Hard- and Soft-Programmable, Multithreaded Micronet Architectures

D. K. Arvind and S. Zhong
Institute for Computing Systems Architecture, University of Edinburgh,
Mayfield Road, Edinburgh EH9 3JZ, Scotland.
Email: dka@dcs.ed.ac.uk

Abstract

It is our belief that future SOC devices will be overwhelmingly programmable, consisting of soft-programmable forms - as instruction set architectures, and hard-programmable forms - realised as field programmable logic. This paper explores an architectural structure which integrates them both in a micronet-based multithreaded architecture.
Keywords*: Asynchronous architectures, Micronets, Multithreading, Field Programmable Logic*

1 Introduction

This paper describes briefly the introduction of hard-programmability, in the form of field programmable logic, in a micronet-based multithreaded instruction set architecture [AR99].

Micronets are networks of entities which compute concurrently and communicate asynchronously. The entities are themselves networks in a recursive fashion, leading to a fractal approach to system design. In its purest form, the implementation of the micronet system will be free of any clocks, with the entities being implemented using self-timed design techniques. However, the micronet approach to system design does allow a mixture of both self-timed and synchronous implementations of the entities.

Micronet-based multithreaded architectures exploit concurrency at different levels: between threads, within threads and between instructions. Within each Thread Processing Unit (TPU), there are functional units which are either fixed or which can be configured. The design framework is able to map C application programs, through the intermediate representations and assembly code, to the fixed and configurable parts (see Figure 1). The architecture is modelled at different levels of abstraction: at the level of thread processing units with gross values for computation and communication latencies and power costs, down to the handshaking protocols of the micronet architecture which reflect more accurately the costs of speed and power consumption.

Quantifiable changes in the speed and power consumption can be affected by program optimisations at the higher levels in the design framework and partitioning the program between the soft- and hard-programmable entities.

2 The Architecture

The multi-threaded architecture consists of a network of Thread Processing Units (TPUs), Switches and Data and Instruction memories as shown in Figure 2. Each TPU (see Figure 3), in turn, is a network of *Functional* Microagents (FMs): instances of Arithmetic and Logic Units (ALU), a Reconfigurable Functional Unit (RFU) and a Memory Unit, which reads from, and writes to, the memory hierarchy via the Switches, and *Communication* Microagents (CMs), which handle the flow of information among the functional microagents and with the Register Bank. The CMs negotiate local data transfers by using a four-phase, request-acknowledge handshaking protocol. The Thread Issue Unit (TIU) fetches and decodes the instructions, after which the register requests are sent through X_FETCH_CM and Y_FETCH_CM, and the address of result register is sent through RQLK_CM. The instructions are issued by asserting the relevant control signals to the microagents, which in turn assert acknowledge signals to the CU after performing the micro-operations, and move on to the micro-operations of the following instructions. The RFU contains programmable logic blocks, lookup tables and carry computation logic. The configuration for the RFU operations can be loaded from memory at runtime, based on the instructions generated by the compiler. The X_BUS and Y_BUS are used to fetch operands for the function units, and the results are written back via the Z_BUS.

3 Operations

This section describes the operations when executing a typical sequence of instructions in a TPU involving the RFU and the fixed ALU (See Figures 4 to 7 for a graphical display of the operations in the simulator).

```
rop_1   $6,$6,$3,$4,$2
li      $7,1
add     $5,$5,$7
```

1. The TIU fetches the next instruction from the Instruction Buffer and the operations continue from step 4 onwards. If the instruction is not present, then it must be fetched from the memory hierarchy.

2. The TIU.TIU_CM handshakes with CU.C2TIU_CM, and the CU fetches a block of instructions from shared memory to fill the Instruction Buffer.

3. The CU.C2TIU_CM handshakes with TIU.TIU_CM to indicate that the requested instruction was now available in Instruction Buffer.

4. After decoding the instruction, the TIU issues the instruction to the corresponding functional unit. In this instance, the instruction "rop_1 $6,$6,$3,$4,$2" is a Reconfigurable Function Unit (RFU) operation, so the TIU activates the TIU.T2RFU_CM.

