RAPPID Synchronization

Satyajit Chakrabarti and Sukanta Pramanik

Department of Computer Science, University of British Columbia, Canada

E-mail: {satyajit, pramanik}@cs.ubc.ca

Abstract – **This paper proposes a solution to the synchronization issue in RAPPID that has prevented it from being used in synchronous processors like the Pentium family of processors, in spite of its higher average case throughput. Our first approach explores the possibility of using an early count of the number of instructions pending in the decoder. If the availability of these instructions can be predicted within a bounded time then the execution unit can carry on upto that point without error. Our second approach moves the possibility of metastability from the data path to the control path. The data is placed in the path before the control signal which ensures a stable data when the control is stable or metastable. If the metastability in control signal is not resolved within a single clock-cycle it is re-sampled and the re-sampled signal overwrites the previous signal, thereby ensuring a worst case latency of one clock-cycle.**

Index Terms – **synchronization, asynchronous circuits, RAPPID, metastability, synchronization failure.**

I. Introduction

ASYNCHRONOUS circuits are being considered as an attractive alternative to synchronous circuits because of their potential to achieve higher average-case performance [10, 13], lower power consumption as well as their ability to eliminate the clock-skew problem. The *fixed* clock period of a synchronous circuit is chosen as a result of worst-case timing analysis. But in an asynchronous circuit an operation begins when all the operations on which it depends have occurred [4]. As the asynchronous circuitry does not wait for the next clock signal to arrive, the faster parts of the circuit are not held back by the high-latency regions. Thus, well designed asynchronous circuits can achieve better operating frequencies than the synchronous circuits.

Synchronous circuits also have their own advantages-ease of implementation, simplicity in dealing with hazards, availability of mature design and testing tools, etc. All asynchronous systems again come with additional operational constraints. As there is no global clock, additional circuitry is used to prevent hazards, or glitches in output. This circuitry along with the handshaking can increase the area and the delay of the asynchronous circuits. It can even make the average case delay in an asynchronous circuit larger than the worst case delay for the synchronous version. Moreover, the introduction of asynchronous communication latencies inside the design may lead to various other overheads which in some cases may offset the power gains due to the absence of the global clock [14], [9].

Thus the solution may lie in a system that uses both synchronous and asynchronous circuits. But the key difficulty in a mixed system is the *synchronization failure*. Synchronization is required when an asynchronous input is sampled into a synchronous system, or when a signal traverses the boundary between two clock domains. If an input signal changes too close to a clock edge, the circuit may enter a *metastable state* [1]. A metastable state is a stable state, which is neither a logic 0 nor a logic 1, but rather lies somewhere in between these two. Moreover the circuit can reside in this state for a nondeterministic amount of time. If this metastable state persists until the next clock cycle then the subsequent logic stages can interpret this data as either a logic 0 or a logic 1. This may lead the system to an incorrect state, causing it to fail. Such a failure is called *synchronization failure* [11].

RAPPID (Revolving Asynchronous Pentium® Processor Instruction Decoder) is a prototype IA32 instruction length decoding and steering unit that was implemented using self-timed techniques[10]. RAPPID achieves three times the throughput and half the latency, dissipating only half the power and requiring almost the same area as an existing 400MHz clocked circuit. But as the rest of the microprocessor is still synchronous, RAPPID chip is not used in any of the Pentium® processors. The main issue that prevents the RAPPID chip from being

used in production line is the synchronization problem.

The simplest solution to the synchronization problem is to use double-latching scheme or pipeline synchronization [12]. These methods actually reduce the probability of failure to an acceptable level without eliminating it. But a major drawback of this scheme is the latency of communication as one or more extra cycle is added to the data path, even in the absence of metastability.

The second method of solution is stopping or stretching the synchronous module's local clock long enough, to ensure that the metastability is resolved. There are many *globally asynchronous locally synchronous* (GALS) solution to this problem that uses pausible clocks [2], [16], [15] in the locally synchronous blocks to eliminate metastability-related failures. Although this approach can effectively eliminate metastability, it requires access to the local clock-generation scheme. The long clock-buffer and PLL also cause some limitations in this approach.

