The design of the control circuit for an asynchronous instruction prefetch unit using signal transition graphs

S.-H. Chung and S.B. Furber
Department of Computer Science, The University of Manchester,
Oxford Road, Manchester M13 9PL, UK.
{chung, sfurber}@cs.man.ac.uk

Abstract. AMULET3 is the third fully asynchronous implementation of the ARM architecture designed at the University of Manchester. It implements the most recent version of the ARM architecture (v4T), including the Thumb instruction set. Significant architectural changes from its predecessors help achieve higher performance without sacrificing the advantages of asynchronous design. One of these changes is to incorporate a highly parallel instruction prefetch unit.

This paper introduces the instruction prefetch unit in AMULET3, highlighting where speed-independent control circuits are implemented using signal transition graphs (STGs). In order to show how control circuits are implemented in the instruction prefetch unit of AMULET3, we presents several examples with relevant STGs and the synthesized circuit results.

1: Introduction

AMULET3 is the third asynchronous implementation of the ARM architecture [1] to be produced at the University of Manchester. Its predecessors, AMULET1 [2] and AMULET2 [3], were intended to demonstrate that asynchronous circuits of this complexity are feasible and practical; AMULET3 has been designed to be a commercially competitive macrocell. It is therefore required to deliver a performance similar to that of the contemporary synchronous ARM, the ARM9TDMI [4], and to implement the most recent version (v4T [5]) of the instruction set architecture including the 16-bit Thumb instruction set [6]. AMULET3 is being implemented in the same generic 0.35 µm 3 metal layer process as the ARM9TDMI. This implies a performance target of well over 100 MIPS (measured with Dhrystone 2.1), compared to the 40 MIPS delivered by AMULET2e on a 0.5 µm process.

Achieving this performance has necessitated a considerably different microarchitecture from the earlier AMULET processors. Most notable among the changes are the use of a Harvard architecture to increase memory bandwidth and the inclusion of a reorder buffer to handle data forwarding and memory faults. To cope with the former change, the instruction prefetch unit and the data interface are decoupled, whereas they were combined in the complex single address unit in AMULET2e. This paper is confined to describing asynchronous control circuit design in the instruction prefetch unit; readers having interests in other aspects of AMULET3 are referred to a related paper [11].

As in the previous AMULET processors, the architectural design is based on an asynchronous Micropipeline [7] structure using four-phase [8] control signals. All control circuits are developed on the basis of the speed-independent circuit assumption and this property is ensured using Petrify, an asynchronous synthesis tool [12]. Most of the control logic was specified with STGs [14][16], and each STG synthesized with Petrify. This paper does not show all of the control circuits implemented in the instruction prefetch unit but presents the general rules used to implement the control circuits including several examples.

In section 2, the specification of the instruction prefetch unit is given. This section is reproduced from a previous paper [11] with only minor changes and presents the context for the work reported here. The most recently updated information is added like the revised figure 1. The top-level STG definition of the instruction prefetch unit is shown in section 3. Section 4 introduces general rules used in the control circuit design. In sections 5 to 7, several design examples are shown to explain how real control circuits are made. In section 8 simulation results are presented to show the performance of the instruction prefetch unit in terms of speed. The conclusions are given in section 9.

2: Instruction prefetch unit

The instruction prefetch unit (figure 1) is responsible for generating addresses for the instruction memory which are sent via the Instruction Address Register (‘IAR’ in figure 1) [13]. The instruction prefetch unit has a highly parallel organization, speculatively computing the outcome of all scenarios in parallel and then selecting the appropriate course of action in the final multiplexer. Although such speculation causes some unnecessary activity (and therefore wastes power) it is necessary here to meet the required throughput.

Usually the output addresses form an ascending sequence and are provided by a simple loop containing an incrementer (‘INC’). When a branch occurs this loop may be interrupted asynchronously (via an arbiter) and loaded with a new address from the ALU. However, in AMULET3 several other functions are performed here also.

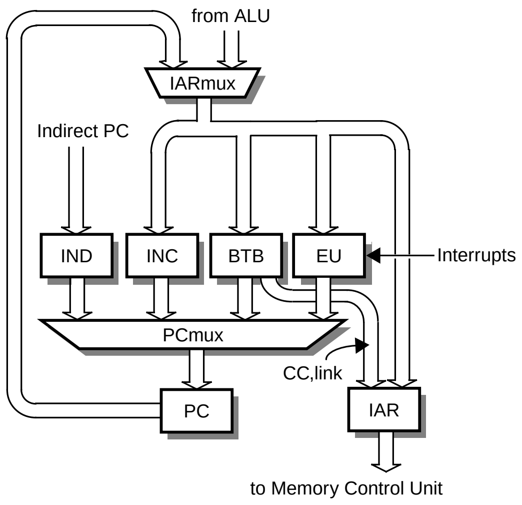

Figure 1: Instruction prefetch unit organization

2.1: Branch prediction

Branches disturb program flow and incur a considerable penalty in deeply pipelined systems. Sophisticated branch prediction mechanisms are now in use on state-of-the-art processors, but even a relatively simple branch predictor can significantly reduce pipeline disruption.