5. TIU.T2RFU_CM handshakes with RFU.RFU_CM, to send the instruction to the RFU.

6. Concurrently, the TIU initiates the fetching of the source register values to TIU.X_FETCH_CM and TIU.Y_FETCH_CM. In this case, four registers, $6,$3,$4,$2 have to be fetched, so the $6 request and $4 requests are sent to TIU.X_FETCH_CM, and $3 and $2 requsts are sent to TIU.Y_FETCH_CM.

7. The TIU CMs in turn handshake with the Register Bank's CMs, RB.X_ADDR_CM and RB.Y_ADDR_CM, and the Register Bank is prepared to fetch the appropriate data.

8. The RB.X2RB_CM and RB.Y2RB_CM handshakes with RFU.X2RFU_CM and RFU.Y2RFU_CM and provides the data via the X_BUS and the Y_BUS, respectively.

9. The TIU activates the TIU.RQLK_CM signal to initiate the locking of the destination register $6.

10. The TIU.RQLK_CM handshakes with the Register Bank's CM: RB.LK_CM, and the Regiser Bank now locks the register $6.

11. The RFU can, if required, move the configuration bitstream from Shared Memory and initialises the circuit for the bitstream.

12. RFU held until all the source value was avaliable on X_BUS and Y_BUS.

13. When the source values are available, the RFU evaluates the reconfigurable operation, rop_1, and the result is ready for the output port. The RFU initiates the prepared the ZRFU_CM for writing back to Register Bank.

14. RFU.ZRFU_CM handshakes with the the Register Bank's CM: RB.Z2RB_CM, and the result is written back to the register $6.

15. These steps are repeated when instructions "li" and "add" are issued to ALU.

4 Discussion

There is a trend towards architectures which combine processor cores with field programmable logic. For example, Xilinx's Virtex-II Pro has IBM's PowerPC processor cores (either one, two or four) immersed in a sea of FPGA, and with upto sixteen 80 Gbit/s transceiver channels. The A7 processor from Triscend integrates an ARM7TDMI core with FPGA fabric. Both follows the tendency of translating board-level system architectural styles directly onto silicon. This has resulted in designs consisting of monolithic processor cores and FPGA blocks, connected by on-chip buses. While this may be convenient for vendors of exisitng IPs, this style of architecture might not be most efficient way forward.
Instead we propose an asynchronous network of soft- and hard-programmable forms based on the micronet model which integrates the two forms more intimately. We have a a compiler environment for programming them and a simulation environment for modelling the new architectures to study the implications of issues such as the granularity and functionality of the TPU, interconnectivity and protocols for communicating between TPUs.

Acknowledgements

We would like to thank Xilinx Inc. San Jose, in particular Hamish Fallside, for funding this research. Shi Zhong is also funded through an ORS scholarship awarded by Universities UK.

References

[AR99] Arvind, D. K. and Rangaswami, R. 1999. "Asynchronous Multithreaded Processor Cores for System Level Integration", In Proc. IP'99, Nov. 1999, Edinburgh, pp 105-10, Miller Freeman.