This project aims towards minimizing the synchronization problem of using RAPPID chip in a synchronous processor design. Our first approach explores the possibility of using an early count of the number of instructions in the decoder. Pipeline synchronization is used to synchronize this count and the synchronous Re-order Buffer in the execution unit subtracts from it the number of instructions it has already seen to determine the instructions still pending in the decoder. If the availability of these instructions can be predicted in a bounded time then the execution unit can continue inside this limit without error.

High performance mixed clock communication can also be performed by re-sampling [5]. Re-sampling can guarantee the validity of data at the server side after a bounded time. Although the metastability still occurs and its resolution time is still unbounded, re-sampling simply over-writes the metastable data. The basic idea is that a delay difference between the inputs of two flip-flops can guarantee that, if the delayed input leads the respective flip-flop to a metastable state, then the un-delayed input will not lead the output of the other flip-flop to a metastable state if the delay difference is larger than the setup time of the flip-flops. The idea is illustrated in Figure 1. At the clock edge *1*, *D1* is unstable at the setup time of the flip-flop. Then *Q1* goes into a metastable state, which is not resolved in one clock cycle. *D1* is sampled again in the clock cycle *2* and then *Q1* becomes stable high even though the metastability from the previous sampling has not resolved yet. Since the delay between *D2* and *D1* is longer than the setup time of the flip-flops, *Q2* is stable whenever *Q1* is stable or metastable. Therefore, *Q1* can be used as control signal to indicate that *Q2* is stable.

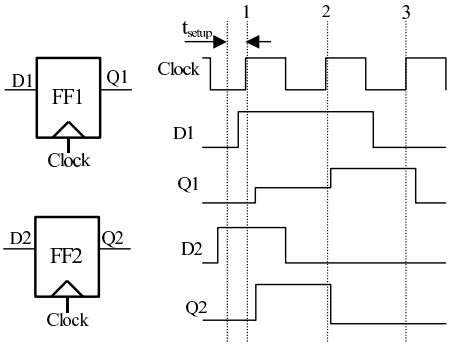

Figure 1: Principle of re-samplings and input delay between two flip-flops

Huang et al [5] used this idea to develop a circuit to communicate between two independent clock domains. Our second approach shows that with minor changes to their circuit it can be used to reduce or even avoid the synchronization problem of RAPPID chips.

The rest of the report is organized as follows. An introduction to the RAPPID and Pentium® processor microarchitecture is given in Section II. The early-counting solution is discussed in Section III and the Re-sampling solution in Section IV.

II. RAPPID ARCHITECTURE

RAPPID implements an asynchronous instruction length decoder, [10] for the Pentium® Processor instruction set [6]. It receives a 16-byte cache line and speculatively computes 16 instruction lengths in parallel, assuming that each byte starts a new instruction. A Tag Unit in the first byte of an instruction passes a tag downstream to the first byte of the next instruction; 4x16 Tag Units are connected in a torus. The instructions are routed on four separate 62-bit crossbar channels to the output, Figure 2.