AMULET3 uses the same branch prediction mechanism as AMULET2 [9], namely a Branch Target Buffer (‘BTB’) which predicts a previously-taken invariant branch as ‘always taken’ until it is displaced from the BTB by a new entry. However there are two significant differences between the BTB in AMULET2 and that in AMULET3 [13].

The AMULET2 BTB records an address containing a branch instruction together with its target address. However, if a branch instruction address subsequently hits in the BTB the instruction is still fetched from memory and executed as it may be conditional and it may require a return address saving (if it is a BL – Branch-and-Link – instruction, used for procedure entry).

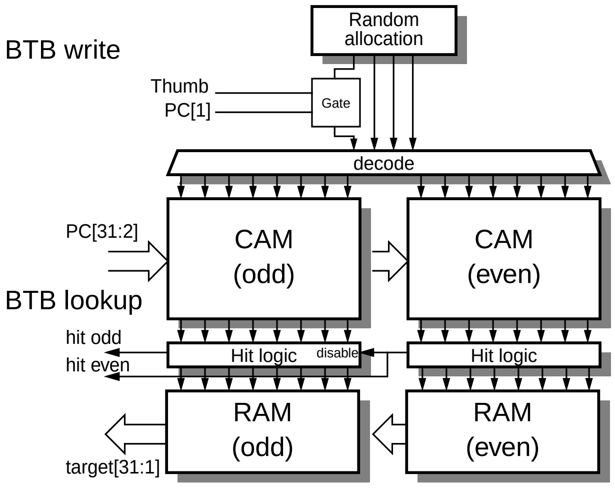

Figure 2: BTB CAM organization

The information which AMULET2 gets from memory when it fetches a predicted branch amounts to only five bits (four condition bits and the ‘L’ bit). In AMULET3 these five bits are stored in the BTB so that the instruction does not have to be fetched in repeat encounters and the instruction memory may be bypassed. As branches account for 10%-15% of ARM instructions [10], and the majority of these are cached in the BTB [9], this reduces the number of instruction fetches, yielding both a considerable power saving and a potential speed advantage (exploited automatically by the asynchronous pipeline).

The second BTB difference from AMULET2 is due to the presence of the Thumb decoder. ARM instructions are fetched as 32-bit words. When running Thumb code a choice must be made whether to fetch the 16-bit Thumb instructions individually or in pairs. As the speed and power consumption of a memory cycle is almost independent of the transfer size the decision was made to fetch Thumb instructions in pairs. However, as either or both of these instructions may be cached branches, the BTB must be able to cope with zero, one or two simultaneous hits.

This is achieved by splitting the BTB Content Addressable Memory (CAM) into two sections (see figure 2). In Thumb mode each section works with one half word of the instruction pair; any potential conflicts are resolved by taking the ‘even’ half word (the Thumb instruction at the lower address) prediction because this will always be first in the instruction sequence.

When running ARM code the two sections are merged. This allows the BTB to cache a mixture of ARM and Thumb branches simultaneously without compromising the number of usable entries in either case

2.2: Halting and interrupts

Most current CMOS technology dissipates very little power when not switching. This has been exploited in AMULET2e by causing the system to halt when no useful work can be performed, with demonstrable power-efficiency benefits [3].

Halting an asynchronous pipeline at any point soon causes the whole system to halt. Because there is no free running clock this reduces the number of transitions – and hence the power consumption – to near zero. In a synchronous system the clock oscillator could be stopped, but this is quite a complex procedure. The asynchronous system also recovers quickly (as there is no clock to restart). AMULET2e and AMULET3 exploit this by decoding a branch back to itself as a ‘Halt’ instruction and use this to stall the pipeline; this is fully compatible with much existing ARM code. The halt state is exited by the assertion of an enabled interrupt.

As alluded to above, the stall can occur anywhere within the pipeline. AMULET2, for example, stalls in the execution stage. AMULET3 adopts a somewhat cleaner model by stalling at the prefetch stage. This means that the processor restarts with an empty pipeline which provides the fastest possible response, any ‘rubbish’ being cleared out at halt entry.

The interrupt signals are fed into the prefetch unit rather than the instruction decoder. This rather unusual feature provides both a clean interrupt model and a low interrupt latency. When an (enabled) interrupt is asserted it is arbitrated into the prefetch cycle and treated much like a predicted ‘BL’. The interrupt ‘hijacks’ an instruction address, bypasses the memory, and proceeds down the execution path to save the return address. The PC is loaded with the address of the service routine (which is a constant, generated in the Exception Unit, ‘EU’ in figure 1) in parallel and the prefetching of this code begins immediately.

A consequence of this approach is that the prefetch unit must store an up-to-date copy of the interrupt enable status. One danger is that this may be out of date because an operation already prefetched may change it. Another, related, problem is that the hijacked address may be in the ‘shadow’ of a branch and the interrupt may try to save an incorrect return address.

