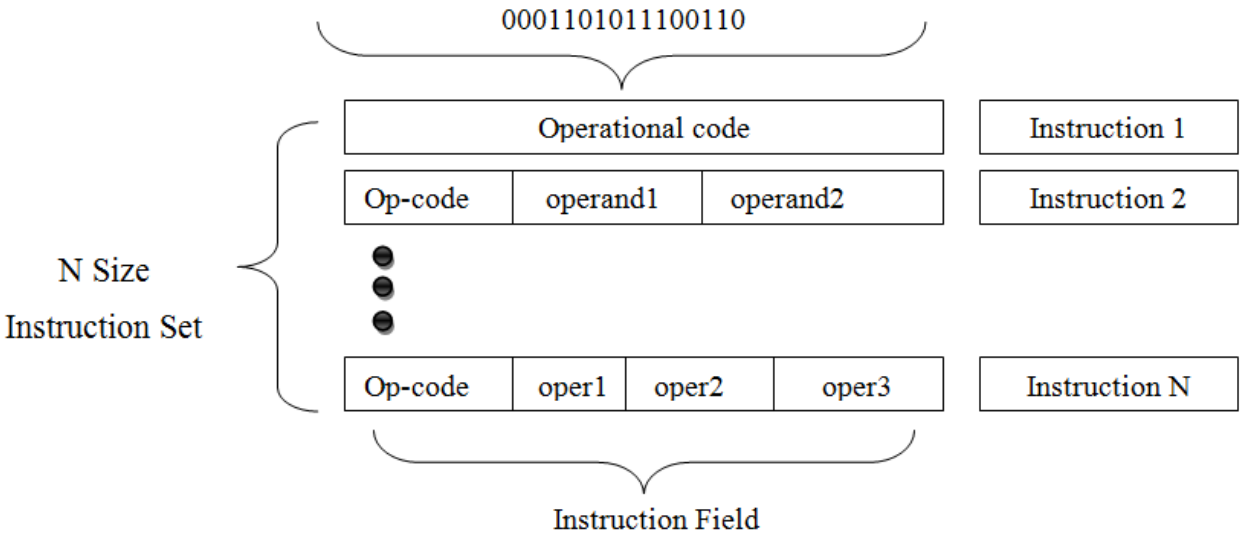


Figure 1, Example Instruction Set

2.2.1 ARM Cortex M0+ Instruction Set

This ARM product is known for its power efficiency and small number of gates, which makes it very popular choice for embedded systems. ARMv6-M architecture is used by Cortex M0+ processor, which supports majority of Thumb 16-bit and 32-bit instructions also few Thumb 2 technology instructions [7]. Register bank contains 13 general – purpose as well Stack Pointer (SP), Link (LR) and Program Counter (PC) registers, which can be used as operands. Moreover, processor has several special purpose registers, which are used for example to store flags from previous instructions, interrupts, execution status etc.

Instruction operational codes vary in position and size and according to ARM encoding can be grouped into following categories [8]:

- Shift (immediate), add, subtract, move, and compare
- Data processing
- Special data instructions and branch and exchange
- Load/store single data item
- Miscellaneous 16-bit instructions
- Hint instructions
- Conditional branch, and Supervisor Call
- Branch and miscellaneous control (32-bit instruction)
- Miscellaneous control instructions (32-bit instruction)

2.3 Conditional Partial Order Graphs (CPOGs)

Synthesis of systems, which contain many behavioral scenarios e.g. CPU microcontroller can be a challenging task. Therefore, sufficient models are needed to capture all these scenarios in compact and efficient form. Models like Petri Nets, Burst – Mode Finite State Machine and Signal Transition Graphs are relatively good as they produce higher performance circuits than syntax-directed translation from HDL approach as well as they capture concurrency and choice. However, these models are limited, when systems have numerous similar scenarios [9].

Novel model was developed by µSystem Research Group at Newcastle University called Conditional Partial Order Graph. CPOG is a combination of partial orders (POs) and its key advantage is compact and efficient representation of numerous behavioral scenarios [10].

As formalism name suggests, partial orders, are foundation for this model and is a concept of ordering actions and are ordered in a way to reflect causal dependencies. Similarly, processes can be expressed as partial orders by splitting whole process into single events and arranging them. Simple daily example could be a washing machine. User specified washing option would start a process, which consists of many intermediate each other dependent or independent events. These events can be expressed graphically or arithmetically to represent a whole process. Very basic washer operation example is given below. This example illustrates one of the partial orders feature, transitive arcs. Transitive arcs are indirect dependencies (dashed lines). Occasionally, they can be neglected without losing vital information.

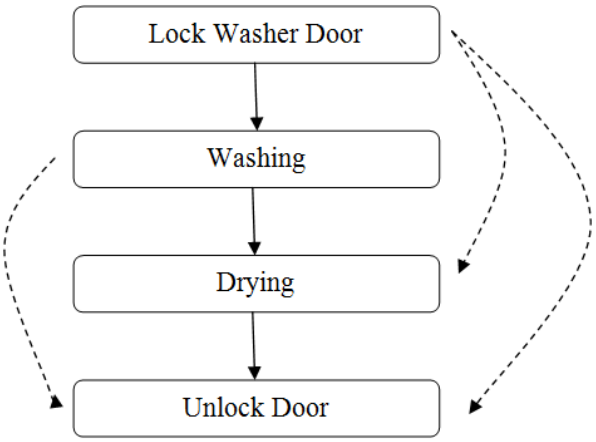


Figure 2, Example Partial Order

CPOG model uses Directed Acyclic Graphs notation, where events are denoted by vertices and dependencies between events by arcs. Operational codes are assigned to individual POs to distinguish between them in CPOG. Furthermore, condition function, ϕ, is used to turn off arcs and/or vertices according to the operational code (opcode) in order to make distinctive CPOG projections.

Likewise other models, CPOGs are represented graphically. In this model, circles denote vertices and arrows – arcs. Vertices are labeled and contain a name and condition separated by semicolon, arcs label only contains condition.

Example 1: Figure 2 shows an example of a CPOG. This CPOG contains two partial orders, which were given a different opcode. Since, there are only two behavioral scenarios, smallest amount of bits needed is one. Arcs ⊸ and vertices ◌ with dashed lines are considered as turned off. Therefore, under op code x = 1, vertices = {A, B, C} and arcs = {A→B, B→C} are enabled, creating projection for one of the scenarios (Figure 2, bottom-left). Second scenario is selected, when op code x is set to 0, this enables vertices = {A, D, E} and arc = {D→E} (Figure 2, bottom-right). In this example vertex A was unconditional, as it belonged to both partial orders, so vertex condition $\phi(A) = \bar{x} \vee x = 1$. Example 1 for simplicity and clarity purposes contained only two scenarios, but CPOG can be extended and contain numerous scenarios in a compact and efficient form, making this model very attractive for e.g. CPU microcontroller modeling.

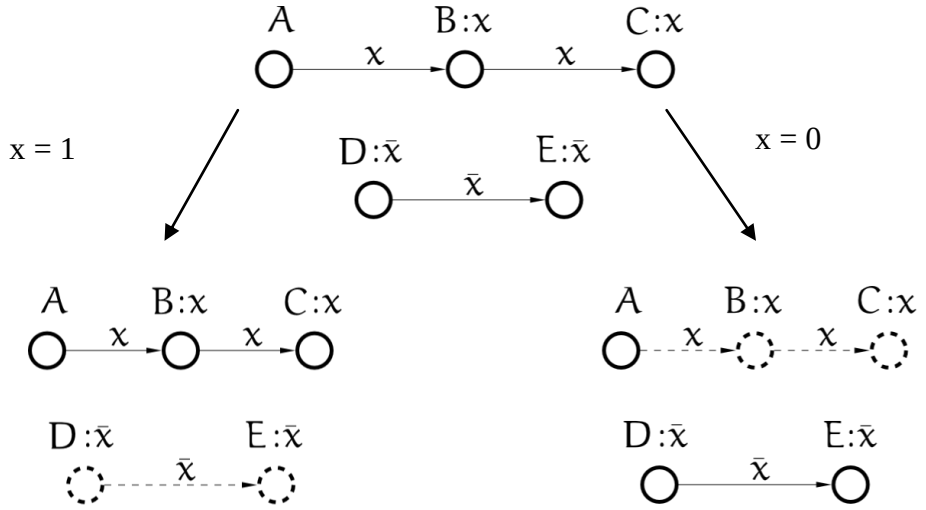


Figure 3, Example of CPOG

Obtained CPOGs complexity can be compared or analyzed in terms of number of literals in arc and vertices conditions. Formula 1.1 sums literals in all vertices and arcs of CPOG [10].

$$C(H) = \sum_{v \in V} C(\phi(v)) + \sum_{e \in E} C(\phi(e)) \qquad (1.1)$$

2.3.1 Further Research on CPOGs

In research project [11] done by μSystem Research Group at Newcastle University new Instruction Set Architecture design approach based on CPOGs was introduced. Initially project explored models like Petri Nets (PNs), Finite State Machines (FSMs) for control logic synthesis. However, it was Conditional Partial Order Graphs (CPOGs) formalism, which attained the most attention. Hence, project progressed with objective to prove this models practicality by realizing it as asynchronous microcontroller.

Intel 8051 instruction set with total 257 (including 2 non-standard) instructions was chosen, as it was well explored and popular. However, regarding implementation researchers made few significant changes for instance asynchronous architecture was used over synchronous also datapath was extended with additional computational units (adder, multiplier and divider).

In the project, control logic design involved extraction of datapath components and instructions partial orders. Due to similarity among instruction execution patterns, they were grouped and had a single partial order for that class. Project composed 37 distinct classes, which were assigned with opcode using Huffman encoding method, which then lead to CPOG generation and mapping it to Boolean equations. Further work was done on designing ALU control logic, data path, testing, and verification of manufactured chip.

Results of this project confirmed practicality of introduced novel design method of microprocessors instruction set architecture and methodology of this work was a followed in Algebraic Specifications of ARM Cortex M0+ Instruction Set project.

2.4 Conclusion

This chapter introduced and briefly discussed two problems: microprocessors power consumption constraints and lack of sufficient model for systems with many behavioral scenarios. However, as of one the solution was proposed optimization of Instruction Set Architecture using novel model developed μSystem Research Groups' at Newcastle University. As it was already been demonstrated that Conditional Partial Order Graph produce compact and efficient form of systems [12] and was used as basis for new ISA design approach. Therefore, this project intends to continue analysis of CPOG model with ARM Cortex M0+ instruction set.

3 Implementation

3.1 Design Flow

Instruction Set Architecture, as discussed in Section II, is one of key microprocessor design flows. Instruction set not only determines microprocessor functionality, but also highly affects performance and energy efficiency. Therefore, adequate design methodology is necessary to deal with large number of behavioral scenarios, but also assure correctness if IS modification is needed.

µSystem Research Group (Newcastle University) introduced new ISA design approach [11], which is based on formalism called CPOG (See Chapter II), which was previously introduced by the same research group. CPOG formalism is an extremely powerful tool for modelling systems with many behavioral scenarios and introduced ISA design method uses it for ISA design. Alternative tools like Event-B [13] and HOL [14] are well known for system specification and verification. However, the lack of hardware consideration and higher cost makes CPOG based method more suitable for ISA design.

Brief description of design flow is provided below. Following sections, explains each step in greater details

Specification of IS: Initial step in introduced methodology is instruction analyzes and transformation as partial order. After that, instructions with the same PO are clustered and instruction groups created.

Encoding: Instruction classes have to be encoded, so that CPOG conditions could be created and POs could be distinguished in CPOG.

Generation of CPOG: After POs are specified and encoded, CPOG, which is a composition of all POs, can be generated.

Mapping: Final methodology stage is designing control logic. Obtained CPOG is mapped into Boolean equations, using request – acknowledgement handshake protocol.

3.2 Specification of Instruction Set

Methodology used in this project required transformation of instructions into partial orders. However oppositely from [11], this project only intended to investigate top – level control, which results in simply partial decoding of instructions. Nonetheless, first step is to express instructions as partial orders, which is done by analyzing instructions. Construction of partial orders was done on Workcraft tool [15][18], which enables user to represent partial orders graphically, assign conditions and generate CPOGs

Past work done by µSystem Research Group, extracted five key datapath units from Intel 8051 instruction set, four of them were reused in this project, since their captured functionality was adequate for examined part of Cortex M0+ instruction set. Table 1 gives a list of datapath components and their description. It is important to mention that at top – level design stage there is no distinction between functions of ALU (addition, subtraction etc.) [16]. Furthermore, some partial orders required to use the same unit more than once, so to distinct between them additional units were given numbered indexes (IFU/2, PCIU/2)

Table 1, datapath components

Component	Description
MAU (Memory Access Unit)	Access internal and external memory
ALU (Arithmetic Logic Unit)	Executes mathematical operations.
PCIU (Program Counter Increment Unit)	Increments Program Counter (PC).
IFU (Instruction Fetch Unit)	Provides opcodes to Instruction Register (IR).