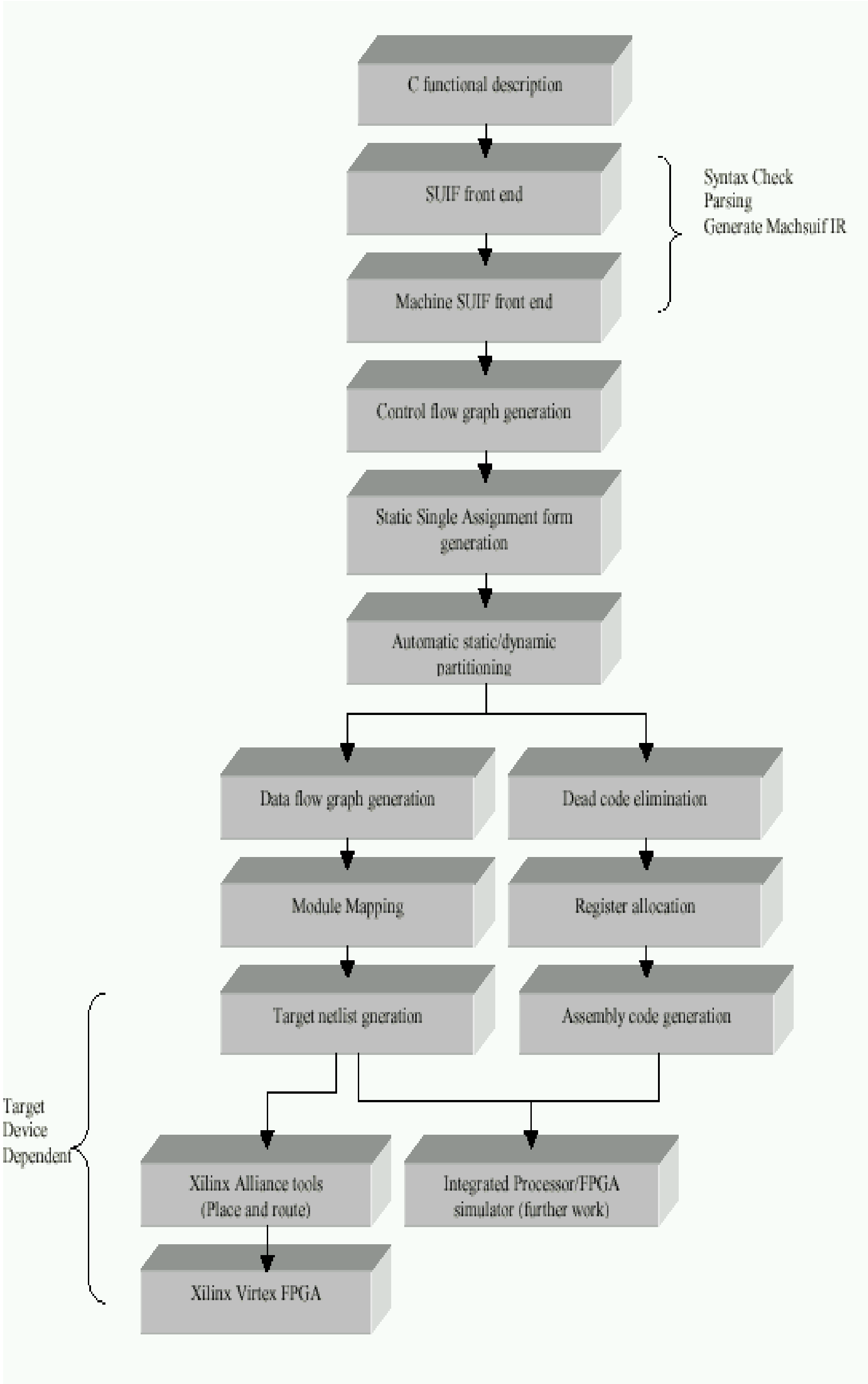

Figure 1. Compilation Flow

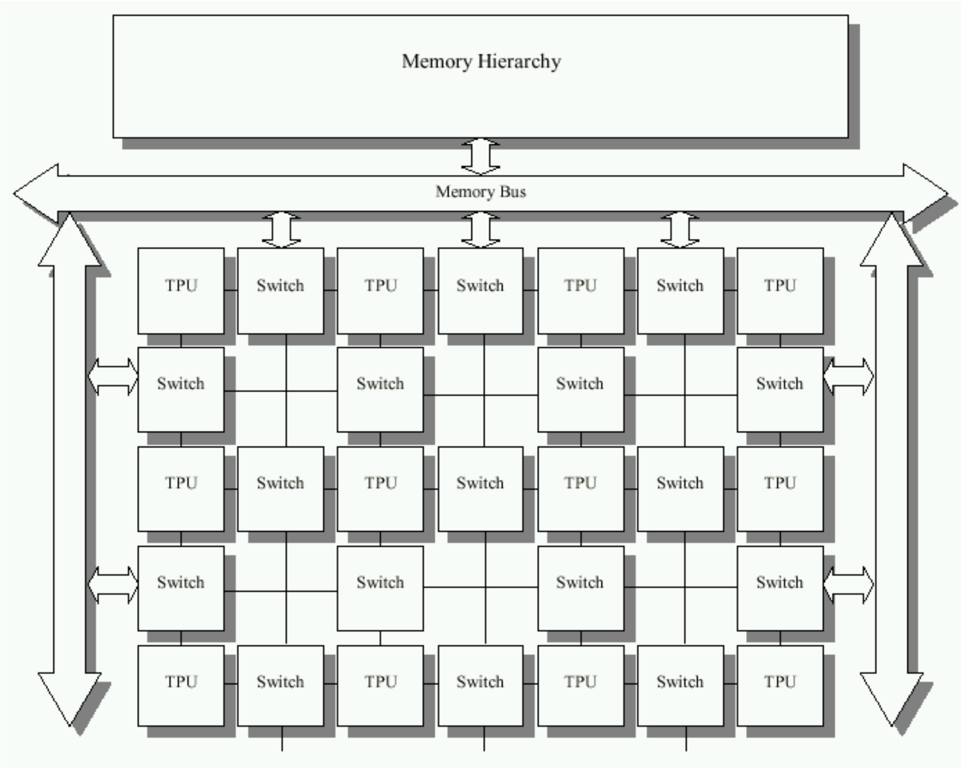

Figure 2. Multithreaded Architecture

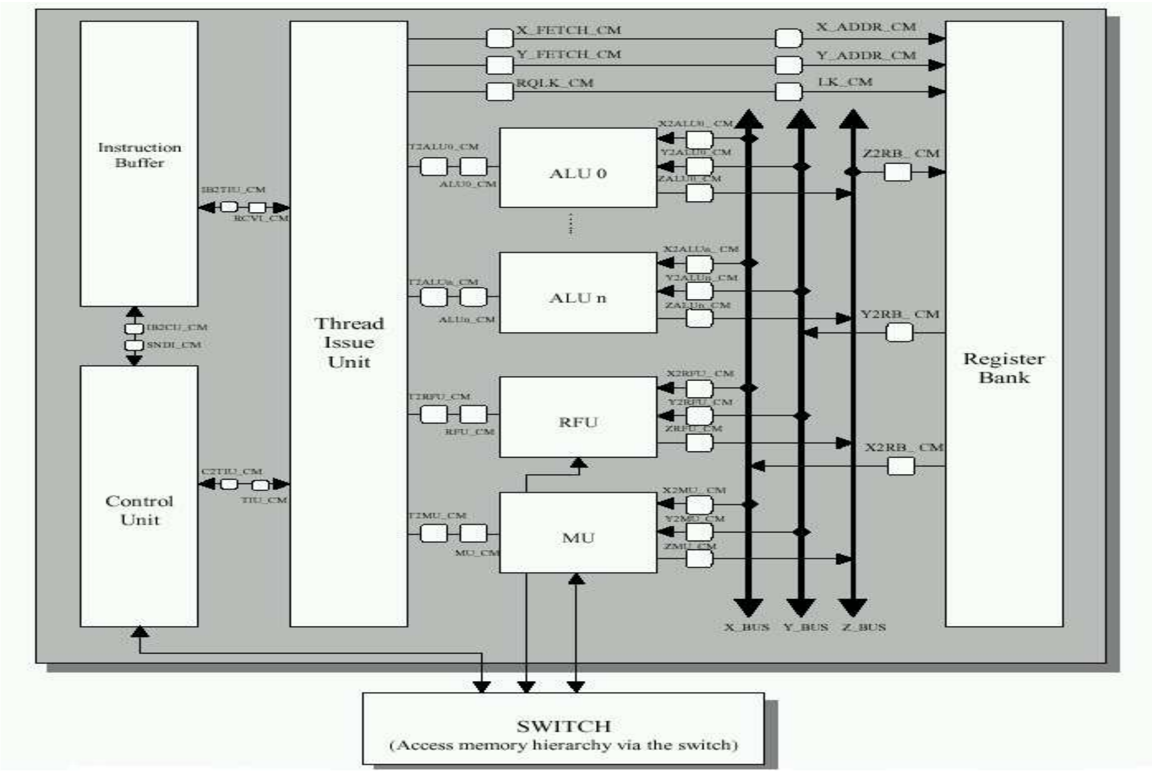