Both these problems are solved by treating control instructions (such as enabling/disabling interrupts) as branches, and branches as potential control instructions. This is not particularly onerous because almost all instructions which can alter these flags (e.g. software interrupts, return from interrupts, etc.) also cause flow changes anyway. If an interrupt has occurred in a branch shadow it will be discarded in the same way as any other erroneously prefetched instruction. Concurrently, the branch will reach the prefetch unit, re-enable interrupts, and immediately cause the interrupt entry mechanism to repeat, this time saving the branch target as the return address.

2.3: Indirect branches

ARM programs often load the Program Counter (PC) directly from memory as part of a subroutine return (and, less frequently, as a result of a jump table lookup). Typically a subroutine return restores the PC together with a set of working registers using a load multiple (LDM) instruction. The load ordering is such that the lowest numbered register is loaded first, and thus the PC (R15) is loaded last. This delays the start of instruction fetching from the return address and compromises performance.

AMULET3 incorporates an optimization which exploits the separate instruction memory port. The execution unit passes the load address of the PC value back to the prefetch unit via the branch address path in parallel with initiating the other register transfers in the data interface. This ‘branch’ terminates prefetching from the redundant instruction stream and prompts a single read cycle which fetches the new PC. This is then returned to the prefetch unit (via ‘IND’ in figure 1) where instruction fetching resumes. With a typical subroutine return much of this should happen whilst the data transfers are proceeding and so the new instructions should be available before the LDM has completed.

Note that this feature imposes a significant constraint on the memory designer: the instruction and data memories must be coherent because a PC value is stored via the data port and then read via the instruction port. The first AMULET3-based system has a unified memory which is dual-ported to give independent instruction and data ports. Coherence is therefore not an issue here.

3: Control path overview

The control path in the instruction prefetch unit can be viewed as a large arbitration block including a fork and join connection as shown in figure 3. Note that figure 3 is a very simplified STG of the instruction prefetch unit from a high-level viewpoint. Furthermore, this STG uses a more abstract labelled Petri net when compared to the original STG definition. We labelled each transition according to a logical function in the instruction prefetch unit, not a detailed control signal transition. However, other STGs used in this paper are based on the original STG definition.

Arbitration occurs between two requests; one is PCreq and the other is BRAreq (see figure 3). The fork is situated in the IARmux and the join in the PCmux (see figure 1 and figure 3).

Each dot in figure 3 represents an initial token when the circuit is in the reset state and each bar represents the behaviour of each block in figure 1.

The name of each transition is the same as the corresponding block name in figure 1 except the PCreq, BRAreq, PCaccept, BRAaccept and BRAdone transitions. The PCreq transition represents the recirculating request from the incremented PC value which is generated entirely within the instruction prefetch unit. The BRAreq transition represents a request for a new value (a branch target PC) from outside the instruction prefetch unit. This new value will replace the old value in the internal recirculating loop.

The box labelled ‘ARB’ represents an asynchronous arbiter. The token in place A enables either the PCaccept or the BRAaccept transition to fire (assuming their respective requests are active), but only one at a time. When both requests fire at nearly the same time the arbiter ensures that one, and only one, is granted.

Assume that the circuit has just been reset. The STG in figure 3 has two initial requests; one is from the PC block and the other is from the ALU block (see figure 1). The former is PCreq and the latter is BRAreq in figure 3.

When the arbitration takes place (in ARB) either the PCaccept or the BRAaccept transition may fire. No matter which one fires, the other can’t fire until the token is back in place A. That is, arbitration starts when place A sends a token to one of transitions; PCaccept or BRAaccept, and ends when a token is put in place F.

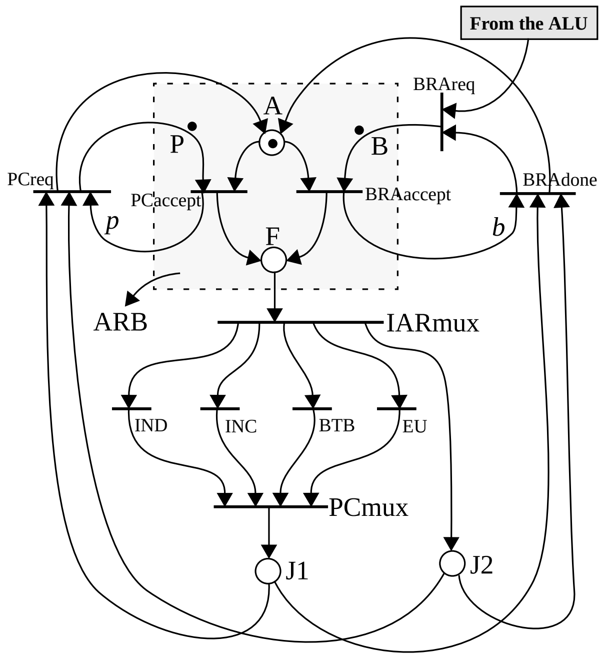

Figure 3: Top-level STG of the instruction prefetch unit

Next, place F gets a token from one of the input transitions, PCaccept or BRAaccept. The token enables to the transition named IARmux in figure 3, which represents the instruction prefetch unit receiving a new address either from the PC block inside the instruction prefetch unit or from the ALU block outside the instruction prefetch unit, depending on the arbitration result. This IARmux transition fires tokens to each block in the instruction prefetch unit such as the IND, the INC, the BTB and the EU (see figure 1) and to the memory control unit in AMULET3. At the moment the IARmux fires, the control path in the instruction prefetch unit forks.