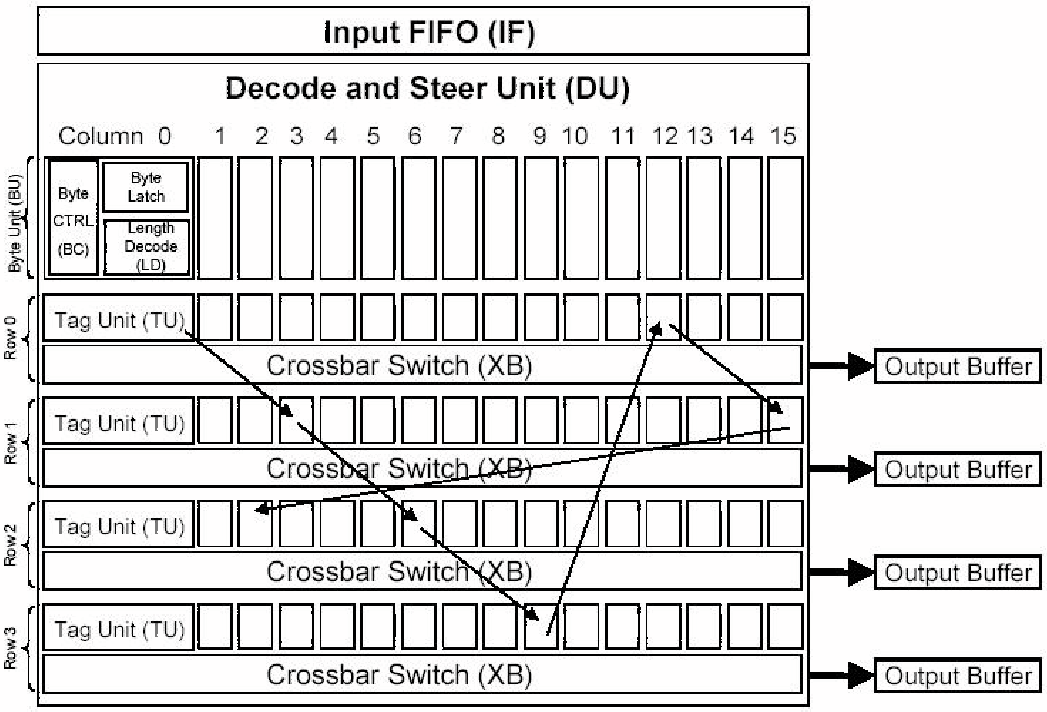

Figure 2: RAPPID Microarchitecture – The Input FIFO holds a 16-byte wide instruction cache line, containing 5 instructions on average. Instructions are decoded parallel over 16 byte Decode Columns with an average rate of 0.72 GOPS, giving a total average of 5 x 0.72 = 3.6 GIPS. To maintain this average at the output side, a 4-step top-down Tag cycle gathers and distributes the instructions over 4 Output Buffers, each operating at 0.9 GIPS. The sequence of arrows in the picture illustrates the 3.6 GIPS flow through the Tag Units for a typical scenario with 5 length-3 instructions.

From the microarchitecture of Intel® Pentium® M processor, Figure 3, [7], we find that the RAPPID chip is meant to be part of the Fetch/Decode stage. Our approaches assume an asynchronous hand-shakable Instruction Decoder follows the Length Decoder while the other following units starting from the Execution unit are synchronous.