Derivation of partial orders involved deep analysis of individual instruction, which was done using ARMv6-M technical manual [8]. Manual provides detailed information about every instruction including encoding, operation pseudocode and function of instruction. Process was simplified due to ability to cluster instruction and give class single PO. Grouping Instructions chapter provides better insight in this.

3.2.1 Grouping Instructions

Clustering instructions can highly simplify process of specifying instruction set. Design flow [11] expressed every instruction as partial order and then grouped instruction with the same PO. Approach in this project was slightly different. Grouping instructions was done first and then class was assigned with appropriate PO. Instructions were grouped according to two main criteria: execution similarity and type of addressing mode. Instructions with similar execution were initially clustered. However, grouped with similar execution instructions could be further split if instructions in that group used different type of addressing. Instruction differentiation using addressing and type of operands is discussed subsequently.

Memory access instructions contain an offset, which is used to determine accessing address. This offset can be immediate or register type. A constant, could be declared alongside opcode, which would indicate that address is calculated using base register address and declared constant, which is added or subtracted from base register address (immediate addressing). Another option is address calculation using offset register, which is also declared by programmer. User specified offset register contains a value, which would be added or subtracted from the base register address to form address, which user tries to access.

Arithmetic instructions, is another class, which uses two addressing modes. Single arithmetic instructions can use immediate operand (a constant) or register operand. Instruction with specified register operand uses value of that registers together with possibly other register values to perform arithmetic operation. Similarly, like memory access instructions immediate operand is also possible. A constant is declared as instruction parameter an example of addition instruction is shown below, where Rd is destination register, Rt – register with first operand, #imm3 – a constant (0 – 7).

ADDS Rd, Rn, #imm3 (immediate addressing) [8]

This type of distinction among instructions was crucial, as instruction could have more than one partial order depending on type of its operands or addressing. Instructions with immediate operand share a common operation, a constant fetching, which in PO was denoted as action PCIU → IFU. However, the same PCIU → IFU has another function (next instruction fetch), when instruction uses register operand/addressing and does not require constant fetching.

3.3 Encoding Partial Orders

Once all partial orders are obtained CPOG-based design flow requires encoding every partial order. Importance of encoding was already discussed in Chapter II and concluded that optimal instruction encoding results in more efficient and compact control logic. Furthermore encoding is necessary for distinguishing POs in CPOG.

Experiments in this project used three types of encoding. SCENCO encoding was used in first part of experiment, where objective was to produce CPOG of analyzed instruction set. Further work required modification of SCENCO encoding and derivation of ARM, because project intended to compare ARM and SCENCO encoded CPOGs. Derivation process of SCENCO - modified and ARM encodings was a more complex compared to SCENCO encoding. Because of, a lot of instructions groups contained instructions with different opcodes and lengths of reserved bits[1], thus highly affecting derivation process. These three encodings are explained below.

3.3.1 SCENCO Encoding

SCENCO encoding was the most straightforward and produced computationally. A Workcraft tool has integrated SCENCO (**SCEN**ario **ENCO**der) plugin, which produces optimal encoding as well as CPOG for given POs. Therefore, it was only necessary to specify obtained POs and number of encoding bits in Workcraft tool and effortlessly optimal encoding was provided. Note that number of bits needed to encode N scenarios, can be calculated by formula $bits = \log_2 N$. Thus, minimum bits required to encode nine different scenarios $bits = \log_2 9 \approx 4$, number has to be rounded to higher value.

3.3.2 ARM Encoding

As it was mentioned particular instruction group may have contained instructions with different opcodes. So, to produce a single opcode for that class further operations were done. Firstly, each instructions opcodes were expressed as Boolean equation using opcodes provided in [8]. Then, using OR operation all Boolean equations from individual group were combined, thus producing single expression for that class. It is important to mention that obtained encoding length varied as some groups used nine bits, while others just four. Moreover, three classes had OR operators in

[1] Reserved bits – bits, which are not used in decoding e.g. operands

their opcode, thus making this operational code off standard. Nonetheless, it was suitable for CPOG generation purposes. Equations were combined and minimized using Logic Friday tool [17].

3.3.3 SCENCO – modified Encoding

Modifications of SCENCO encoding were necessary in order to make reasonable comparison between ARM and SCENCO encoded CPOGs. Major issue, was reserved bits constrains. Five groups out of nine did not allow using full 4-bits encoding produced by SCENCO plugin. So, some bit length manipulations were necessary.

Figure 3 gives a brief look of encoding derivation process (See Appendix for full worksheet). Spreadsheet illustrates: new encodings together with indicated reserved bits, name of the instructions and group it belongs. Some groups, like G1 (row 3) was allowed to use SCENCO 4-bit encoding, since it only had single instruction and had enough unrestricted bits. On the other hand, for some of the groups 4-bit encoding was not permitted e.g. G2 due to number of bits for low – level opcode[2]. Therefore, G2 top – level opcode[3] had to be reduced to 2-bits. Similar process was continued with remaining groups, with operational code uniqueness criterion in mind. It is important to mention that since this encoding design process was just partially automated, obtained opcodes were not optimal.

[2] Low – level opcode – operational code used to differentiate instructions within class
[3] Top – level opcode – operational code used to differentiate between classes of instructions

1	!x15	!x14	!x13	!x12	!x11	!x10	!x9	!x8	!x7	!x6	!x5	!x4	!x3	!x2	!x1	!x0	Instruction	Group
2	x15	x14	x13	x12	x11	x10	x9	x8	x7	x6	x5	x4	x3	x2	x1	x0		
3	1	1	1	0	RESERVED												B	1
4	0	1	0	0	0	0	0	RESERVED									ADD(imm.)	2
5	0	1	0	0	1	RESERVED											ADD(SP+imm.)	2
6	0	1	0	1	0												ADR	2
7	0	1	1	0	0												ASR (imm.)	2
8	0	1	1	0	1												CMP(imm.)	2
9	0	1	1	1	0												LSL(imm.)	2
10	0	1	0	1	1												LSR(imm.)	2
11	0	1	1	1	1												MOV(imm.)	2
12	0	1	0	0	0	0	1	X	X	X	RESERVED						RSB(imm.)	2
13	0	1	0	0	0	1	0	RESERVED									SUB(imm.)	2
14	0	1	0	0	0	1	1	X	X	RESERVED							SUB(SP-imm.)	2
15	1	1	0	1	1	1	0	0	0	0	RESERVED						ADC	3
16	1	1	0	1	0	0	0	RESERVED									ADD(reg.)	3
17	1	1	0	1	0	0	1	X	R	X	X	X	X	RESERVED			ADD(SP+reg.)	3
18	1	1	0	1	1	1	0	0	0	1	RESERVED						AND (reg.)	3
19	1	1	0	1	1	1	0	0	1	0							ASR(reg.)	3
20	1	1	0	1	1	1	0	1	0	0							BIC	3
21	1	1	0	1	1	1	0	1	1	0							CMN(reg.)	3
22	1	1	0	1	1	1	0	0	1	1							CMP(reg.)	3
23	1	1	0	1	1	1	0	1	0	1							EOR(reg.)	3
24	1	1	0	1	1	1	0	1	1	1							LSL(reg.)	3
25	1	1	0	1	1	0	1	0	0	0							LSR(reg.)	3
26	1	1	0	1	1	0	0	X	RESERVED								MOV(reg.)	3

Figure 4, part of SCENCO-modified encoding

3.4 Producing CPOG

After instructions are grouped and expressed as partial orders also assigned with opcode, CPOG can be generated. Synthesis of CPOG creates arcs and vertices conditions according to generated operational codes, thus allowing enabling/disabling particular PO in CPOG. Likewise instruction groups encoding, project completed computational and manual CPOG generation. Necessity for manual CPOG generation rose from lack of suitable computational tools.

Computational part of synthesis used SCENCO plugin, which was previously discussed in encoding chapter. This plugin is not only capable of producing optimal encodings, but is also able to generate CPOG in Workcraft tool. However, at the moment this tool is not capable of generating CPOGs with different length encodings. Thus, ARM and SCENCO – modified encoded CPOGs were generated manually.

Manual synthesis of CPOG was mainly concerned with creating arcs and vertices conditions. Idea was to express ARM and SCENCO – modified opcodes as Boolean equations, then combine particular equations to create a condition for specific arc/vertex. In example, only instruction classes 5, 6, 7 and 8 contained MAU unit in their partial order. Therefore, MAU condition was produced by merging Boolean expression of operation codes of 5, 6, 7 and 8

instruction classes. In other words, vertex or arc has to be enabled under one of these groups opcode. Process was extended to all arcs and vertices. Logic Friday tool [17] was used to minimize Boolean equations. Figure 4 shows a layout of CPOG as well as instruction group numbers beside arc/vertex, which indicate what groups were used to construct specific condition (see Section IV for Instruction Groups).

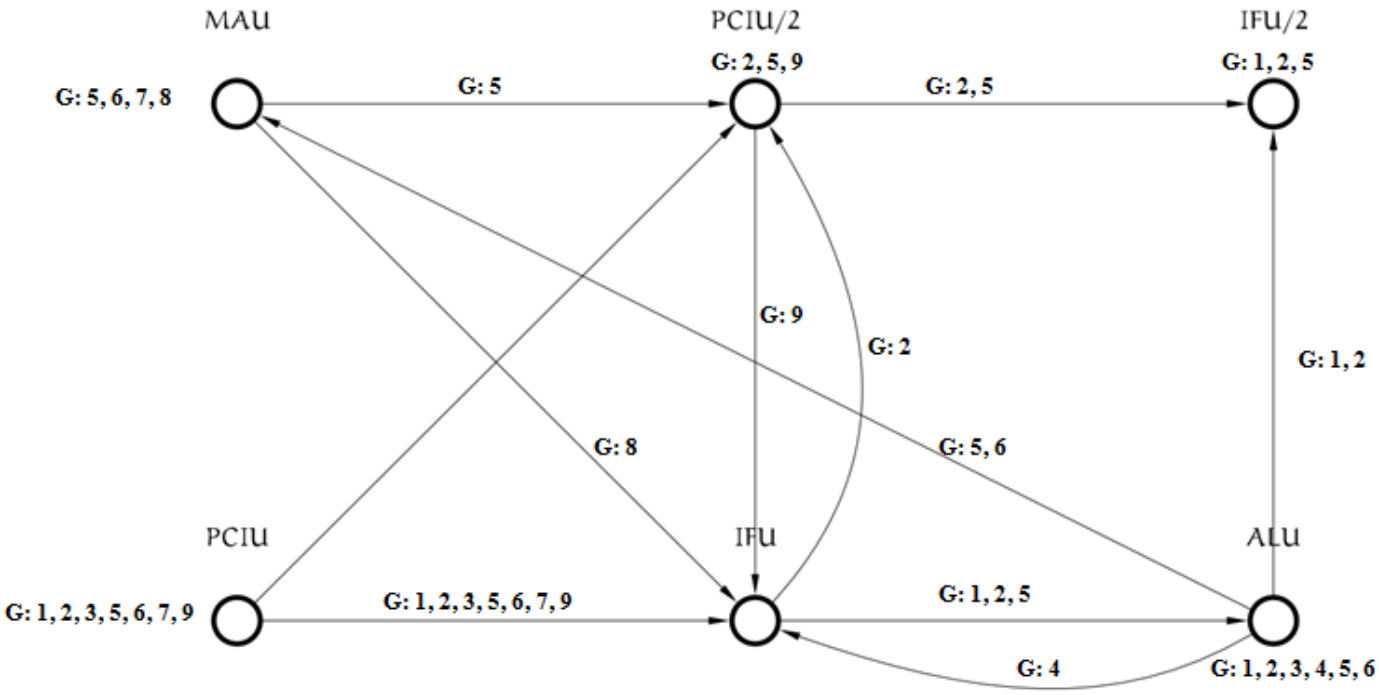


Figure 5, Groups for synthesis of vertices/arcs conditions and CPOG layout

3.5 Mapping

Once system's CPOG is obtained, design process is continued with mapping it to Boolean equations in order to produce control logic. Mapping method used in this project exploits asynchronous architecture's request – acknowledgement handshake protocol. In this protocol request signals are generated by controller, which receives opcodes and acknowledgement signals and decides what components need to be enabled and in what order.

Acknowledgment signals are generated by previously requested components, which completed their execution. Design flow on, which this project is based on also, introduced go and done signals.

Input signal go, enables instruction execution, while output signal done is produced after instruction is completed. Visually controller is shown in Figure 5 [10].