The IND, INC, BTB and EU transitions in figure 3 represent the behaviour of each block in the instruction prefetch unit and these transitions are described in the following sections in detail. After these transitions fire, four tokens are joined at the transition named PCmux. The behaviour of the PCmux is to select one of addresses from the IND, the INC, the BTB and the EU for the next program counter and then store this address in the program counter address flip/flop.

The places named J1 and J2 play the role of feeding tokens to either the PCreq transition or the BRAdone transition. Whether tokens go to PCreq or BRAdone is decided by the arbitration that happened in the ARB. If BRAaccept fires, the arrow *b* feeds a token to BRAdone and if PCaccept fires, the arrow *p* provides a token to PCreq. When either PCreq or BRAdone fires, a token is put back in place A. This means that all of the arbitration behaviour is finished in the instruction prefetch unit and a new arbitration can occur between PCreq and BRAreq. Note that the BRAreq transition can fire only when the ALU sends a new branch PC to the instruction prefetch unit.

4: 4-phase control design strategy

Since the 4-phase broad protocol is used for the AMULET3 design, there is a return-to-zero phase to recover the control signal as shown in figure 4. This phase could be a disadvantage in terms of speed since the phase does nothing but the return-to-zero. When a designer tries to implement dynamic datapath circuits, however, this return-to-zero phase could be usefully employed for precharging the circuits.

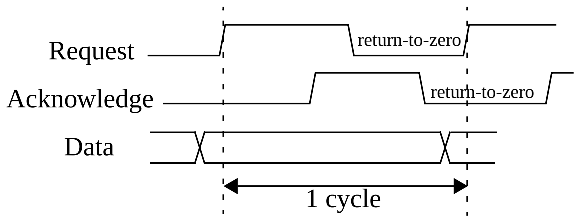

Figure 4: 4-phase broad protocol

If we view the request signal (rising edge) in figure 4 as the arrow from place F in figure 3, and the acknowledge signal (falling edge) in figure 4 as the arrow to place A either from PCreq in figure 3 or from BRAdone in figure 3, one cycle time in the instruction prefetch unit is defined from the rising edge of the request signal in figure 4 to the next rising edge of the same signal. The AMULET3 specification requires this cycle time to be less than 7 nanoseconds and this guarantees that the final chip speed will be well over 100 MIPS which is comparable with the latest synchronous ARM microprocessors.

The strategy used to design control circuits depends on the manner in which the datapath circuits are implemented; static or dynamic. When static datapath circuits are used, the control circuit design is rather straightforward since the AMULET3 design adopts the bundled data scheme, where completion signals are generated by mimicking the worst-case time consumed in the datapath. That is, delay elements are inserted into the path of the request signal and the resulting delayed request signal provides the request signal to the next block as shown in figure 5.

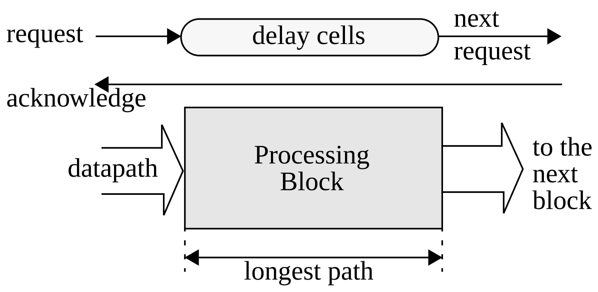

Figure 5: Control scheme for a datapath

What is needed is to detect the longest path (in terms of time) of the processing block and to put delay cells (the same delay time as the longest path plus a silicon process margin) on the request signal.

When dynamic datapath circuits are used, the return-to-zero phase could be used for precharging the dynamic datapath circuits. During the evaluation phase, the same technique is used for the request signal; putting delay cells to mimic the longest path (in terms of time) of the processing block. When the return-to-zero phase starts on the request signal (at the falling edge), this could be a trigger for the precharge phase of the dynamic datapath circuits (see scheme 2 in figure 6).

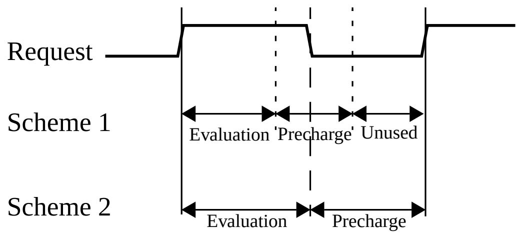

Figure 6: Dynamic circuit control timing

The EU block in figure 1 is comprised only of static datapath circuits and therefore the return-to-zero phase of the request signal performs no useful work. The BTB and INC blocks consist of dynamic datapath circuits. For these blocks the return-to-zero phase could be used to perform the precharge phase.