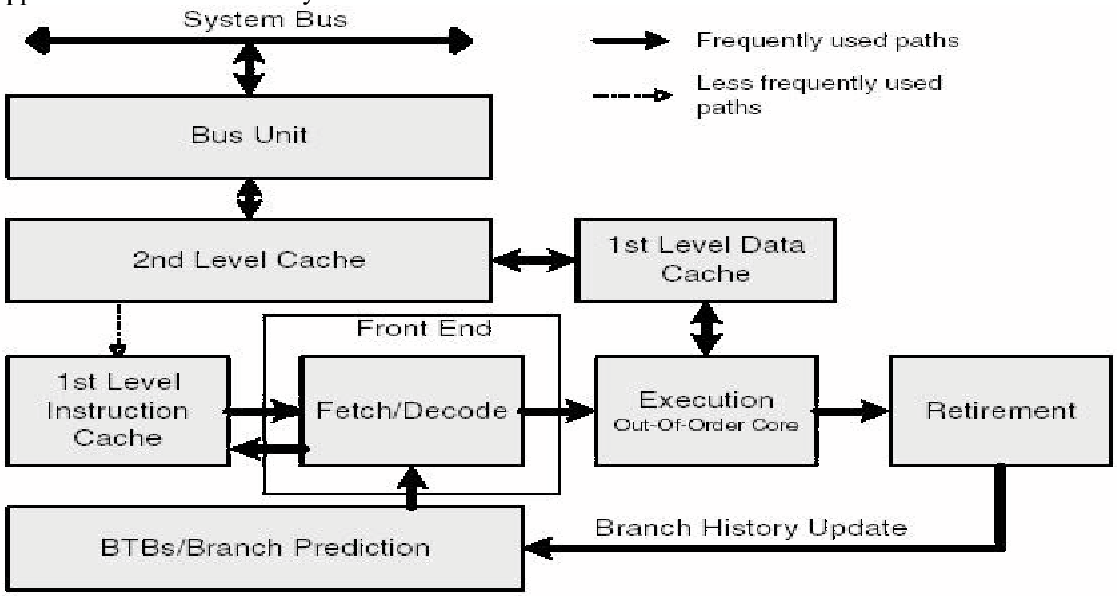

Figure 3: Intel® Pentium® M processor microarchitecture. P6 core (Pentium Pro through Pentium III).

III. EARLY COUNT OF DECODED INSTRUCTION

Our initial approach emphasizes on predicting the number of instructions that will be available inside a definite interval of time. A block diagram of the basic idea is given in Figure 4.

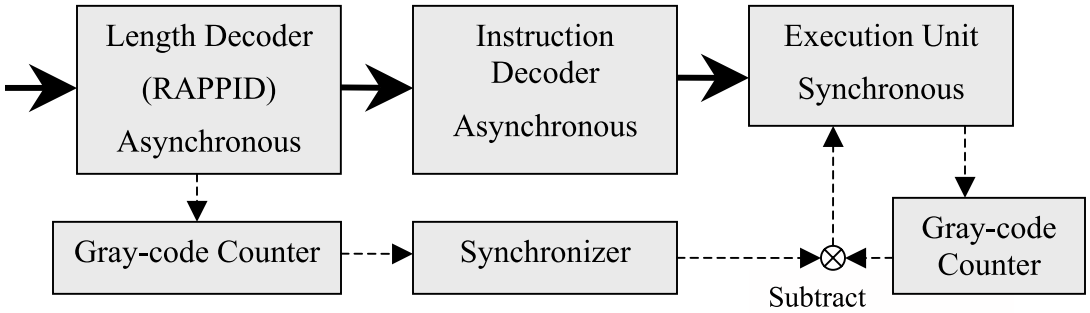

Figure 4: Block diagram of early counting

As the instruction lengths are being decoded, the 4-bit gray code counter keeps track of the number of instructions already decoded. The execution unit keeps count of how many instructions it has already taken. The difference between these two indicates the number of instructions already in the decoding stage. The decoded instruction count can be synchronized in parallel with the Instruction Decoder using Pipeline Synchronization. In that case the instruction decoding time is the amount of time available for synchronization.

Figure 5 shows the general instruction format for Pentium® processor [8]. The instruction length varies from 2 to 15 bytes. The instructions of length 2-3 are very frequent whereas others are much less frequent and instructions of length greater than 7 are extremely rare [10]. The RAPPID design exploits this trend and the length decoding cycle in it is optimized for common instructions.