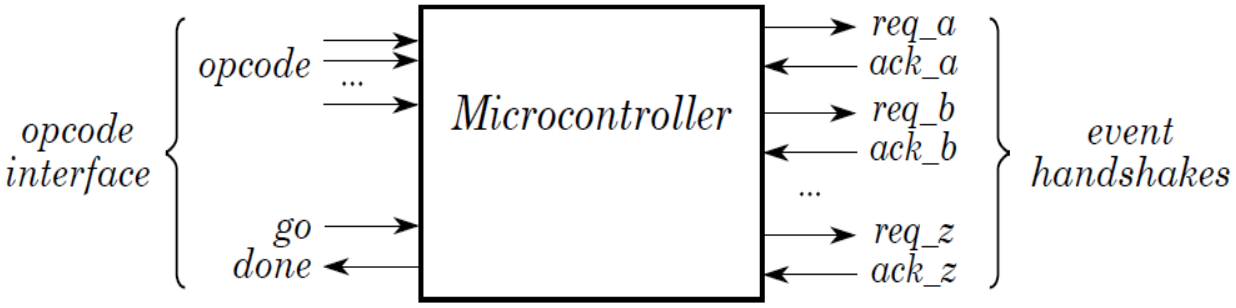


Figure 6, request-acknowledgment protocol microcontroller

Request signals are sent if following conditions are satisfied (see Eq. 2.1):

- Previous instruction completed, go signal set to 1
- Vertex condition $\phi(v) = 1$, specific vertex is enabled according to received opcodes
- $(\phi(u) \cdot \phi(u,v) \Rightarrow ack\_u) = 1$, all previous vertices completed their execution and so acknowledgement signals were received

Done signal is sent when all vertices acknowledgements are received. This is captured by equation 2.2.

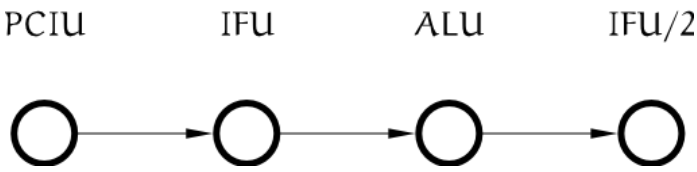
$$\text{req\_v} = \text{go} \cdot \phi(\text{v}) \cdot \prod_{u \in V} (\phi(u) \cdot \phi(u,v) \Rightarrow ack\_u) \tag{2.1}$$

$$done = \prod_{u \in V} (\phi(v) \Rightarrow ack\_u) \tag{2.2}$$

4 Results

4.1.1 Class 1

Partial Order



Description: Class 1 instruction class contained only an unconditional branch instruction. Branch instruction causes a jump to specific place of the program, specified by the programmer and comes in two flavors: conditional and unconditional. If condition of the instruction is met, instruction proceeds normally and causes a jump; otherwise instruction works as NOP instruction and continues with next instruction. However, this PO only captured unconditional side of branch instruction.

Firstly, events PCIU and IFU fetch a constant, which is an offset value of label calculated by a compiler. These events are followed by vertex ALU, which adds obtained constant to Program Counter (PC). Lastly, next instruction is fetched by IFU/2.

Note that this PO does not capture a fact of conditional branching, though ALU has access to status register, which contains status flags and could determine whether condition was met or not. So, this PO could potentially be combined with Class 9 (NOP instruction), though for simplicity purpose these two classes are separated.

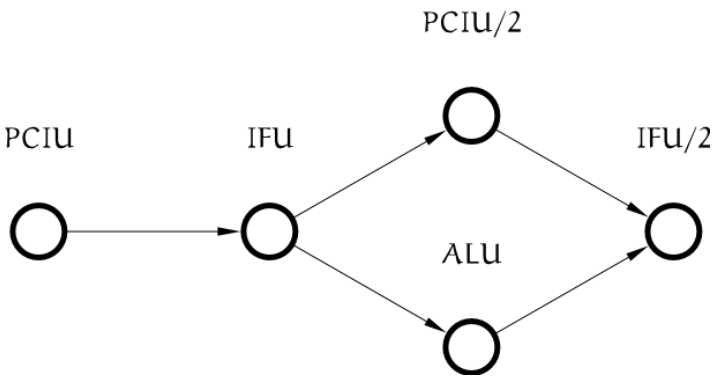
SCENCO Encoding	x0	x1	x2	x3
Class	1	1	1	0

SCENCO - modified Encoding	x15	x14	x13	x12
Class	1	1	1	0

ARM Encoding	x15	x14	x13	x12	x11	x10	x9	x8	x7	x6
Class	1	1	0	1	X	X	X	X	X	X

4.1.2 Class 2

Partial Order



Instructions: ADD (imm.), ADD (SP + imm.), ADR, ASR (imm.), CMP (imm.), LSL (imm.), LSR (imm.), MOV (imm.), RSB (imm.), SUB (imm.), SUB (SP - imm.);

Description: Class 2 contained 11 instructions, which shared similarity in their execution and type of addressing. Abbreviation used in instruction brackets stands for immediate and it is one of addressing types discussed in Section 3, Instruction Grouping.

PO starts with vertices PCIU and IFU, since instructions with immediate addressing require to fetch a constant into Instruction Register (IR). After, constant is fetched; ALU can perform its task concurrently with increment of program counter (PCIU/2). Lastly, when PCIU/2 and ALU are completed, next instruction is fetched, IFU/2.

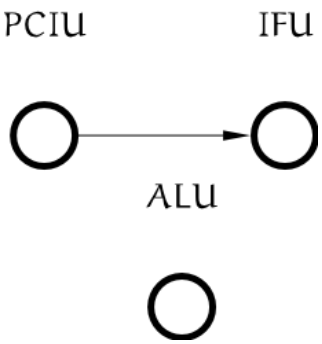
SCENCO Encoding	x0	x1	x2	x3
Class	1	1	1	1

SCENCO - modified Encoding	x15	x14	x13	x12
Class	0	1	X	X

ARM Encoding	x15	x14	x13	x12	x11	x10	x9	x8	x7	x6
Class	See Appendix A									

4.1.3 Class 3

Partial Order



Instructions: ADC (reg.), ADD (SP + reg.), AND (reg.), ADD (reg.) ASR (reg.), BIC (reg.), CMN (reg.), CMP (reg.), EOR (reg.), LSL (reg.), LSR (reg.), MOV (reg.), MUL, MVN (reg.), ORR (reg.), SXTB, SXTH, TST, UXTB, UXTH;

Description: Similarly to Class 2 this group covered arithmetical, logical and data – copy instructions. However, key difference between these classes is addressing mode, which resulted in different partial orders.

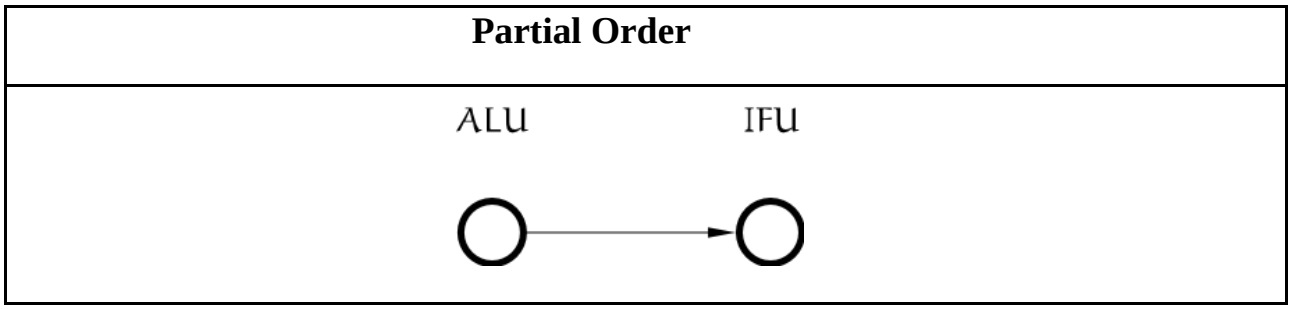
Partial Orders comprises of two concurrent processes, ALU operation and next instruction fetching denoted by events PCIU and IFU. This is allowed due to register addressing mode, where operands are stored in registers, thus oppositely from immediate addressing does not require fetching into IR.

SCENCO Encoding	x0	x1	x2	x3
Class	1	1	0	1

SCENCO - modified Encoding	x15	x14	x13	x12
Class	1	1	0	1

ARM Encoding	x15	x14	x13	x12	x11	x10	x9	x8	x7	x6
Class	See Appendix A									

4.1.4 Class 4



Instructions: BLX (reg.), BX;

Description: Instructions included in Class 4 are branch type: Branch with Link and Exchange (BLX), Branch and Exchange (BX). BLX and BX instruction functionality and addressing mode is similar, as they both cause branch to specific address location and use register addressing. The only difference is that BLX instruction saves current address to Link Register (LR) before branching. After branched instruction is executed, address, which was saved at Link Register, is used as next instruction address.

ALU event in this partial order could do few things. Even though, as discussed in Chapter 3, Specification of Instruction Set at this design stage there is no differentiation among specific ALU functions (addition, subtraction etc.). ALU, in BLX and BX instructions, would move base register address to PC counter register and specifically in BLX instruction ALU would also copy current address into Link Register. Regarding addition instruction, ALU would execute register addition. Vertex ALU is followed, by IFU event, which indicates next instruction fetch.

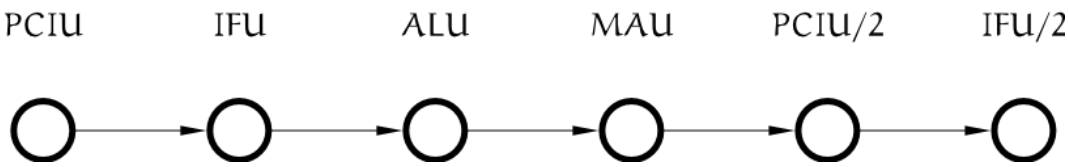
SCENCO Encoding	x0	x1	x2	x3
Class	1	1	0	0

SCENCO - modified Encoding	x15	x14	x13	x12
Class	1	1	0	

ARM Encoding	x15	x14	x13	x12	x11	x10	x9	x8	x7	x6
Class	0	1	0	0	0	1	1	1	X	X

4.1.5 Class 5

Partial Order



Instructions: LDR (imm.), LDR (literal), LDRB (imm.), LDRH (imm.), STR (imm.), STRB (imm.) STRH (imm.);

Description: Instructions covered by this class are memory related. Load Register (LDR) instructions loads programmer defined memory location into register, while Store Register (STR) instructions do opposite and store register into memory. Moreover, these instructions come in few flavors depending on store/load information size (byte, half word, and word). Address is calculated using base register address, in addition to specified constant offset, indicating immediate addressing.

Likewise all immediate addressing POs it starts with constant fetching, events PCIU and IFU, followed by ALU, which uses offset and base register address to calculate memory address, which is being accessed. Memory access unit (MAU) uses that address to transfer data. Lastly, next instruction can be fetched. A PCIU/2 increments PC, which now points to next instruction and event IFU/2 fetches next instruction.

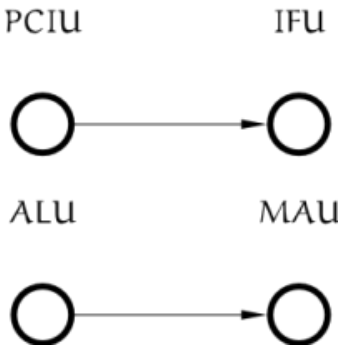
SCENCO Encoding	x0	x1	x2	x3
Class	1	0	1	1

SCENCO - modified Encoding	x15	x14	x13	x12
Class	0	0	X	X

ARM Encoding	x15	x14	x13	x12	x11	x10	x9	x8	x7	x6
Class	See Appendix									

4.1.6 Class 6

Partial Order



Instructions: LDR (reg.), LDRB (reg.), LDRH (reg.), LDRSB (reg.), LDRSH (reg.), STR (reg.), STRB (reg.), STRH (reg.);

Description: Even though, functionality of instructions in Classes 5 and 6 are the same, type of addressing differentiate these two POs. This class uses, already discussed register addressing mode. Hence, address is calculated using base register address and offset register value.

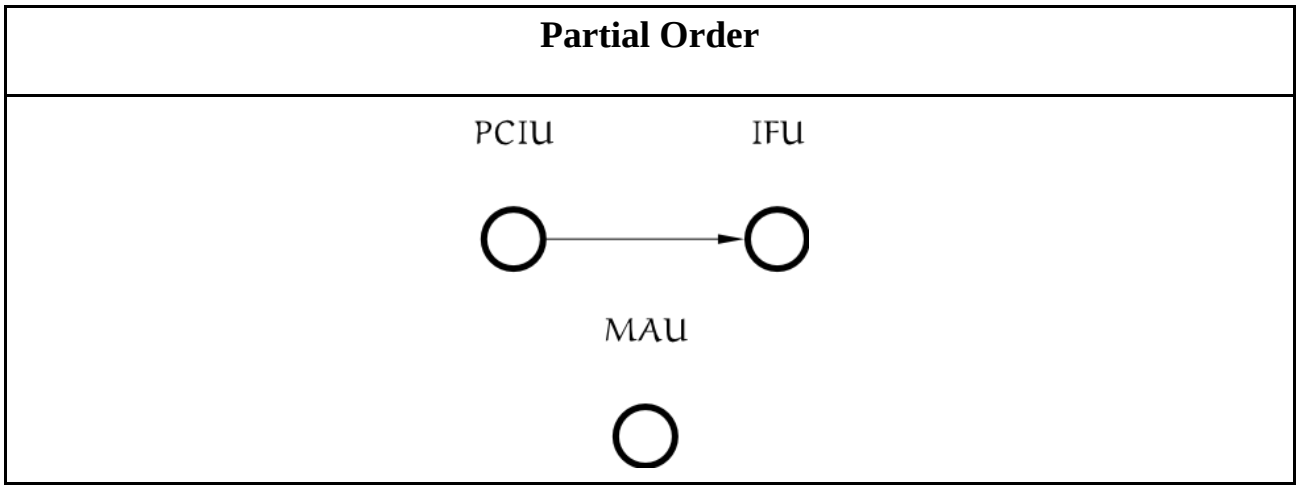
Register addressing mode does not require constant fetching to IR. So, this partial order, consist of two concurrent processes. In one of the processes, address is calculated and then data is transferred, events ALU and MAU. Simultaneously, next instruction is fetched by vertices PCIU and IFU.