However, even though dynamic datapath circuits are used, the return-to-zero phase is of no use if a fully decoupled control circuit [8] is used to give a highly concurrent circuit operation. In general higher concurrency might be expected to make a faster circuit, since as soon as the request signal is transferred to the processing block, the acknowledge signal of the block is issued back to the previous block (without waiting for the acknowledge signal from the next block) and the next request signal is transferred to the next block concurrently.

Does higher concurrency always mean a faster control circuit for dynamic datapath circuits? The answer is no in our design.

If we use fully decoupled control circuits for the INC and BTB blocks, we can use control timing following scheme 1 in figure 6. This looks faster than scheme 2 in figure 6. However, there are two disadvantages in terms of the speed and size of circuits.

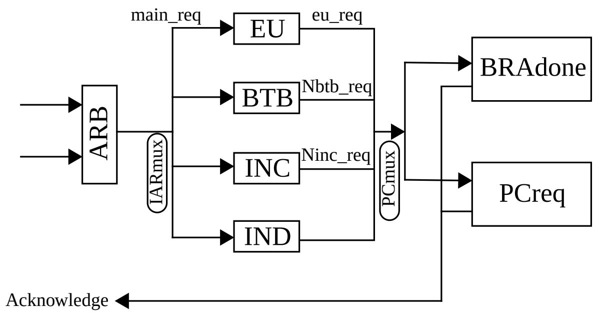

Figure 7: Request and acknowledge signals in the instruction prefetch unit

Let's consider the speed issue first. As shown in figure 7, the INC and BTB transitions occur in the middle of the instruction prefetch unit between the IARmux and the PCmux transitions and the cycle of the instruction prefetch unit is finished when either the PCreq transition or the BRAdone transition sends an acknowledge signal (falling edge) back to the arbiter. The BRAdone and PCreq transitions are implemented at the IAR and PC blocks in figure 1 respectively.

The cycle time in the instruction prefetch unit depends on how fast the request signal can reach the BRAdone and PCreq transitions. If we use control scheme 1 in figure 6 for the INC and BTB blocks, a more complex circuit is needed to implement a fully decoupled circuit, since the fully decoupled circuit must retain some control state. The state is needed for two possible cases. One is when the precharge phase is finished before a new request rising edge arrives at the control circuit and the other is the reverse case. In the former case, the information that the precharge phase is finished must be memorized until the new request rising edge enters the control circuit and vice versa in the latter case. The more complex circuit will reduce the speed of forwarding the request signal from the main_req to the BRAdone and PCreq transitions in figure 7. Since the INC and BTB blocks are in the middle of the instruction prefetch unit, there is no advantage if the rest of blocks are not finished, even though these blocks finish their functions.

Secondly, consider the size issue. The PCmux transition will happen after the EU, BTB, INC and IND transitions are finished as shown in figure 3. In order to use a fully decoupled control circuit, we need storage elements in the datapath circuits at the ends of the BTB and the INC blocks. Otherwise data could change when the PCmux transition is being activated. Therefore we must add a large number of latches or flip-flops (in our case 31x2 latches). As was pointed out, we need more complex control circuits and this will increase the number of transistors in the control circuit as well.

So we use scheme 2 in figure 6 for the BTB and the INC dynamic datapath circuits. Since the precharge time is always smaller than the return-to-zero phase time of the request signal in our design, there is no increase on the cycle time of the instruction prefetch unit. In cases where the return-to-zero phase time is smaller than the precharge time, a fully decoupled control circuits could be considered.

5: The design of the INC block

The behaviour of the INC block is shown in figure 8, where the main_req and the Ninc_req signals are as same as in figure 7. The Nprech signal controls the precharge phase of the INC block and the Ncomp signal indicates the evaluation phase of the INC block is finished (completion signal).

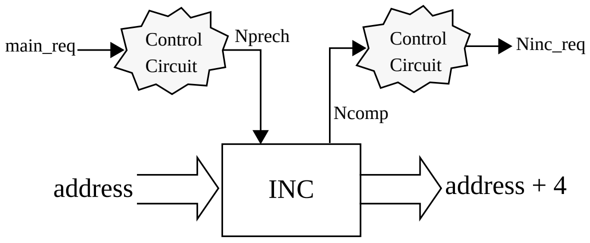

Figure 8: INC control circuit

$$\begin{tikzcd}
\text{main\_req+} \arrow[r] & \text{Nprech+} \arrow[r] & \text{Ncomp-} \arrow[r] & \text{Ninc\_req-} \arrow[d] \\
 & \text{Ncomp+} \arrow[u, "\bullet"'] & \text{Nprech-} \arrow[l] & \text{main\_req-} \arrow[l] \arrow[d] \\
 & & & \text{Ninc\_req+} \arrow[uulll, bend left, "\bullet"]
\end{tikzcd}$$

Figure 9: STG for the INC block

Since we choose to use scheme 2 in figure 6, we have a serial forward connection from the main_req to the Ninc_req as shown in the first line of figure 9. Note that Nprech, Ncomp and Ninc_req are active low signals and main_req is an active high signal. However, we can use a decoupling technique for the precharge phase as shown in the second and third lines of figure 9. The second line shows that the precharge phase starts immediately after main_req- (triggering the precharge phase of the INC datapath circuit) enters the INC block and the third line shows that the return-to-zero phase of the request signal is transferred to the next block (PCmux in figure 7) at the same moment as the precharge phase starts. (This is called semi-decoupling [8].) Therefore we can reduce the time needed for the return-to-zero phase of the request signal.