Figure 3. Thread Processing Unit

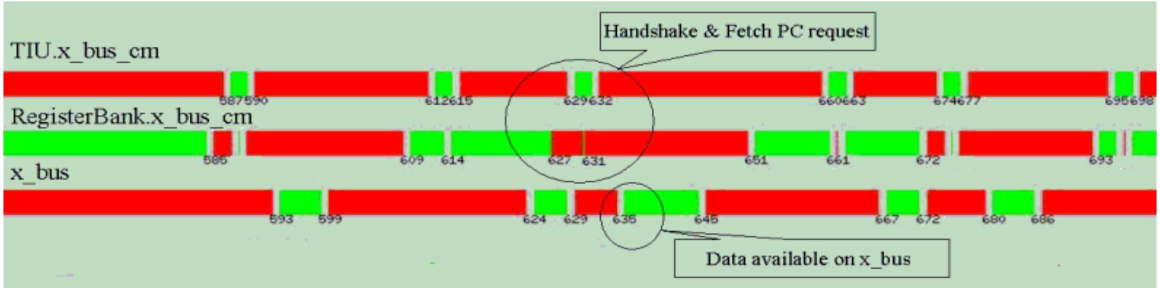

Figure 4. Fetching the PC

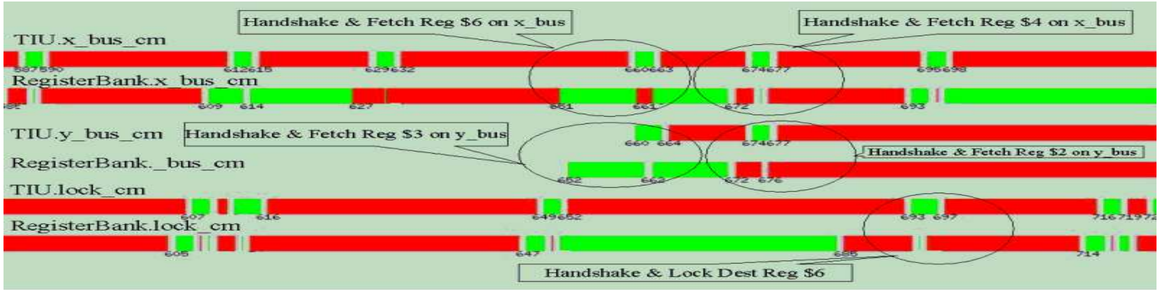

Figure 5. Fetching the X and Y operands