SCENCO Encoding	x0	x1	x2	x3
Class	1	0	0	1

SCENCO - modified Encoding	x15	x14	x13	x12
Class	1	0	0	1

ARM Encoding	x15	x14	x13	x12	x11	x10	x9	x8	x7	x6
Class	0	1	0	1	X	X	X	X	X	X

4.1.7 Class 7



Instructions: LDM, LDMIA, LDMFD, PUSH, STM, STMIA, STMEA;

Description: Class 7 is another memory access related instruction group. Instructions in this class are capable of transferring more than one memory location or register. Although, only general purpose registers (R0 – R7) are allowed to be used in loading or storing process, thus for specification of register list 8 – bits are used.

Similarly to other register addressing mode instruction classes, two concurrent processes are taking place. Memory access unit, MAU, can transfer data between registers and memory, while next instruction can be fetched, events PCIU and IFU.

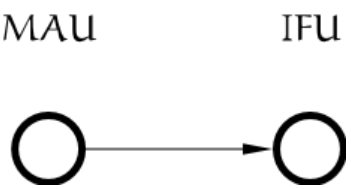
SCENCO Encoding	x0	x1	x2	x3
Class	0	0	0	1

SCENCO - modified Encoding	x15	x14	x13	x12
Class	1	0	1	X

ARM Encoding	x15	x14	x13	x12	x11	x10	x9	x8	x7	x6
Class	See Appendix									

4.1.8 Class 8

Partial Order



Description: Class 8 contained only a single instruction, POP. POP instruction is memory type instruction, more specifically stack instruction. This instruction, loads multiple memory locations from stack into registers and if program counter is used as destination register, branch occurs. Instruction field uses 8-bits, to specify register list (R0 – R7) and extra P – bit, to specify PC if necessary.

PO of this class is pretty straightforward, as MAU, uses base address to transfer data between memory and registers. After, data transfer is finished, next instruction can be fetched, IFU. Note, that this PO does not indicate in any way if branched occurred or not, next instruction is fetched whether PC register was used as destination for transfer or not.

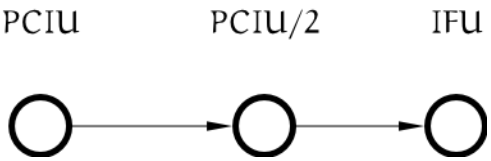
SCENCO Encoding	x0	x1	x2	x3
Class	0	0	0	0

SCENCO - modified Encoding	x15	x14	x13	x12
Class	1	1	1	1

ARM Encoding	x15	x14	x13	x12	x11	x10	x9	x8	x7	x6
Class	1	0	1	1	1	1	0	1	1	1

4.1.9 Class 9

Partial Order



Description: Last instruction class has a single instruction, called NOP. This instruction does nothing and just proceeds to next instruction.

SCENCO Encoding	x0	x1	x2	x3
Class	0	1	1	1

SCENCO - modified Encoding	x15	x14	x13	x12
Class	1	0	0	0

ARM Encoding	x15	x14	x13	x12	x11	x10	x9	x8	x7	x6
Class	1	0	1	1	1	1	1	1	0	0

4.2 Generation of CPOG

This section provides generated CPOGs, which were obtained using derived POs with different encodings. Firstly, computationally generated CPOG is shown, which used optimal SCENCO encoding and is followed by few of it PO projections. Subsequently shown CPOGs were derived manually with intention to compare ARM and SCENCO encoded CPOGs. Greater CPOG analysis is provided in Discussion chapter.

4.2.1 SCENCO Encoded CPOG

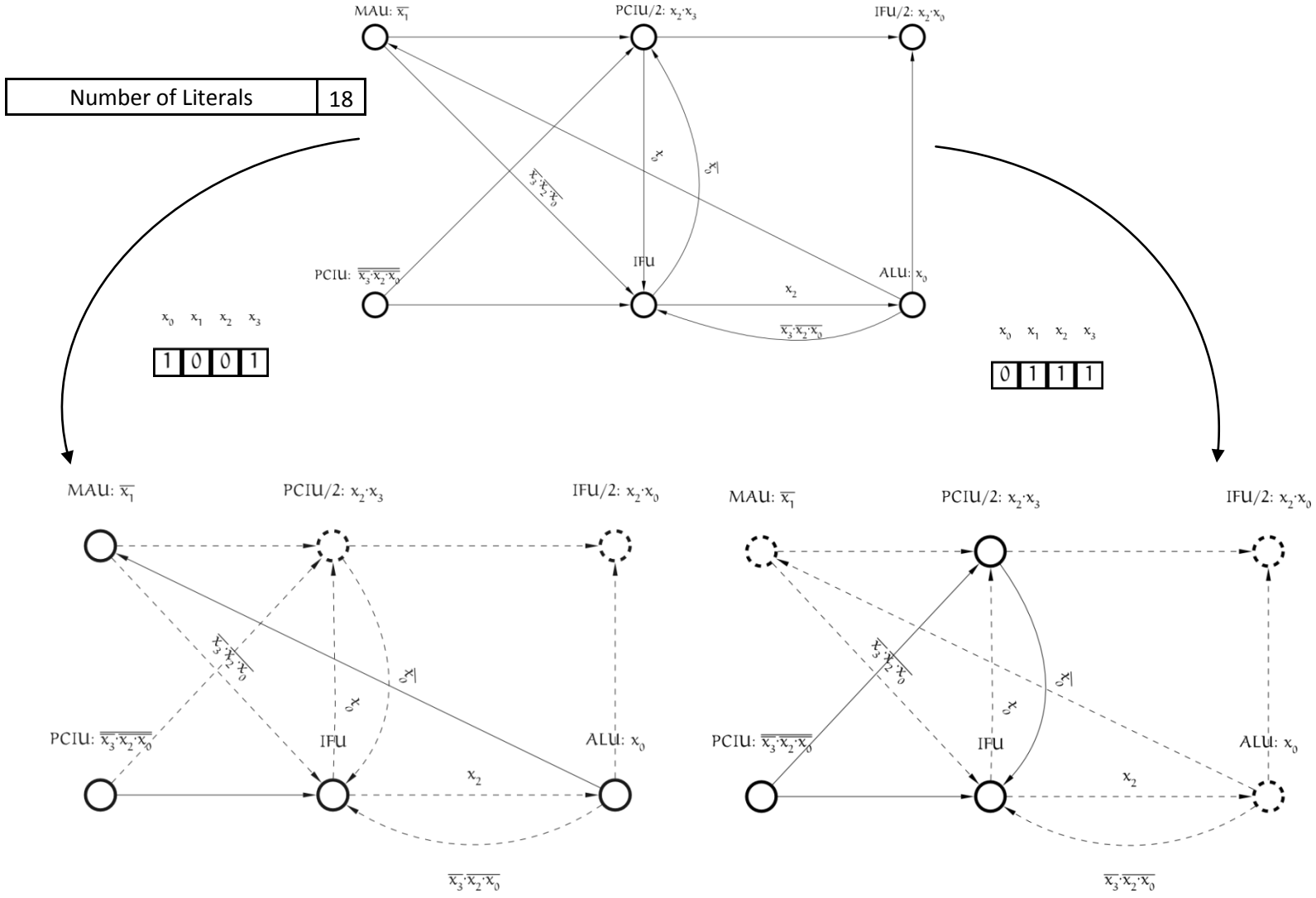


Figure 7, SCENCO Encoded CPOG

4.2.2 SCENCO – modified Encoded CPOG

Number of Literals	70

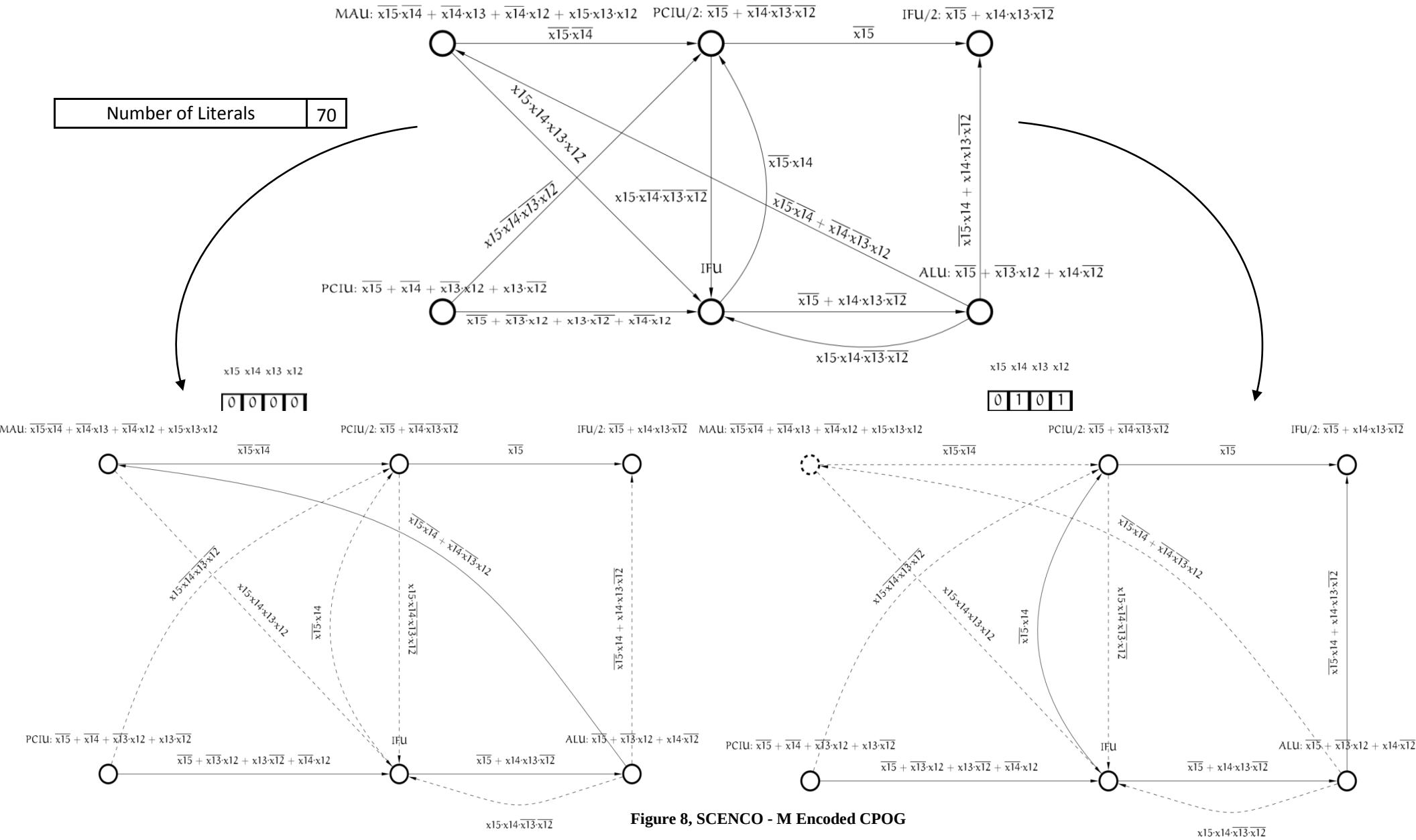


Figure 8, SCENCO - M Encoded CPOG

4.2.3 ARM Encoded CPOG

This CPOG vertices and arcs conditions are given in Appendix J.