The synthesized control circuit of the STG in figure 9 is shown in figure 10, where a C-gate is used.

The C-gate is widely used in asynchronous design. If we have a C-gate as in figure 11, the behaviour is as follows. When the A, B and C inputs are high, the output O will change to high. In this case the input D does not affect the change of the output O. When the B, C and D inputs are low, the output O will change to low. The input A does not have an impact on the output O in this case. Between these two cases, the output O will remain at the same logic level. Assume that the A, B and C inputs are now high. This leads the output O to change to high. Then the input B changes low. The output O will remain high rather than change to low. The same notation is used for the rest of this paper.

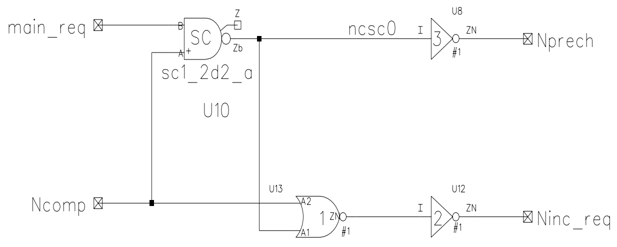

Figure 10: Schematic of the INC control circuit

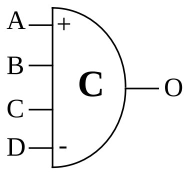

Figure 11: 4 input asymmetric C-gate

6: The design of the BTB block

The behaviour of the BTB block is different from the INC block in that it has a Boolean input signal labelled bypass (see figure 12). This bypass signal can be used when the BTB block is not to be turned on. When this signal is activated the request signal bypasses the BTB block and the BTB operation is not activated. Other signals in figure 12 have the same functions as defined in the INC block, though some signals have different names such as Nenable, done and Nbtb_req (equivalent to Nprech, Ncomp and Ninc_req in figure 8 respectively in terms of their functions). Note that Nenable and Nbtb_req are active low signals and main_req and done are active high signals. The bypass boolean input must reach the BTB block before main_req and must maintain its logic level until the return-to-zero phase of the Nbtb_req is finished.

The STG of the BTB block is more complex than that of the INC block since there is a boolean logic signal named bypass (see figure 13). The P0 and P1 places and the bypass+ and bypass- transitions are to model the boolean input signal. Depending on the logic level of the bypass signal, when main_req enters the BTB block the token in place P3 will go either to the Nbtb_req- transition or to the Nenable- transition. The former is the case when bypassing the BTB block and is a straightforward 4-phase protocol. The latter is the same behaviour as used in the INC block as shown in figure 9.

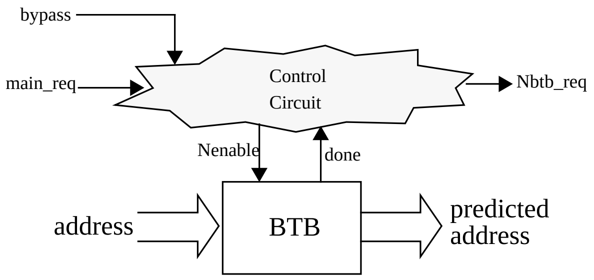

Figure 12: BTB control circuit

The synthesized result for the BTB control circuit is shown in figure 14.

In figure 14, the done signal is forked; one fork going to a complex gate and the other to an inverter. The inverted done signal is then used to drive an input to a C-gate. This may violate the conditions of speed-independence [15]. However, the control circuit is synthesized locally (ensuring that the logic components of the circuit are laid out closely) and simulation of the layout confirms that the circuit works correctly. The same procedure is used to check the speed-independence assumption throughout the control circuit design whenever there is an inverted input.