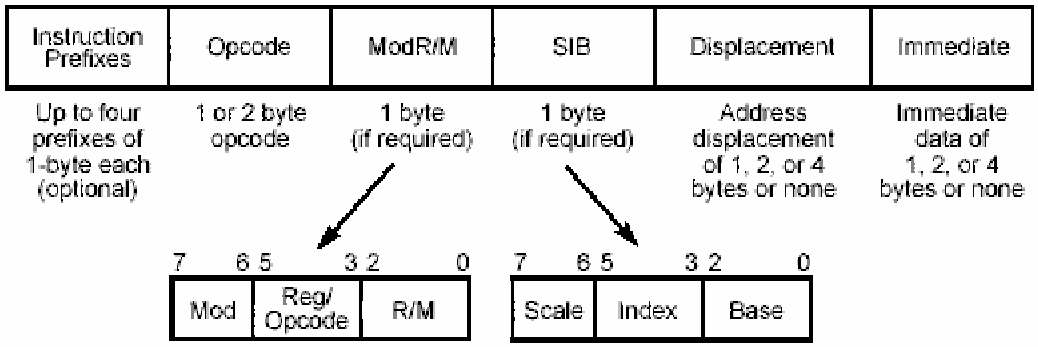

Figure 5: Pentium® instruction format

P6 processor's pipeline which consists of 10 stages is given in Figure 6, [7]. In it the instruction-length decoding is done overlapping in stages 2 and 3. Remaining stage 3 followed by stages 4 and 5 is used for decoding of the instruction. But the P6 architecture is a synchronous design and thus although the instruction decoding takes two and a half clock cycles, it can not be said definitively that the decode-cycle remains busy throughout this entire period. Again, decoding time may also be dependent upon the length of the instruction which is hidden under the worst-case clock cycle. This dependency cannot be tested without tapping into the decode cycle. So we cannot still say whether the completion of length decoding ensures the availability of the decoded instruction from the decoder after a bounded amount of time.

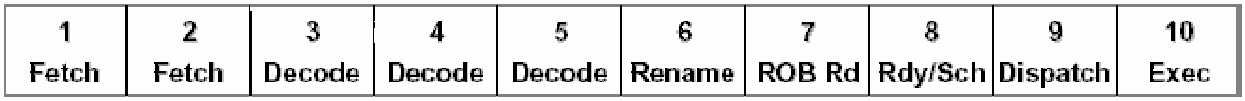

Figure 6: Pentium® pipeline stages

IV. RE-SAMPLING SYNCHRONIZATION

The block diagram of the communication scheme used by Huang et al [5] for communicating between a locally synchronous sender and a locally synchronous receiver is shown in Figure 7. Both the sender and the receiver use fixed-frequency clocks running at different speeds.

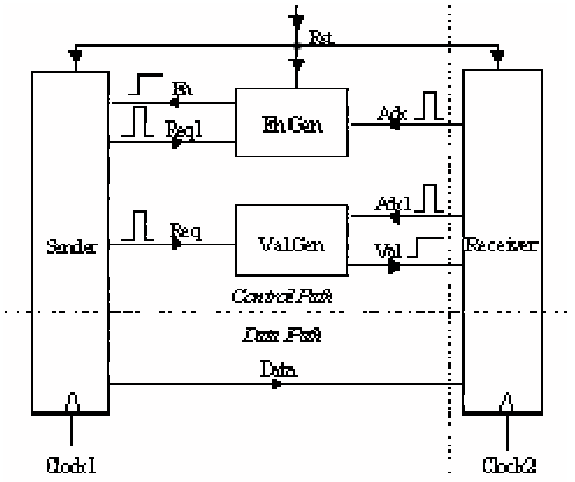

Figure 7: Interface Adapter between a sender and a receiver

Two pulse-mode signals *Req* and *Ack* are used to signal request and acknowledge of data. Signal *Req1* and *Ack1*, both also pulse-mode, are past a negative delay from *Req* signal and *Ack* signal respectively. The *Data* signal signifies data sent from the sender to the receiver. The *Val* signal represents validity of the data at the input of the receiver: whenever *Val* is high, data is stable. The *En* signal represents the enable signal of the sender: when *En* is high, the sender may update the D*ata* signal. The *Rst* is the master reset that sets the *En* signal.

A typical communication cycle is as given below,

1. *Rst* pulse or a positive pulse on *Ack* sets the *En* signal, which will be sampled by the sender at the rising edge of *Clock1*.
2. When the sender samples a high *En* signal, it will generate a positive *Req1* pulse to clear the *En* signal, and a positive *Req* pulse.
3. A positive pulse on *Req* will set the *Val* signal, which will be sampled by the receiver at the rising edge of *Clock2*.
4. Upon sampling *Val* high, the receiver produces a positive *Ack1* pulse to clear the Val signal and a positive *Ack* pulse, and samples data from the sender.

The implementations of the EnGen unit and the ValGen unit are shown in Figure 8. A point to note in this circuit is that *En* is an asynchronous input to the sender and *Val* is an asynchronous input to the receiver. Thus the asynchronous inputs may generate metastability in the sampling circuits. Re-sampling is used to over-write this metastability at the expense of one clock cycle delay in the worst case.