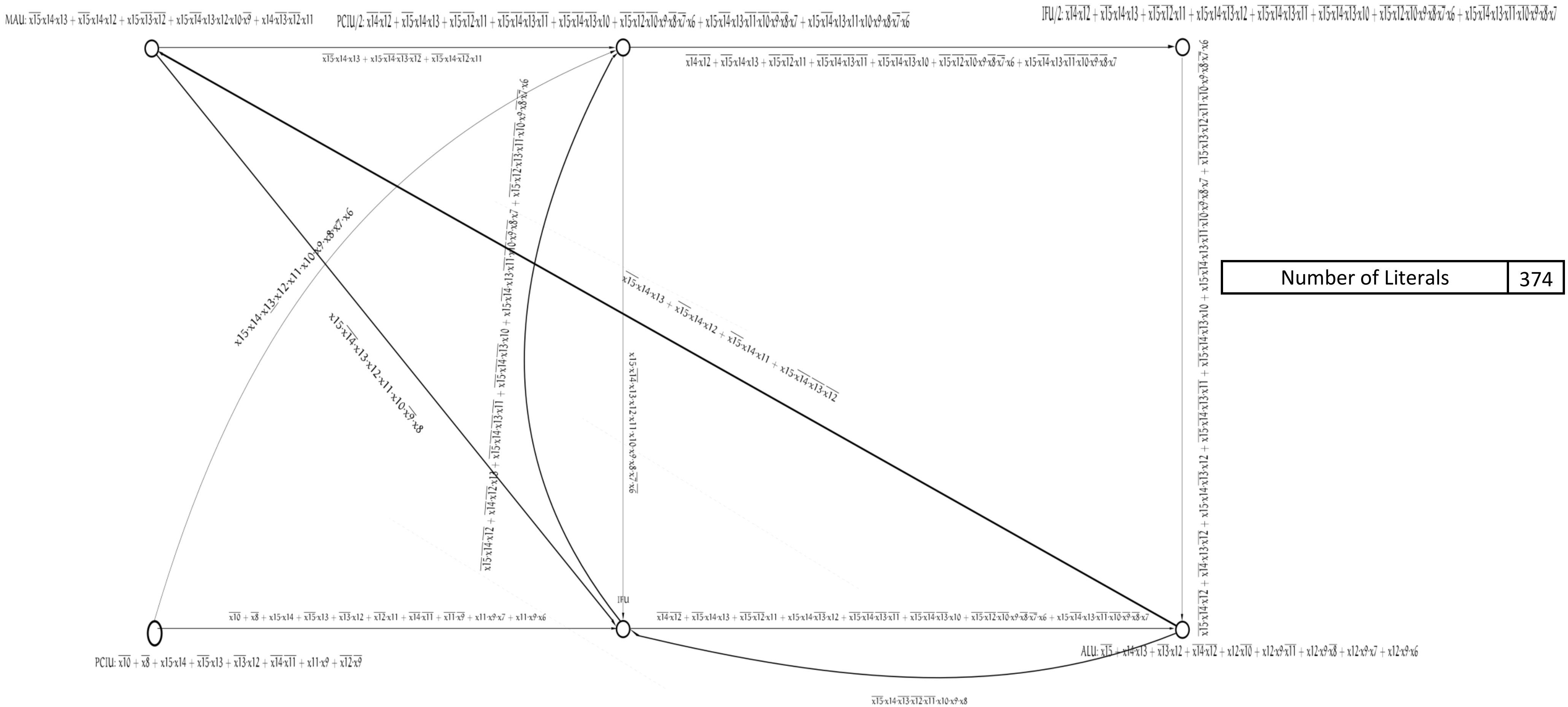


Figure 9, ARM Encoded CPOG

4.3 Top – Level Control Logic

Methodology followed in this project requires mapping CPOGs into Boolean equations in order to produce control logic. This was done by expressing CPOG conditions into equations, which are given in Section 3.5 and combination of them shaped top – level control circuit. Note that this was done for all three CPOGs, though in this chapter only SCENCO encoded CPOG is shown. Transformation process of Boolean equations to control circuit used Logic Friday tool [17]. Tool not only is capable of producing logic circuit, but also has a function called Trace Gate Logic, which allows user to enter opcode and simulate instruction. This is a very useful tool for circuit correctness verification.

Table 2, summarizes key feature of control logic, number of gates per controller. Due to Logic Friday limitation of using more than 16 variables, ARM encoded CPOG does not have DONE signal. Therefore, for reasonable comparison purposes two separate controllers were produced for SCENCO – modified encoding.

Remaining control circuits are provided in Appendix.

Table 2, number of controller gates for different encoded CPOGs

Type of Encoding	Number of Gates
SCENCO	58
SCENCO - M (without done signal)	60
SCENCO - M	86
ARM (without done signal)	153

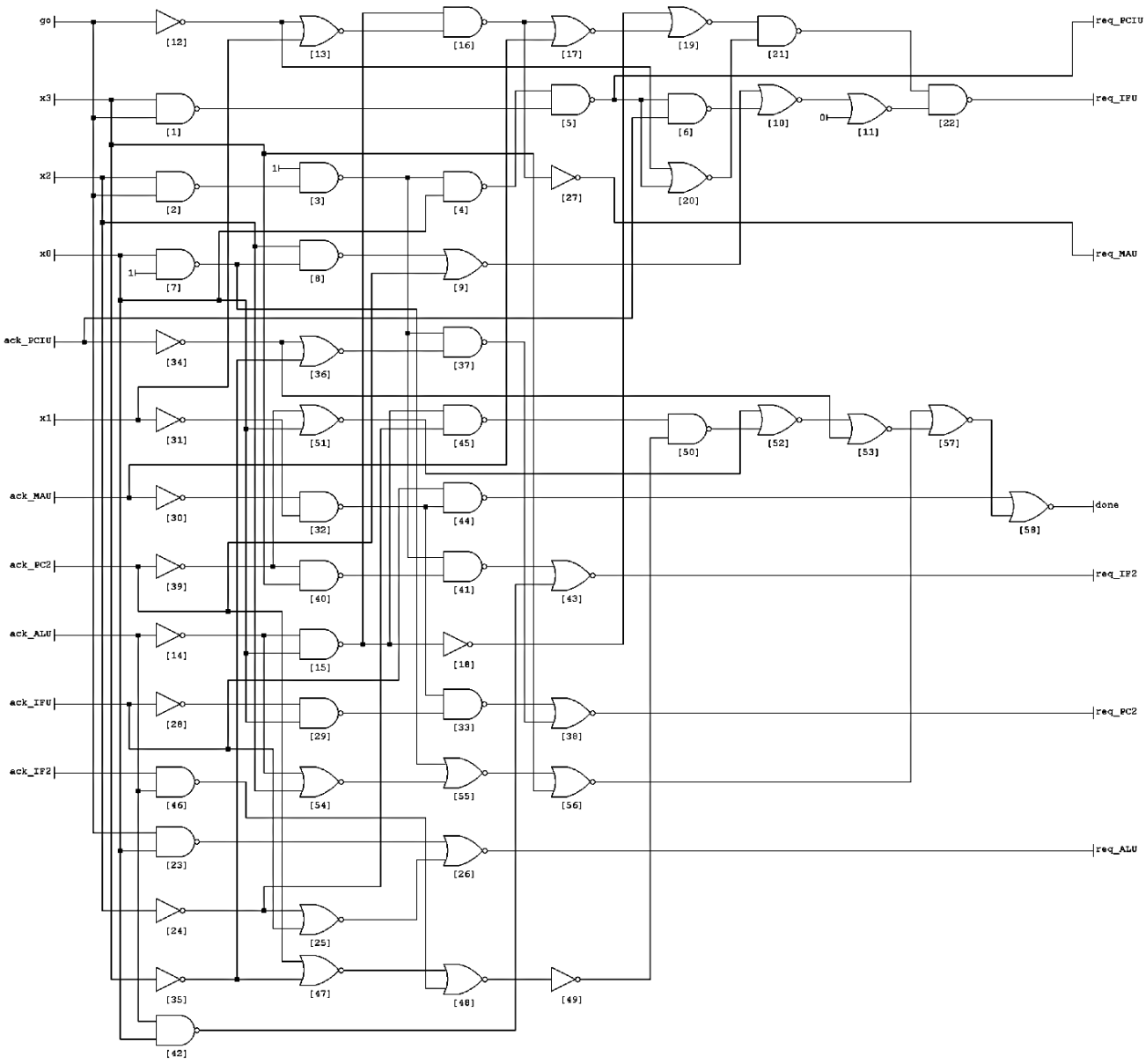


Figure 10, SCENCO Encoded CPOG Control Logic

5 Discussions

5.1 Specification of Instruction Set

Specification of Instruction Set procedure extracted 9 different partial orders, which covered in total 64 ARM Cortex M0+ instructions. Majority of covered instructions were arithmetical and logical and mostly belonged to Class 3 (see Section 4.1.3), which clustered third of all operations.

As discussed in Section 3.2.1 instruction addressing was one of the key factors for instruction grouping. Comparing extracted partial orders some similarities can be noticed regarding addressing differentiation. Immediate addressing instruction Classes 1, 2 and 5 contained Program Counter Increment Unit (PCIU) and/or Instruction Fetch Unit (IFU) more than once in their partial order. This can be explained by immediate addressing instruction operation. Immediate addressing instructions, contains two fetch cycles: constant fetching and next instruction fetching. Both these processes are denoted by events PCIU and IFU, so multiple units are required.

Likewise, similarity of partial orders among register type addressing classes is fairly clear, 3 out of 5 classes had concurrent processes, whereas one of the processes was next instruction fetch. Simultaneous next instruction fetch is allowed, since operands of these classes are stored general – purpose registers and ALU/MAU can access them directly. Remaining two classes cannot fetch next instruction concurrently, as they are branch instructions and have to wait for ALU/MAU event to complete.

Inspection of partial orders shows that only IFU was used in all partial orders. This will be further discussed, as it has effect on other results like complexity of CPOG and size of control logic.

However, not all ARM Cortex M0+ instructions were covered. In particularly hint type instructions, which are associated with interrupts, microprocessor state mode. These instruction required more insightful analyzes and due to project time constraints and very high level description of these instruction in [8] they were left out. It is reasonable to predict that effect these instructions could have had on size and complexity of CPOG and control circuit are

insignificant. However, additional instructions could have over complicated or even made impossible to derive SCENCO – M encoding, due to encoding constraint.

Results of specification of instruction set procedure can be compared to results obtained in. Few of derived POs matched POs extracted in that project for similar functionality instructions, thus confirming correctness of these POs. Referenced project used processor with much larger Instruction Set (257 instructions), therefore the number of obtained instructions groups (37 Instructions Groups) differ so significantly.

5.2 Instruction Group Encoding

Methodology used in this project requires encoding obtained partial orders. As discussed in Section 3.3 three type of encoding were derived. Firstly, optimal encoding was computed, which did not require much effort, since was completed by SCENCO plugin, which is integrated in Workcraft tool [15] [18]. Plugin key target is to produce optimal opcodes, which would result in lower complexity CPOG, which can be measured by equation 1.1. Discussions' Section 5.3 provided more detailed analyzes on how optimality of opcodes correlate with complexity of CPOG. Therefore, further analyses will concentrate on next two encoding derivation, which were done manually.

ARM and SCENCO encoded CPOG comparison was one of the project objectives and in order to complete this objective two new instruction encoding were derived. At this stage, initial concern was level of instruction decoding. SCENCO encoded CPOGs are only for partial instruction decoding, while ARM opcodes could specify individual instruction within instruction class. Hence, in order make fair comparison SCENCO encoding was transformed. Appendix A provides complete SCENCO – modified encoding scheme. Green highlighted bits indicate that SCENCO encoding was used, because number of bits was sufficient, red points out modified encoding. Appendix C provides ARM encoding, which for some instruction groups resulted in rather expressions than opcodes.

5.3 Conditional Partial Order Graphs

After instructions groups were encoded, project was preceded with CPOG generation. As discussed in Section 3.4 generation of CPOG was completed by automated and semi – automated methods.

Computationally obtained SCENCO encoded CPOG is shown in Section 4.3.1 Figure 6. Graph only had 18 literals, which compared to other produced CPOGs was a significantly lower number. Furthermore, 6 out of 11 arcs were unconditional and remaining vertices and arcs conditions could be computed with just 3 2 – input gates. It is rather interesting result, as almost whole instruction set can be modeled in such a compact form.

Considerably more complicated CPOGs were obtained with SCENCO – M and ARM encodings. Even though, SCENCO – M encoding (Figure 7) used the same number of opcodes bits (4 - bits), produced graph's vertices and arcs contained 70 literals, which was nearly 4 times more than SCENCO. Even more complicated CPOG was generated with ARM encodings (Figure 8), which had 374 literals.

Significant increase in complexity could be explained by transitive arc property. SCENCO plugin exploited this feature to reduce conditions, while still maintains correctness of partial orders. In example, projection of Class 9 is shown in Figure 6, bottom – right graph. It can be clearly noticed that projection of this PO does not entirely match with derived PO (see Section 4.1.9). This CPOG projection used indirect dependency or in other words, transitive arc, which helped to reduce PCIU → IFU arc condition, but did not violate a concept of PO. Similar process can be done with other POs transitive arcs. Note that not all projected POs had transitive arcs in the Figure 6, bottom – left; CPOG is shown under 1001 opcode, which did not have indirect dependency. SCENCO – M and ARM encoded CPOG generation process did not use arc condition reduction through transitive arcs, because synthesis was nearly done manually. Therefore, it is realistic to predict that if this reduction method was used in SCENCO – M and ARM CPOG synthesis lower complexity graphs would have been obtained.