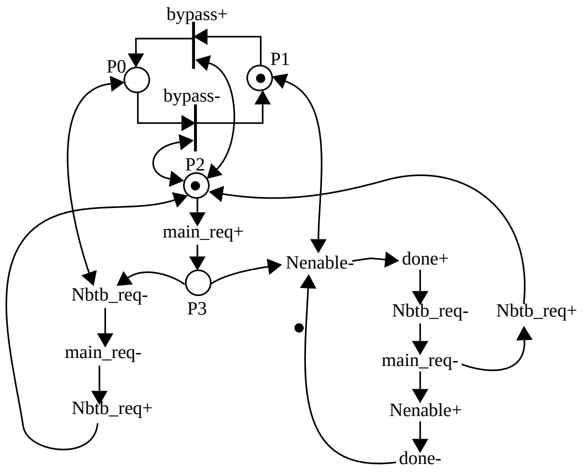

Figure 13: STG for the BTB control circuit

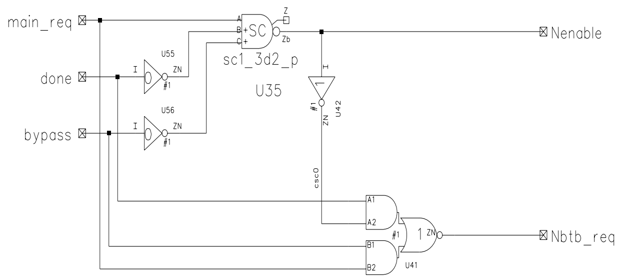

Figure 14: Schematic of the BTB control circuit

7: The design of the EU and IND blocks

The EU block has static datapath circuits and the control circuits are directly implemented using the delay matching method explained in section 4.

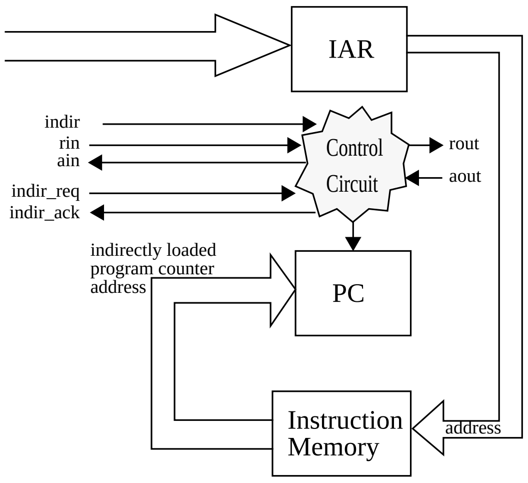

Figure 15: Block diagram of the indirect branch mechanism

The IND transition in figure 3 is chopped off and the function is merged into place J1 in figure 3 (equivalent to the PC block in figure 1), since indirect program address loading is not always initiated but when the boolean signal ‘indir’ at the control circuit in figure 15 is activated. The indirect branch mechanism is shown in the block diagram of figure 15. The IAR and PC blocks are the same as in figure 1. When the indir signal is high (meaning that an indirect branch operation is needed), the control circuit for the PC block waits for the indirect request signal and the bundled indirect channel. This bundled data comes from the instruction memory by indirect addressing via the IAR.

After finishing the indirect channel communication, the main_ack goes back to the ARB in figure 3. This behaviour can be explained as follows. The token in place J1 of figure 3 goes to either the PCreq transition or the BRAdone transition depending on the result of the arbitration in ARB. However, when an indirect branch is required, the PCreq and the BRA-done transitions can't fire until the indirect branch is finished.

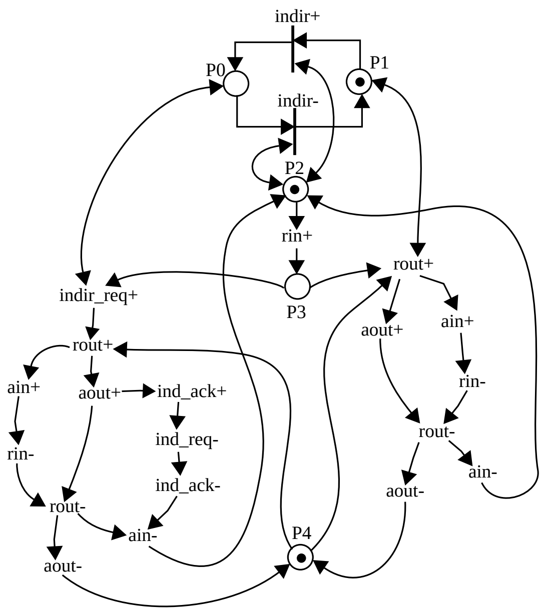

Figure 16: STG for the PC block

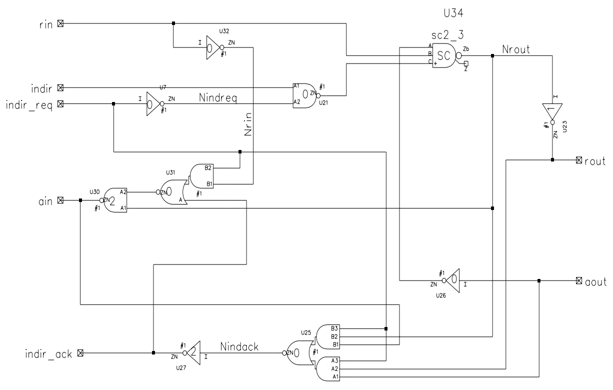

Figure 17: Schematic of the PC block

The STG of this control circuit is shown in figure 16. The indir signal is a boolean signal and the same technique used for the bypass signal in figure 13 is exploited again in figure 16. Therefore the change of the indir signal must arrive at the control circuit before the request signal ‘rin’ reaches the circuit and must be maintained until the return-to-zero phase of the acknowledge signal ‘ain’ is finished. The semi-decoupled technique is adopted for this control circuit to boost speed when rout+ in figure 16 is activated. At the same moment that rout+ is invoked to send a request signal to the next block, ain+ happens to return an acknowledge back to the previous block. There is no need for ain+ to wait for the aout+ response from the next block.