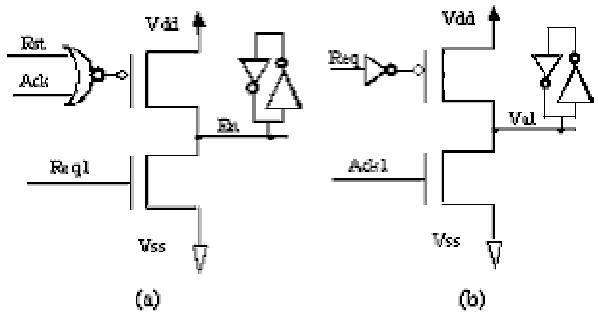

Figure 8: Implementation of (a) EnGen, (b) ValGen.

The implementation of the sender and receiver is shown in Figure 9.

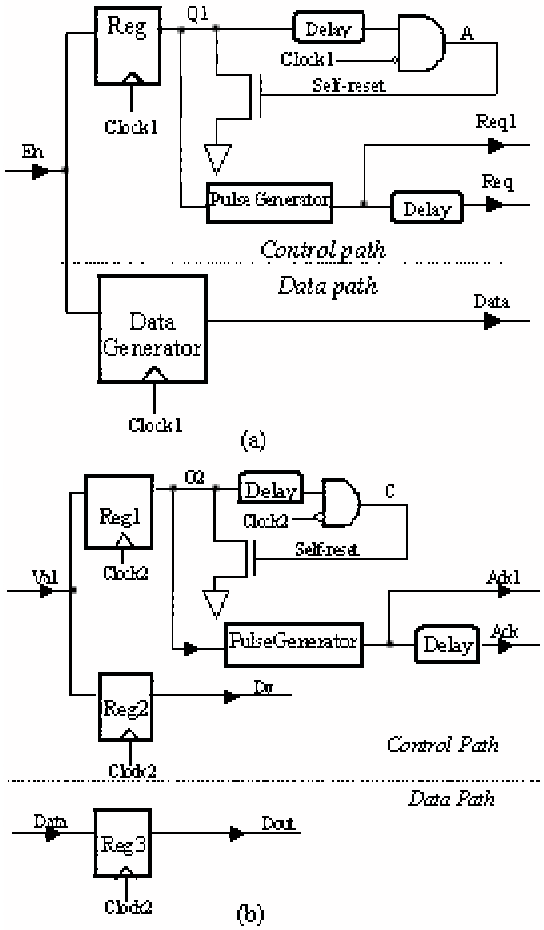

Figure 9: Implementation of the (a) locally synchronous sender, (b) locally synchronous receiver

The locally synchronous sender generates a synchronous request signal on *Req* bundled with data to inform the locally synchronous receiver that new data is available. The receiver end samples the data lines and generates an acknowledge event as a positive pulse on *Ack* when it is ready to accept new data. If we look at the sender circuit we will find that the *data* is made available before the *Req* signal is placed. If the delay between these two is greater than the t_{setup}, then *data* is stable whenever *Req* is stable or metastable.

Our synchronization problem has the same scenario, except in this case we have an asynchronous sender, the decoder. We have assumed that our asynchronous decoder has two handshake signals *Ready* and *Start*. The decoder places a decoded instruction to the data path and asserts the signal *Ready*. The *Start* signal works as an acknowledgement and it pulls down the Ready signal to low. The *Ready* signal is again raised high after placing the instruction to the data path. Our Decoder-Synchronizer circuit, Figure 10, makes a minor change to the locally synchronous sender as the *Ready* signal is used to drive the control path register.