Furthermore, selection of opcodes could have had some meaningful impact on SCENCO – M CPOG conditions complexity. Even though, this encoding was based on optimal plugin produced encoding, some necessary changes were made as explained in Section 3.3.3. In Appendix A red

highlighted opcodes indicate modifications from original encoding and very little can be done with these opcodes in order to obtain more optimal encoding. In example, Class 2 and 5 also Class 8 and 9 opcodes could have been swapped or even alternative variations explored. Though, it is unlikely that significant improvement would be accomplished.

In regards to ARM encoding, no optimization could be made as objective was to analyze ARM encoding, which was taken from [8]. Therefore, complexity of ARM CPOG was highly affected by increase in length of opcode, which used 9 bits, though only three classes used all them for others these lower bits were Don't Cares.

Comparison between ARM and SCENCO – M encoded CPOGs was not an easy task. Even though, literal count points out that SCENCO – M resulted in dramatically less complex control, level of decoding has to be considered more critically. Project attempted to be as reasonable as possible regarding decoding, thus both utilize partial instruction decoding. However, ARM encoding used more opcode bits, which in ARM decoder would be used to specify e.g. ALU control. Therefore, this comparison is a bit ambiguous and requires greater analyses. Possible solution and future work could be ALU control logic specification as well as creation of tool, which could produce optimal encoding for varied length opcodes. This further work could lead to comparison between control logic of SCENCO encoding and original ARM encodings.

5.4 Mapping

Project final procedure was CPOG mapping to Boolean equations, so that control logic could be produced. As explained in Section 3.5 request – acknowledgement protocol was used, which is captured by equations 2.1 and 2.2. Obtained equations were used to generate control circuit for each of CPOGs with Logic Friday tool.

Objective of this step was to observe how complexity of CPOGs correlates with a size of final logic circuit. This comparison can be done in terms of literals of CPOGs and number of gates in control circuits.

As suspected, more complex CPOGs with higher number of literals resulted in circuits with more gates. However, relation was not that linear. In example, optimally encoded SCENCO CPOG had 18 literals and produced control logic 58 gates. Modified SCENCO CPOG had nearly 4 times more literals in its conditions, 70, although size of this CPOG control logic did not

different so significantly and had 86 gates. Similar result correlation can be noticed and between SCENCO – M and ARM CPOGs.

Encoding of partial orders was essential part of this design flow and choosing an optimal encoding should be a priority. However, results indicated that relationship between complexity of CPOGs and number of gates in control circuit is not directly proportional. Nonetheless, finding optimal encoding in this design method should be main concern as results clearly indicate that optimal encoding results in smaller control logic.

5.5 Overview

Throughout results evaluation possibly obvious, but some personal important observations were made. As it was shown CPOG size and complexity directly correlates with size of mapped control logic and Section 2.2 discussed control logic importance on power consumption, latency. Hence, it was necessary to understand what criteria affect size and complexity of CPOG and thus control logic in this design flow. Results suggested that number of POs as well as PO encoding have the most influence on control logic size.

Firstly, it can be deduced that the number of POs depends on microprocessor instruction sets variety of functions and addressing modes. It was noticed that even if microprocessor has a lot of instructions in its instruction set, but has quite uniformed functionality and just few addressing modes, instructions tend to fall in the same instruction class and thus number of partial orders does not increase that significantly. However, instruction set functionality and addressing modes seem to be fundamental criteria, which affect size of CPOG.

Project results clearly indicated that PO encoding has great impact on complexity of CPOG. Optimal SCENCO encoding resulted in significantly lower number of literals compared to other encodings. However, important to note that other derivations were not completed computationally and did not reduce arc conditions with transitive arc property, which evidently had dramatic impact on arc conditions, thus number of literals.

Even, derivation process of partial orders is also an important criterion, which could affect results. In this project high level instruction descriptions were used to express instruction as partial orders, although there is no formal method how to derive or verify PO correctness.

Possible future work could also include creation formal methodology or even automated PO derivation software, which could use high level instruction set specifications to derive POs.

6 Conclusion

This project further explored introduced ISA design approach based on CPOG formalism and obtained some interesting results, which were analyzed and discussed. Project methodology was successfully followed and all steps completed. However, some steps did not have formal completion procedures; hence some new manual approaches to be developed.

Initial aim was to express instructions as partial orders and was only partially completed. Due to time constraints and not adequate high level descriptions of few instructions, they were left out. Nonetheless, in the project, 9 different scenarios or partial orders were obtained, which covered in total 64 ARM instructions. Derivation process used ARM provided technical manual, which contained high level instructions descriptions. Yet for some instructions level of description was not satisfactory. Therefore, in Section 5.5 some formal method, which could interpret high level instruction descriptions and perhaps convert them to partial orders, was suggested as possible future work.

Important step of methodology was scenario encoding. Throughout this step, few encodings were derived and analyzed. Results clearly signified importance of optimal encoding as SCENCO plugin computationally generated opcodes produced the most optimal CPOG with only 18 literals. Section 5.3 analyzed this interesting result and highlighted importance of transitive arcs, which helped to reduce arcs conditions. Manually produced SCENCO – M and ARM CPOGs did not use this property and resulted in significantly higher condition complexity, which eventually reflected on number of gates in their control logic. Some indirect relations between size of control logic and processors instruction set were discussed in Section 5.5

To my mind, project met its key technical objectives as some interesting results were obtained and evaluated. Moreover, personal goals were completed as well. I was introduced to completely new field so had to adapt and get familiar with new concepts and formalisms. I think that was successfully achieved.

7 References

[1] H. Iwai, "Roadmap for 22nm and beyond (Invited Paper)," *Microelectronic Engineering*, vol. 86, pp. 1520-1528, 7// 2009.

[2] J. Rabaey, *Low Power Design Essentials*: New York ; London : Springer 2009.

[3] R. K. Krishnamurthy and V. G. Oklobdzija, *High-performance energy-efficient microprocessor design*. Dordrecht, the Netherlands Springer, 2006.

[4] M. Yasir Qadri, H. S. Gujarathi, and K. D. McDonald-Maier, "Low Power Processor Architectures and Contemporary Techniques for Power Optimization," *JOURNAL OF COMPUTERS*, vol. 4, October 2009.

[5] A. Mokhov, A. Iliasov, D. Sokolov, M. Rykunov, A. Yakovlev, and A. Romanovsky, "Synthesis of Processor Instruction Sets from High-Level ISA Specifications," Computers, IEEE Transactions on Volume 63, Issue 6, pp. 1552 - 1566 , June 2014.

[6] R. J. Baron and L. Higbie, *Computer Architecture*: Reading, Mass. : Addison-Wesley Pub. Co. , 1992.

[7] ARM Ltd. (2012). *Cortex-M0+ Technical Reference Manual*. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0432c/index.html

[8] ARM Ltd. (2010). *ARMv6-M Architecture Reference Manual*. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0419c/index.html (registration required)

[9] A. Mokhov and A. Yakovlev, "Conditional Partial Order Graphs: Model, Synthesis, and Application," *Computers, IEEE Transactions on*, vol. 59, pp. 1480-1493, 2010.

[10] A. Mokhov, "Conditional Partial Order Graphs," PhD Thesis, School of Electrical & Electronic Engineering, Newcastle University, 2009.

[11] M. Rykunov, "Design of Asynchronous Microprocessor for Power Proportionality," PhD Thesis, School of Electrical & Electronic Engineering, Newcastle University, 2013.

[12] Andrey Mokhov, Maxim Rykunov, Danil Sokolov, Alex Yakovlev. "Design of Processors with Reconfigurable Microarchitecture". Journal of Low Power Electronics and Applications, Volume 4, Issue 1, pp. 36-43. 20 January 2014.

[13] F. Yuan and K. I. Eder, "A Generic Instruction Set Architecture Model in Event-B for Early Design Space Exploration," University of Bristol2009.

[14] A. Fox and M. Myreen, "A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture," in *Interactive Theorem Proving*. vol. 6172, M. Kaufmann and L. Paulson, Eds., ed: Springer Berlin Heidelberg, 2010, pp. 243-258.

[15] (2009). *The Workcraft framework homepage*. Available: http://workcraft.org

[16] A. Mokhov, A. Alekseyev, and A. Yakovlev, "Encoding of processor instruction sets with explicit concurrency control," *Computers & Digital Techniques, IET*, vol. 5, pp. 427-439, 2011.

[17] A. Mokhov, V. Khomenko, A. Alekseyev, and A. Yakovlev, "Algebra of Parameterised Graphs," in *Application of Concurrency to System Design (ACSD), 2012 12th International Conference on*, 2012, pp. 22-31.

[18] Ivan Poliakov, Danil Sokolov, Andrey Mokhov. "Workcraft: A static data flow structure editing, visualisation and analysis tool". Petri Nets and Other Models of Concurrency–ICATPN 2007, pp. 505-514, 2007

8 Appendices

8.1 Appendix A: SCENCO – modified Encoding Table

x15	x14	x13	x12	x11	x10	x9	x8	x7	x6	x5	x4	x3	x2	x1	x0	Instruction	G
1	1	1	0	RESERVED												B	1
0	1	0	0	0	0	0	RESERVED									ADD(imm.)	2
0	1	0	0	1	RESERVED											ADD(SP+imm.)	2
0	1	0	1	0												ADR	2
0	1	1	0	0												ASR (imm.)	2
0	1	1	0	1												CMP(imm.)	2
0	1	1	1	0												LSL(imm.)	2
0	1	0	1	1												LSR(imm.)	2
0	1	1	1	1												MOV(imm.)	2
0	1	0	0	0	0	1	X	X	X	RESERVED						RSB(imm.)	2
0	1	0	0	0	1	0	RESERVED									SUB(imm.)	2
0	1	0	0	0	1	1	X	X	RESERVED							SUB(SP-imm.)	2
1	1	0	1	1	1	0	0	0	0	RESERVED						ADC	3
1	1	0	1	0	0	0	RESERVED									ADD(reg.)	3
1	1	0	1	0	0	1	X	R	X	X	X	X	RESERVED			ADD(SP+reg.)	3
1	1	0	1	1	1	0	0	0	1	RESERVED						AND (reg.)	3
1	1	0	1	1	1	0	0	1	0							ASR(reg.)	3
1	1	0	1	1	1	0	1	0	0							BIC	3
1	1	0	1	1	1	0	1	1	0							CMN(reg.)	3
1	1	0	1	1	1	0	0	1	1							CMP(reg.)	3
1	1	0	1	1	1	0	1	0	1							EOR(reg.)	3
1	1	0	1	1	1	0	1	1	1							LSL(reg.)	3
1	1	0	1	1	0	1	0	0	0							LSR(reg.)	3
1	1	0	1	1	0	0	X	RESERVED								MOV(reg.)	3
1	1	0	1	1	0	1	0	0	1	RESERVED						MUL	3
1	1	0	1	1	0	1	0	1	0							MVN(reg.)	3
1	1	0	1	1	0	1	1	0	0							ORR(reg.)	3
1	1	0	1	1	0	1	1	1	0							REV	3
1	1	0	1	1	0	1	1	1	1							REV16	3
1	1	0	1	0	1	1	0	0	0							REVSH	3
1	1	0	1	0	1	1	0	0	1							ROR(reg.)	3
1	1	0	1	0	1	1	0	1	0							SBC(reg.)	3
1	1	0	1	1	1	1	RESERVED									SUB(reg.)	3
1	1	0	1	0	1	1	1	0	0	RESERVED						SXTB	3
1	1	0	1	0	1	1	0	1	1							SXTH	3
1	1	0	1	0	1	1	1	1	0							TST	3
1	1	0	1	0	1	1	1	0	1							UXTB	3
1	1	0	1	0	1	1	1	1	1							UXTH	3
1	1	0	0	1	X	X	X	X	RESERVED							BLX (reg.)	4

1	1	0	0	0	X	X	X	X	RESERVED	X	X	X	BX	4
0	0	0	0	0	RESERVED								LDR(imm.)	5
0	0	0	0	1									LDR(literal)	5
0	0	0	1	0									LDRB(imm.)	5
0	0	1	0	0									LDRH(imm.)	5
0	0	0	1	1									STR(imm.)	5
0	0	1	1	0									STRB(imm.)	5
0	0	1	1	1									STRH(imm.)	5
1	0	0	1	0	0	0	RESERVED						LDR(reg.)	6
1	0	0	1	0	0	1							LDRB(reg.)	6
1	0	0	1	0	1	0							LDRH(reg.)	6
1	0	0	1	1	0	0							LDRSB(reg.)	6
1	0	0	1	0	1	1							LDRSH(reg.)	6
1	0	0	1	1	0	1							STR(reg.)	6
1	0	0	1	1	1	0							STRB(reg.)	6
1	0	0	1	1	1	1							STRH(reg.)	6
1	0	1	0	0	RESERVED								LDM, LDMIA	7
1	0	1	0	1	X	X	RESERVED						POP	7
1	0	1	1	0	X	X							PUSH	7
1	0	1	1	1	RESERVED								STM, STMIA	7
1	1	1	1	X	X	X	X	RESERVED					POP	8
1	0	0	0	X	X	X	X	X	X	X	X	X	X	X	NOP	9