The synthesized result for the PC control circuit is shown in figure 17.

8: Simulation results

Now, we present the performance figures of the instruction prefetch unit in terms of speed.

The definition of the speed in asynchronous design is different from synchronous design since synchronous design can only be measured in terms of the worst case. In synchronous design the worst delay of a block in a chip defines a fixed global clock cycle time regardless of different delays in each block.

In asynchronous design, however, the same logic block can finish its process in different times depending on its running function. In our instruction prefetch unit design two cases can be defined by different running conditions. Since we use a fork and join form in our design, forked blocks must finish their processes before the join. The EU, INC and BTB blocks must finish their processes before the PCmux transition is activated in figure 3. (The IND block is merged in the PC block as explained in section 7.) The EU and the INC blocks must be used every time for the function of the instruction prefetch unit, whereas the BTB block can be turned off by a user. Therefore we can have two operation modes in our performance simulation as shown in table 1.

The normal case in the instruction prefetch unit is that the BTB block is turned on. When an indirect branch happens the cycle time will be longer than the normal case since more time is required for the indirect channel communication. However, this condition is not included in the performance test since it depends on the speed of the instruction memory.

Operation modes	cycle time (nanoseconds)
BTB on	7.0
BTB off	5.5

Table 1: Cycle time in the instruction prefetch unit

9: Conclusions

We have shown how asynchronous design can be achieved using STGs, which can give designers reliable speed-independent circuits. We have exploited different asynchronous design techniques for the control circuits depending on different situations given by a static or dynamic implementation of the datapath circuits, and determined by the decoupling of the dependency of the input and output sides of the control circuit.

10: Acknowledgments

The development of AMULET3 was supported primarily within the EU-funded OMI-DE2 and OMI-ATOM projects, and the authors are grateful to the European Commission for their continuing support for this work. ARM Limited coordinated these projects; their support, and that of the other project partners, is also acknowledged.

The VLSI design work has leant heavily on CAD tools from Compass Design Automation (now part of Avant!) and EPIC Design Technology, Inc. (now part of Synopsys).

11: References

[1] Furber, S.B., “ARM System Architecture”, Addison Wesley Longman, 1996. ISBN 0-201-40352-8.

[2] Woods, J.V., Day, P., Furber, S.B., Garside, J.D., Paver, N.C., Temple, S., “AMULET: An asynchronous ARM microprocessor”, IEEE Transactions on Computers, Vo. 46, No. 4, pp. 385-398, April 1997

[3] Furber, S.B., Garside, J.D., Riocreux, P., Temple, S., Day, P., Liu. J., Paver, N.C., “AMULET2e: An Asynchronous Embedded Controller”, Proceedings of the IEEE, volume 87, number 2, pp. 243-256, February 1999

[4] Segars, S., “The ARM9 Family - High Performance Microprocessors for Embedded Applications”, Proc. ICCD’98, Austin, October 1998, pp. 230-235.

[5] Jaggar, D., “Advanced RISC Machines Architecture Reference Manual”, Prentice Hall, 1996. ISBN 0-13-736299-4

[6] Segars, S., Clarke and Goudge, “Embedded Control Problems, Thumb, and the ARM7TDMI”, IEEE Micro, **15** (5), October 1995, pp. 22-30.

[7] Sutherland, I.E., “Micropipelines”, Communications of the ACM, **32** (6), June 1989, pp 720-738

[8] Furber, S.B. and Day, P., “Four-Phase Micropipeline Latch Control Circuits”, IEEE Trans. on VLSI, 4 (2), June 1996, pp. 247-253

[9] York, R., “Branch Prediction Strategies for Low Power Microprocessor Design”, M.Sc. Thesis, University of Manchester 1994

[10] Jagger, D., “A Performance Study of the Acorn RISC Machine”, M.Sc. Thesis, University of Canterbury, 1990

[11] Garside, J.D., Furber, S.B., Chung, S.-H., “AMULET3 revealed”, Accepted for the 5th International Symposium on Advanced Research in Asynchronous Circuits and Systems, Barcelona, 19-21 April 1999

[12] J. Cortadella et al, “Petrify: a Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers”, IEICE Transactions on Information and Systems, E80-D(3): 315-325, 1997

[13] Chung, S.-H., "The Design of the Branch Target Cache for an Asynchronous Microprocessor", M.Phil. Thesis, University of Manchester, October 1998

[14] Chu, T-.A., "On the Models for Designing VLSI Asynchronous Digital Circuits", Integration, the VLSI journal, 4(2): 99-113, June 1986

[15] K. van Berkel, "Beware the Isochronic Fork", Integration, the VLSI journal, vol. 13, pp. 103-128, 1992

[16] Rosenblum, L. Y., Yakovlev, A. V., "Signal graphs: from self-timed to timed ones", Proc. of the Int. Workshop on Timed Petri Nets, Torino, Italy, July 1985, IEEE Computer Society Press, NY, 1985, pp. 199-207