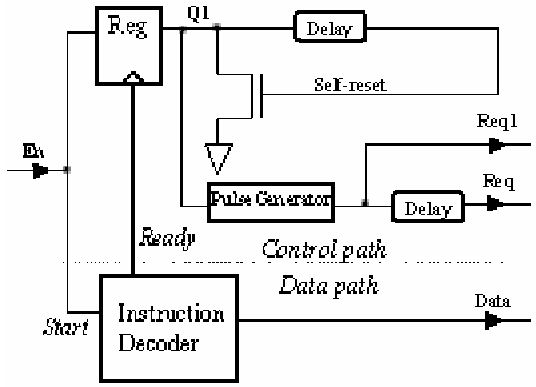

Figure 10: Implementation of the Decoder-Synchronizer circuit

Here Reg is a C^2MOS edge-triggered D flip-flop which samples data at the rising edges of *Ready* signal. A high on the output of Reg, *Q1*, triggers PulseGenerator [3] to generate a positive pulse *Req1* upon a rising edge of *Q1*. After the *En* signal becomes high, when the Instruction decoder finishes decoding an instruction, it lowers the *Ready* signal, updates *data* line and then raises the *Ready* line again. Thus the *En* signal actually can never change when the *Ready* changes its state. Thereby there is no metastability in this part of the circuit.

The locally synchronous receiver circuit from Figure 9(b) is used in the execution unit for the synchronization. The Reg1, Reg2 and Reg3 are C^2MOS edge-triggered D flip-flops sampling data at the rising edges of Clock2. Output from Reg1, *Q2* is floating while the Clock2 becomes low, so *Q2* can be reset externally. A high on *Q2* triggers PulseGenerator to generate a positive pulse upon its rising edge. At the same time, the high *Q2* will reset itself through a NMOS transistor when *Clock2* becomes low. Signal *Dv* is used to indicate that *Dout* is valid.

A timing diagram of the control signals in the synchronization process is given in Figure 11. The signal transitions are numbered sequentially. So a transition marked (*n*) causes one or more transitions numbered as (*n+1*).

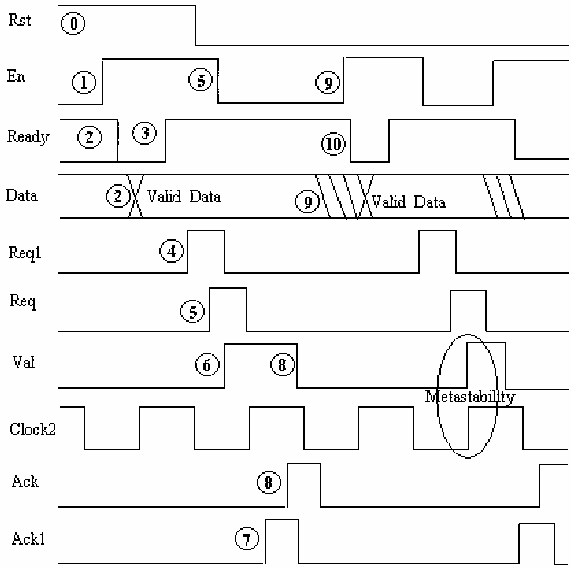

Figure 11: Timing diagram of the control signals in synchronization process

As *Val* is asynchronous to Clock2 in Fig. 9(b), it can cause metastability in the outputs of Reg1 and Reg2 (*Q2* and *Dv*) when it changes too close to the rising edge of *Clock2*. One such case is marked in Figure 11. If metastability on *Q2* lasts longer than the positive phase of *Clock2*, the delay element in the self-reset circuit ensures that a positive *Ack1* pulse won't be generated until *Q2* is free of metastability. If metastability on the outputs of the Reg1 and Reg2 lasts longer than one clock cycle of Clock2, *Val* and *Data* will keep their values to the next clock cycle of *Clock2* and be re-sampled on the next rising *Clock2* edge, overwriting *Q2*, *Dv* and *Dout* with stable values.

This re-sampling ensures that metastability can't last infinitely in the control path when it occurs. Thus the worst case for unstable time of Reg1 and Reg2 is one

extra clock cycle before their metastable outputs are over-written.

The decoder cannot send a new instruction until it gets an acknowledgement in *Start*. If we consider T_{total} as the time between the *Ready* going up and the acknowledgement in *Start*, it includes

i. Two D-Flip-flop propagation delay (Reg, Reg1), t_{prop}
ii. Two pulse-generator latency, t_{pgen}
iii. Two set-up time delays (for *Req*, *Ack*), t_{setup}
iv. One pull-up time for CMOS(*En* going up), t_{up}

So the combination requires a tight coupling between decode time and T_{total}. If the decode time is much longer than T_{total} then the execution cycle waits for the new instruction. Whereas if decode time is less, then the decoder waits to place the decoded instruction in the data path.