8.2 Appendix B: ARM Encoding Table

x15	x14	x13	x12	x11	x10	x9	x8	x7	x6	x5	x4	x3	x2	x1	x0	Instruction	Group
1	1	0	1													B	1
0	0	0	1	1	1	0										ADD(imm.)	2
1	0	1	0	1												ADD(SP+imm.)	2
1	0	1	0	0												ADR	2
0	0	0	1	0												ASR (imm.)	2
0	0	1	0	1												CMP(imm.)	2
0	0	0	0	0												LSL(imm.)	2
0	0	0	0	1												LSR(imm.)	2
0	0	1	0	0												MOV(imm.)	2
0	1	0	0	0	0	1	0	0	1							RSB(imm.)	2
0	0	0	1	1	1	1										SUB(imm.)	2
1	0	1	1	0	0	0	0	1								SUB(SP-imm.)	2
0	1	0	0	0	0	0	1	0	1							ADC	3
0	0	0	1	1	0	0										ADD(reg.)	3
0	1	0	0	0	1	0	0		1							ADD(SP+reg.)	3
0	1	0	0	0	0	0	0	0	0							AND (reg.)	3
0	1	0	0	0	0	0	1	0	0							ASR(reg.)	3
0	1	0	0	0	0	1	1	1	0							BIC	3
0	1	0	0	0	0	1	0	1	1							CMN(reg.)	3
0	1	0	0	0	0	1	0	1	0							CMP(reg.)	3
0	1	0	0	0	0	0	0	0	1							EOR(reg.)	3
0	1	0	0	0	0	0	0	1	0							LSL(reg.)	3
0	1	0	0	0	0	0	0	1	1							LSR(reg.)	3
0	1	0	0	0	1	1	0									MOV(reg.)	3
0	1	0	0	0	0	1	1	0	1							MUL	3
0	1	0	0	0	0	1	1	1	1							MVN(reg.)	3
0	1	0	0	0	0	1	1	0	0							ORR(reg.)	3
1	0	1	1	1	0	1	0	0	0							REV	3
1	0	1	1	1	0	1	0	0	1							REV16	3
1	0	1	1	1	0	1	0	1	1							REVSH	3
0	1	0	0	0	0	0	1	1	1							ROR(reg.)	3
0	1	0	0	0	0	0	1	1	0							SBC(reg.)	3
0	0	0	1	1	0	1										SUB(reg.)	3
1	0	1	1	0	0	1	0	0	1							SXTB	3
1	0	1	1	0	0	1	0	0	0							SXTH	3
0	1	0	0	0	0	1	0	0	0							TST	3
1	0	1	1	0	0	1	0	1	1							UXTB	3
1	0	1	1	0	0	1	0	1	0							UXTH	3

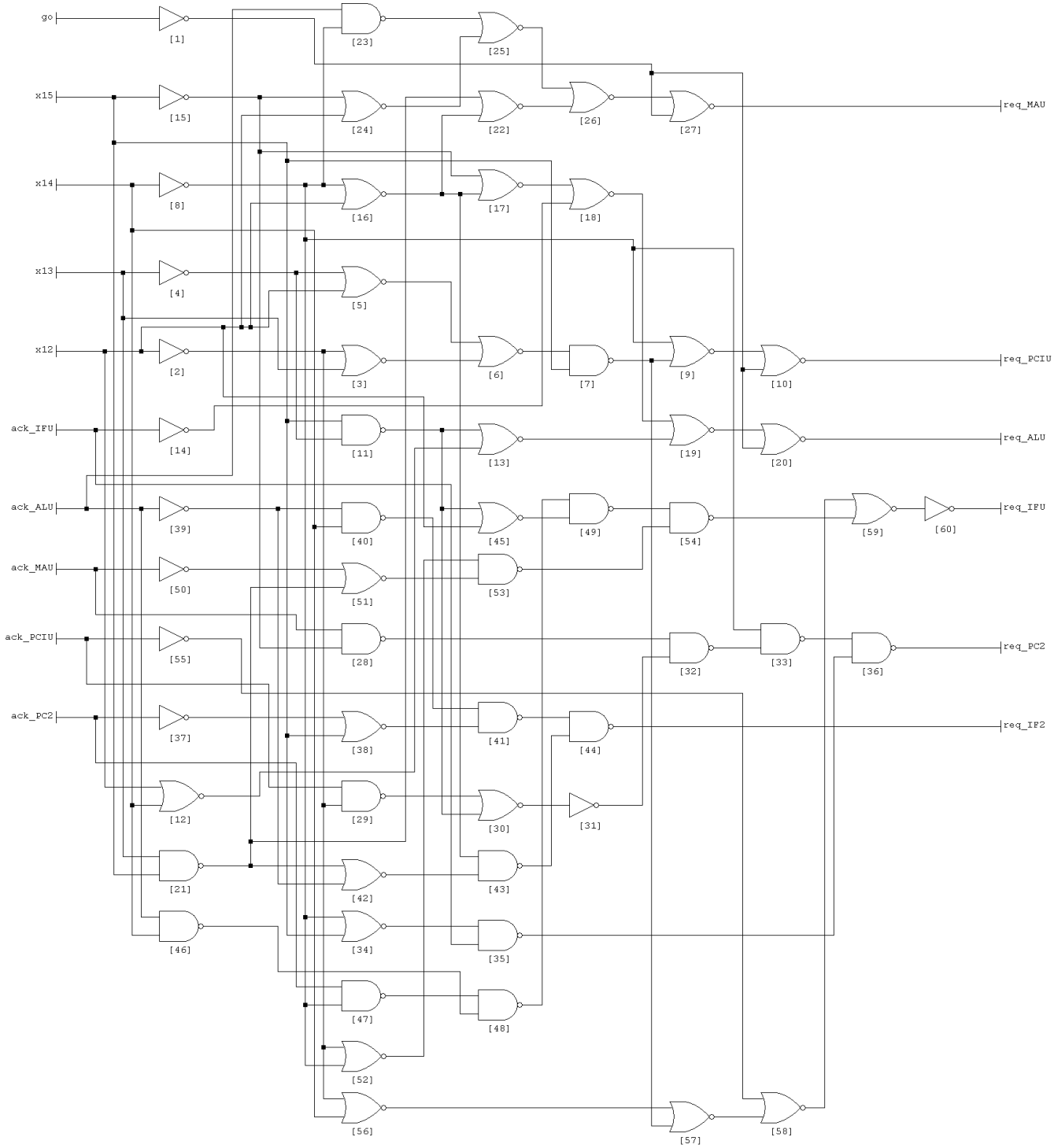
0	1	0	0	0	1	1	1	1								BLX (reg.)	4
0	1	0	0	0	1	1	1	0					0	0	0	BX	4
0	1	1	0	1												LDR(imm.)	5
0	1	0	0	1												LDR(literal)	5
0	1	1	1	1												LDRB(imm.)	5
1	0	0	0	1												LDRH(imm.)	5
0	1	1	0	0												STR(imm.)	5
0	1	1	1	0												STRB(imm.)	5
1	0	0	0	0												STRH(imm.)	5
0	1	0	1	1	0	0										LDR(reg.)	6
0	1	0	1	1	1	0										LDRB(reg.)	6
0	1	0	1	1	0	1										LDRH(reg.)	6
0	1	0	1	0	1	1										LDRSB(reg.)	6
0	1	0	1	1	1	1										LDRSH(reg.)	6
0	1	0	1	0	0	0										STR(reg.)	6
0	1	0	1	0	1	0										STRB(reg.)	6
0	1	0	1	0	0	1										STRH(reg.)	6
1	1	0	0	1												LDM, LDMIA	7
1	0	1	1	1	1	0										POP	7
1	0	1	1	0	1	0										PUSH	7
1	1	0	0	0												STM, STMIA	7
1	0	1	1	1	1	0	1									POP	8
1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	NOP	9

8.3 Appendix C: ARM Encodings

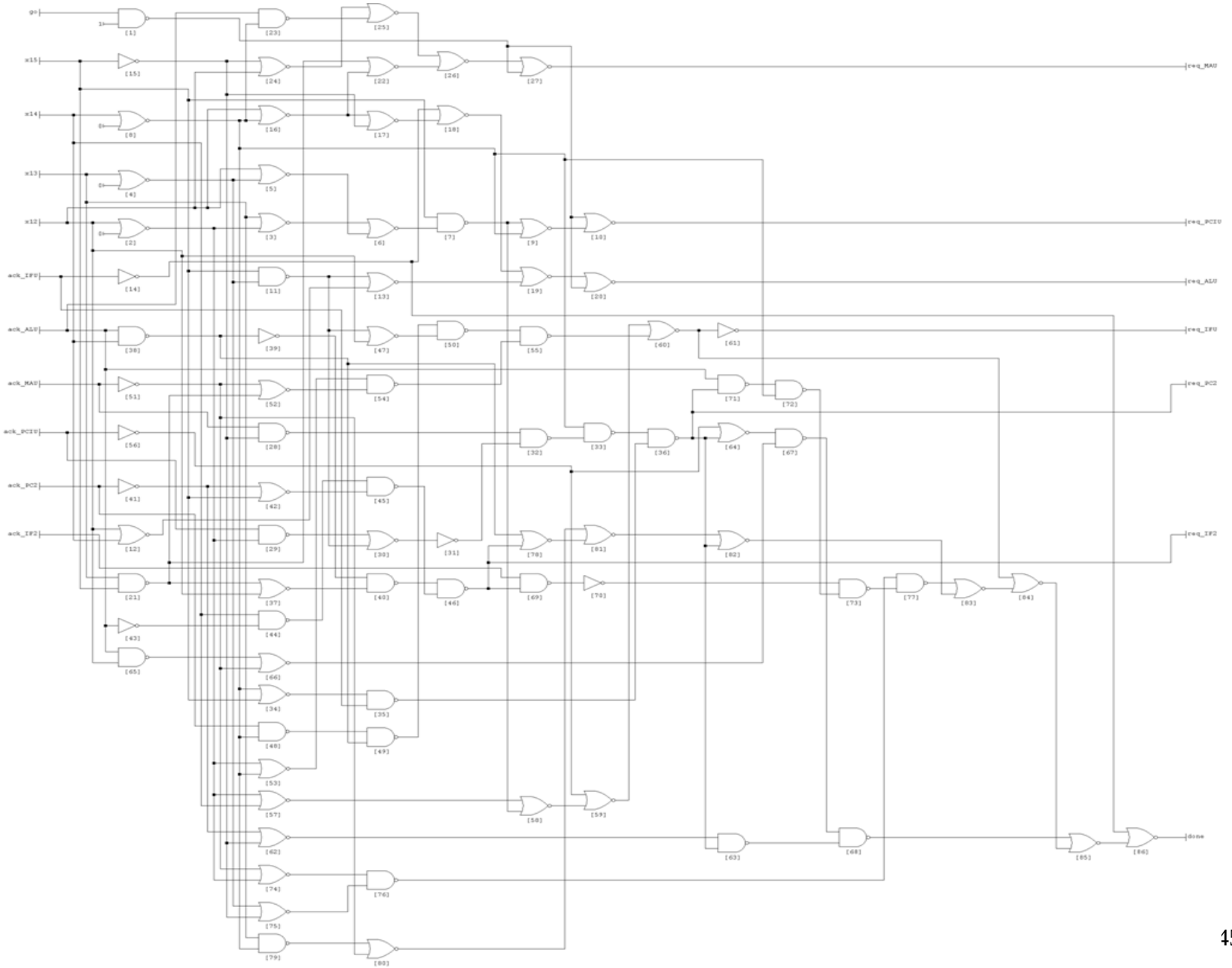
Note: x13' is logical not.