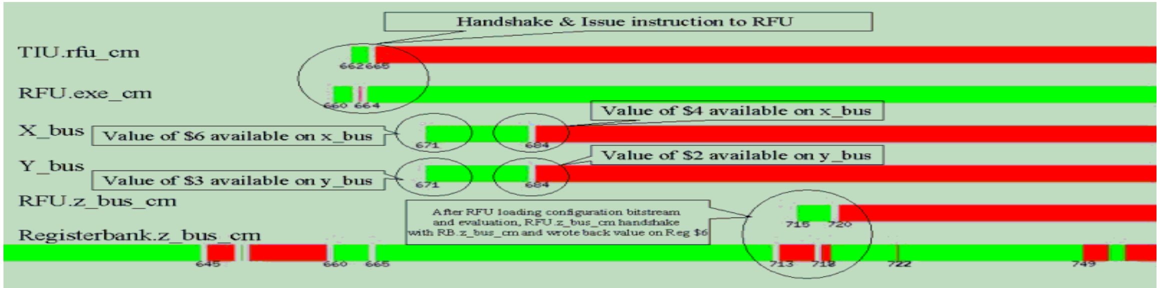

Figure 6. Operations in the Reconfigurable Logic Unit

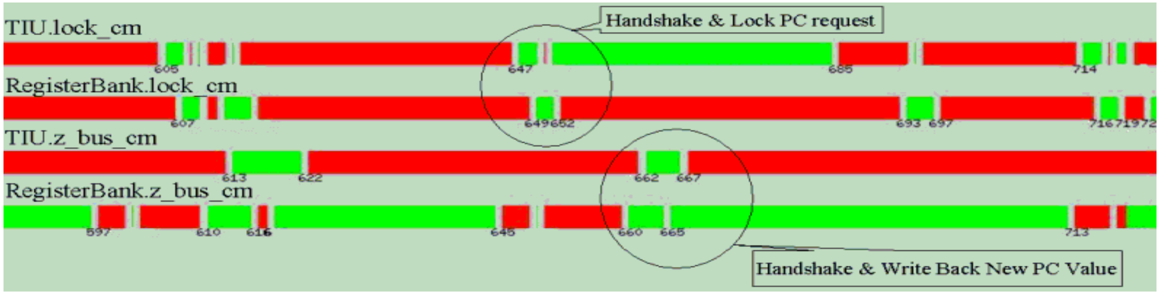

Figure 7. Updating the PC