V. CONCLUSION

Both of our approaches require further investigation to the decoding latency for different types of instructions to make some strong comment on them. Our first approach tries to place the synchronization time in parallel to the instruction decoding latency. But to do that we need to make sure that the decode-latency provides sufficient time for minimizing the synchronization failure probability to an acceptable level. Another point is to find how much does the decode-latency vary with the length of an instruction.

Our second design uses re-sampling to over-write the metastable control signal. And as the data is placed t_{setup} time before the control, it is always stable. This approach guarantees that there is a worst-case delay of one clock-cycle. But the effectiveness of this design still depends on the relationship between decode-time and T_{total}, as starving would occur if these two times are not closely related.

VI. REFERENCES

[1] T.J.Chaney, and C.E.Molnar, "Anomalous behavior of synchronizer and arbiter circuits," In *IEEE Transaction on Computers*, vol. C-22, pp. 412-422, Apr 1973.

[2] D. M. Chapiro, "Globally-Asynchronous Locally-Synchronous Systems," *PhD thesis,* Stanford University, 1984.

[3] M.D. Clercq and R. Negulescu, "1.1-GDI/s Transmission Between Pausible Clock Domains," In *IEEE International Symposium on Circuits and Systems (ISCAS 2002)*, Vol. 2, 2002, Pages: 768–771.

[4] A.Davis, and S.M.Nowick, "An Introduction to Asynchronous Design," *Technical Report UUCS-97-013*, Computer Science Department, University of Utah, Sep. 1997.

[5] S.Huang, and R.Negulescu, "High-performance mixed-clock communication with re-sampling", yet unpublished.

[6] Intel Corporation, *Pentium Processor User's Manual.*

[7] Intel Corporation, *IA-32 Intel® Architecture Optimization User's Manual.*

[8] Intel Corporation, *Intel® Architecture Software Developer's Manual.* vol 1, 2.

[9] A.Iyer, and D.Marculescu, "Power and performance evaluation of globally asynchronous and locally synchronous processors," In Proceedings of *International Symposium on Computer Architecture,* pp- , May, 2002.

[10] S.Rotem, K.Stevens, R.Ginosar, P.Beerel, C.Myers. K.Yun, R.Kol, C.Dike, M.Roncken, and B.Agapiev, "RAPPID: An asynchronous instruction length decoder," In Proceedings of *International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 60–70, 1999.

[11] C.L.Seitz, "System Timing", In *Introduction to VLSI Systems*, by C.Mead and L.Conway, Addison-Wesley, 1980, ch 7.

[12] J.N.Seizovic, "Pipeline synchronization", In Proceedings of *International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 87–96, 1994.

[13] K.Stevens, S.Rotem, R.Ginosar, P.Beerel, C.Myers. K.Yun, R.Kol, C.Dike, and M.Roncken, "An asynchronous instruction length decoder," In *IEEE J. Solid-State Circuits*, vol. 36, pp. 217-228, Feb 2001.

[14] A.E.Sjorgen, and C.J.Myers, "Interfacing synchronous and asynchronous modules within a high-speed pipeline," In *IEEE Transactions on Very Large Scale Integration Systems*, vol 8, no 5, pp – 573-583, Oct, 2000.

[15] A.E. Sjogren and C. J. Myers, "Interfacing synchronous and asynchronous modules within a high-speed pipeline," In Proceedings of the *Seventeenth Conference on Advanced Research in VLSI, 1997,* Pages: 47–61.

[16] K.Y.Yun, and R.P.Donohue. “Pausible Clocking: A First Step Toward Heterogeneous Systems,” In Proceedings of the *IEEE International Conference on VLSI in Computers and Processors (ICCD)*, 1 996, pages: 118–123.