Class1	x15 x14 x13' x12
Class2	x15' x14' x12' + x14' x13 x12' + x15' x14' x13' x11' + x15' x14' x13' x10 + x15 x14' x13 x11' x10' x9' x8' x7 + x15' x13' x12' x11' x10' x9 x8' x7' x6
Class3	x15' x14' x13' x12 x11 x10' + x15' x14 x13' x12' x11' x10' x8 + x15 x14' x13 x12 x11' x10' x9 x8' + x15 x14' x13 x12 x10' x9 x8' x7' + x15 x14' x13 x12 x10' x9 x8' x6 + x15' x14 x13' x12' x11' x10' x9' + x15' x14 x13' x12' x11' x9 x8' x7 + x15' x14 x13' x12' x11' x10 x8' x6 + x15' x14 x13' x12' x11' x9 x8' x6';
Class4	x15' x14 x13' x12' x11' x10 x9 x8
Class5	x15' x14 x13 + x15 x14' x13' x12' + x15' x14 x12' x11
Class6	x15' x14 x13' x12
Class7	x15 x14 x13' x12' + x15 x14' x13 x12 x10 x9'
Class8	x15 x14' x13 x12 x11 x10 x9' x8
Class9	x15 x14' x13 x12 x11 x10 x9 x8 x7' x6'

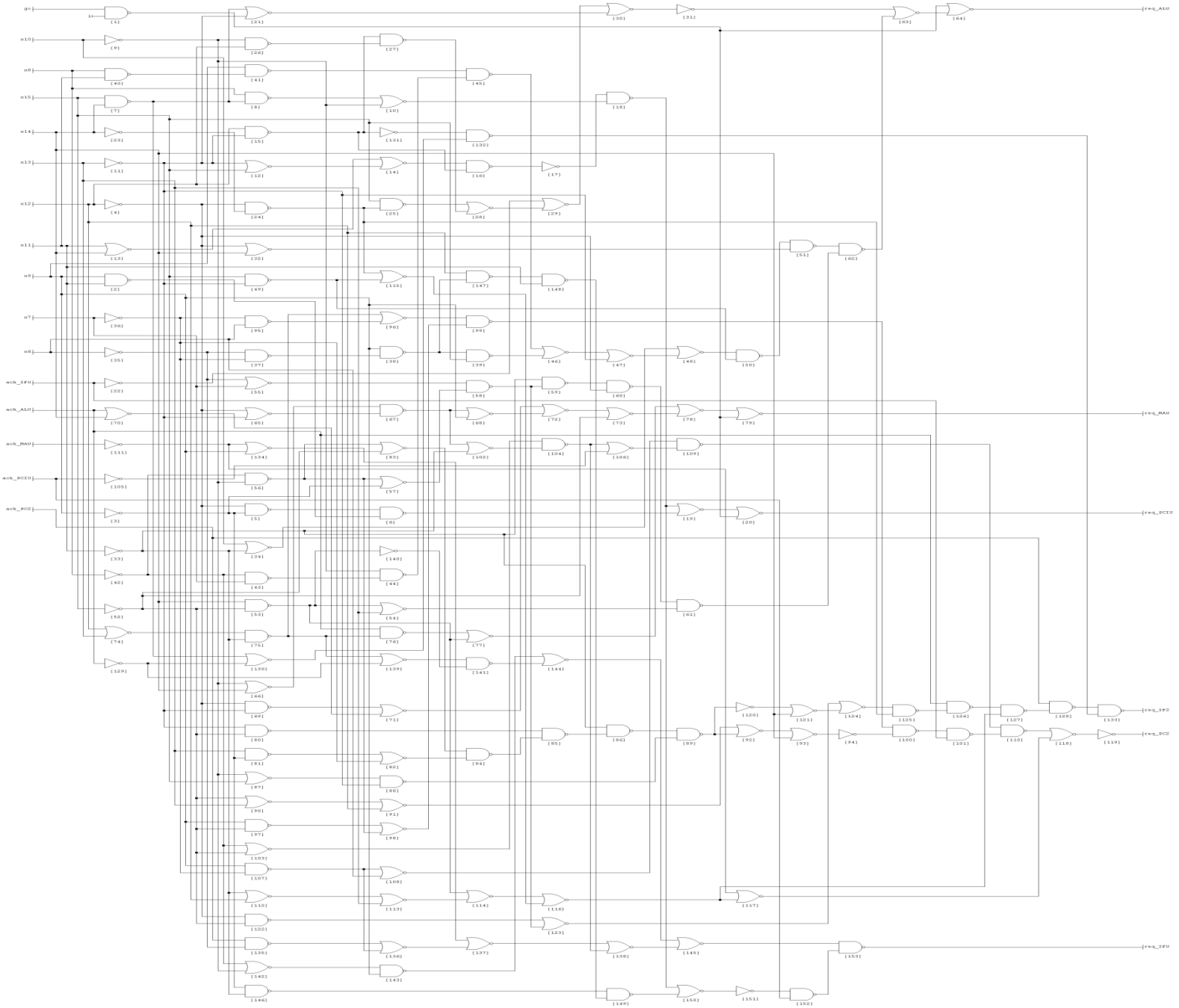
8.4 Appendix D: SCENCO – M Encoded CPOG Control Logic (without DONE)



8.5 Appendix E: SCENCO – M Encoded CPOG Control Logic (with DONE)



8.6 Appendix F: ARM Encoded CPOG Control Logic (without DONE)



8.7 Appendix G: Minimized SCENCO – M Controller Equations

REQ_PCIU = GO X15' + GO X14' + GO X13' X12 + GO X13 X12' ;

REQ_ALU = GO X15' ACK_IFU + GO X15 X14 X13' + GO X15 X13' X12 + GO X14 X12' ACK_IFU ;

REQ_MAU = GO X15 X14' X13 + GO X15 X13 X12 + GO X15' X14' ACK_ALU + GO X14' X12 ACK_ALU ;

REQ_PC2 = X15' X14 ACK_IFU + X15' X14' ACK_MAU + X15 X14' X13' X12' ACK_PCIU ;

REQ_IF2 = X15' X14' ACK_PC2 + X15' ACK_ALU ACK_PC2 + X15 X14 X13 X12' ACK_ALU ;

REQ_IFU = X15' ACK_PCIU + X13' X12 ACK_PCIU + X13 X12' ACK_PCIU + X15 X14 X13' X12' ACK_ALU + X15 X14 X13 X12 ACK_MAU + X15 X14' X13' X12' ACK_PC2 + X14' X12 ACK_PCIU ;

DONE = X15 X14 X13' X12' ACK_IFU ACK_ALU + X15 X14 X13 X12 ACK_IFU ACK_MAU + X15 X14 X13' ACK_IFU ACK_ALU ACK_PCIU + X15 X14' X13 ACK_IFU ACK_MAU ACK_PCIU + X15 X12 ACK_IFU ACK_ALU ACK_MAU ACK_PCIU + ACK_IFU ACK_ALU ACK_MAU ACK_PCIU ACK_PC2 ACK_IF2 + X15 X14' X13' X12' ACK_IFU ACK_PCIU ACK_PC2 + X15 X14

8.8 Appendix H: Minimized SCENCO – M Controller Equations (without DONE)

REQ_PCIU = GO X15' + GO X14' + GO X13' X12 + GO X13 X12' ;

REQ_ALU = GO X15' ACK_IFU + GO X15 X14 X13' + GO X15 X13' X12 + GO X14 X12' ACK_IFU ;

REQ_MAU = GO X15 X14' X13 + GO X15 X13 X12 + GO X15' X14' ACK_ALU + GO X14' X12 ACK_ALU ;

REQ_PC2 = X15' X14 ACK_IFU + X15' X14' ACK_MAU + X15 X14' X13' X12' ACK_PCIU ;

REQ_IF2 = X15' X14' ACK_PC2 + X15' ACK_ALU ACK_PC2 + X15 X14 X13 X12' ACK_ALU ;

REQ_IFU = X15' ACK_PCIU + X13' X12 ACK_PCIU + X13 X12' ACK_PCIU + X15 X14 X13' X12' ACK_ALU + X15 X14 X13 X12 ACK_MAU + X15 X14' X13' X12' ACK_PC2 + X14' X12 ACK_PCIU ;

8.9 Appendix I: Minimized ARM Controller Equations (without DONE)

REQ_PCIU = GO X10' + GO X8' + GO X15 X14 + GO X15' X13 + GO X13' X12 + GO X14' X11' + GO X11 X9 + GO X12' X9' ;

REQ_ALU = GO X15' ACK_IFU + GO X15 X14 X13 + GO X15' X14' X13 X12 + GO X15' X14 X13' X12 + GO X15 X14' X13' X12 + GO X10 X15' X14 X13' X11' + GO X8 X15' X14 X13' X11' + GO X15' X14 X13' X11' X9' + GO X15' X14 X13' X11' X7 + GO X15' X14 X13' X11' X6' + GO X10' X12 ACK_IFU + GO X13' X12 ACK_IFU + GO X14' X12' ACK_IFU + GO X10' X14' X12 X11 + GO X10' X8 X14' X13 X12 + GO X8' X14' X13 X12 X9 + GO X14' X13 X12 X11' X9 + GO X14' X13 X12 X9 X7 + GO X10' X14' X13 X12 X7' + GO X14' X13 X12 X9 X6 ;

REQ_MAU = GO X15 X14 X13' X12' + GO X15' X14 X13 ACK_ALU + GO X15' X14 X12 ACK_ALU + GO X15 X13' X12' ACK_ALU + GO X10 X15 X14' X13 X12 X9' + GO X14 X13' X12' X11 ACK_ALU ;

REQ_PC2 = X15' X14' X12' ACK_IFU + X14' X13 X12' ACK_IFU + X15' X14 X13 ACK_MAU + X10 X15' X14' X13' ACK_IFU + X15' X14' X13' X11' ACK_IFU + X15 X14' X13' X12' ACK_MAU + X15' X14 X12' X11 ACK_MAU + X10' X8' X15 X14' X13 X11' X9' X7 ACK_IFU + X10' X8' X15' X13' X12' X11' X9 X7' X6 ACK_IFU + X10 X8 X15 X14' X13 X12 X11 X9 X7' X6' ACK_PCIU ;

REQ_IF2 = X15' X14 X13 ACK_PC2 + X14' X12' ACK_ALU ACK_PC2 + X15 X14 X13' X12 ACK_ALU + X15 X14' X13' X12' ACK_PC2 + X15' X14 X12' X11 ACK_PC2 + X10 X15' X14' X13' ACK_ALU ACK_PC2 + X15' X14' X13' X11' ACK_ALU ACK_PC2 + X10' X8' X15' X12' X9 X7' X6 ACK_ALU ACK_PC2 + X10' X8' X15 X14' X13 X11' X9' X7 ACK_ALU ACK_PC2;

REQ_IFU = X10' ACK_PCIU + X8' ACK_PCIU + X10 X8 X15' X14 X13' X12' X11' X9 ACK_ALU + X10 X8 X15 X14' X13 X12 X11 X9' ACK_MAU + X10 X8 X15 X14' X13 X12 X11 X9 X7' X6' ACK_PC2 + X15 X14 ACK_PCIU + X15' X13 ACK_PCIU + X13' X12 ACK_PCIU + X12' X11 ACK_PCIU + X14' X11' ACK_PCIU + X11' X9' ACK_PCIU + X11 X9 X7 ACK_PCIU + X11 X9 X6 ACK_PCIU ;

8.10 Appendix J: ARM Encoded CPOG Conditions

```
ALU = x15’ + x14 x13 + x13' x12  + x14' x12'  + x12 x10'  + x12 x9 x11'  + x12 x9 x8'  +
x12 x9 x7  + x12 x9 x6;

PCIU = x10'  + x8' + x15 x14  + x15' x13  + x13' x12  + x14' x11'  + x11 x9  + x12' x9';

MAU = x15' x14 x13  + x15' x14 x12  + x15 x13' x12'  + x15 x14' x13 x12 x10 x9'  + x14
x13' x12' x11 ;

PCIU2 = x14' x12'  + x15' x14 x13  + x15' x12' x11  + x15' x14' x13' x11'  + x15' x14'
x13' x10  + x15' x12' x10' x9 x8' x7' x6 + x15 x14' x13 x11' x10' x9' x8' x7  + x15 x14'
x13 x11 x10 x9 x8 x7' x6';

IFU2 = x14' x12'  + x15' x14 x13  + x15' x12' x11  + x15 x14 x13' x12  + x15' x14' x13'
x11'  + x15' x14' x13' x10  + x15' x12' x10' x9 x8' x7' x6 + x15 x14' x13 x11' x10' x9'
x8' x7 ;

ALU-IFU = x15' x14 x13' x12' x11' x10 x9 x8;

ALU-IFU2 = x15' x14' x12'  + x14' x13 x12'  + x15 x14 x13' x12  + x15' x14' x13' x11'  +
x15' x14' x13' x10  + x15 x14' x13 x11' x10' x9' x8' x7  + x15' x13' x12' x11' x10' x9 x8'
x7' x6;

ALU-MAU = x15' x14 x13  + x15' x14 x12  + x15' x14 x11 + x15 x14' x13' x12' ;

IFU-ALU = x14' x12'  + x15' x14 x13  + x15' x12' x11  + x15 x14 x13' x12  + x15' x14' x13'
x11'  + x15' x14' x13' x10  + x15' x12' x10' x9 x8' x7' x6 + x15 x14' x13 x11' x10' x9'
x8' x7 ;

IFU-PC2 = x15' x14' x12'  + x14' x12' x13  + x15' x14' x13' x11'  + x15' x14' x13' x10  +
x15 x14' x13 x11' x10' x9' x8' x7  + x15' x12' x13' x11' x10' x9 x8' x7' x6;

PC2-IFU = x15 x14' x13 x12 x11 x10 x9 x8 x7' x6';

x14' x12'  + x15' x14 x13  + x15' x12' x11  + x15' x14' x13' x11'  + x15' x14' x13' x10  +
x15' x12' x10' x9 x8' x7' x6 + x15 x14' x13 x11' x10' x9' x8' x7 ;

PCIU-IFU = x10'  + x8'  + x15 x14  + x15' x13  + x13' x12  + x12' x11  + x14' x11'  + x11'
x9'  + x11 x9 x7  + x11 x9 x6;

PCIU-PC2 = x15 x14' x13 x12 x11 x10 x9 x8 x7' x6';
```