The Lutonium: A Sub-Nanojoule Asynchronous 8051 Microcontroller

Alain J. Martin, Mika Nyström, Karl Papadantonakis, Paul I. Pénzes, Piyush Prakash, Catherine G. Wong, Jonathan Chang, Kevin S. Ko, Benjamin Lee, Elaine Ou, James Pugh, Eino-Ville Talvala, James T. Tong, Ahmet Tura
Department of Computer Science
California Institute of Technology
Pasadena, CA 91125

***Abstract*—**
We describe the Lutonium, an asynchronous 8051 microcontroller designed for low Et^2. In 0.18-μm CMOS, at nominal 1.8 V, we expect a performance of 0.5 nJ per instruction at 200 MIPS. At 0.5 V, we expect 4 MIPS and 40 pJ/instruction, corresponding to 25,000 MIPS/Watt. We describe the structure of a fine-grain pipeline optimized for Et^2 efficiency, some of the peripherals implementation, and the advantages of an asynchronous implementation of a deep-sleep mechanism.

I. Introduction

The Lutonium is a quasi delay-insensitive (QDI) asynchronous 8051-architecture microcontroller designed for energy efficiency. It is the demonstration vehicle of a DARPA-sponsored research project investigating the energy advantages of asynchronous design. At the moment of writing, the design is not complete, but all units are described at the transistor level, and we can already estimate the performance. In TSMC's 0.18-μm CMOS process offered via MOSIS, we expect an energy consumption of 500 pJ per instruction and an instruction rate of 200 MIPS at the nominal 1.8 V. At 0.9 V, we expect 140 pJ and 66 MIPS.

The energy efficiency of the Caltech asynchronous QDI design approach was already apparent in the performance of the 1989 Caltech Asynchronous Microprocessor [2] and of the 1997 Caltech MiniMIPS [4], an asynchronous MIPS R3000 microprocessor. In addition to increased energy efficiency at the nominal operating voltage, experimental data from these chips revealed that the robustness of QDI circuits to delay variation allows these circuits to run at very low voltages, even slightly below the transistor threshold voltage. In designing the Lutonium, we have made full use of our ability to adjust E and t through voltage scaling: the voltage-independent Et^2 is the metric we are striving to minimize, where E is the average energy per instruction, and t is the cycle time. The Lutonium is therefore not, strictly speaking, designed for low power but for the best trade-off between energy and cycle time. This metric has been introduced and justified in several papers, and we shall recapitulate the argument in the next section.

We cannot justify the choice of the 8051 ISA entirely on the basis of energy efficiency. It is a complex and irregular instruction set, which is bound to increase the energy cost of fetching and decoding the instructions. Also the use of registers is highly irregular, another source of energy consumption. The choice of the 8051 is justified by the fact that it is the most popular microcontroller, hence it is often found in applications where energy efficiency is important.

The paper is organized as follows. We first briefly recapitulate the arguments in favor of the Et^2 metric. We then describe the general design style and the design methodology, both very similar to the ones introduced for the MiniMIPS, and explain the main changes and refinements introduced for the Lutonium. We then present the instruction set and the general organization of the pipeline. We describe several parts of the architecture in more detail: the fetch loop and the decode, the deep-sleep protocol, the tree buses, and the interrupt mechanism. Finally, we discuss the performance of the prototype and compare it with other implementations of the 8051.

II. The Et^2 Metric and Energy-efficient Design

In first approximation, the (dynamic) energy dissipated during a transition in a CMOS system is proportional to CV^2, and the transition delay is inversely proportional to V. Therefore energy and delay can be traded against each other simply by changing the supply voltage, and it would be foolish not to take advantage of this freedom; this means that when we can adjust the supply voltage, we should not simply optimize one of the two metrics and ignore the other. The best way to combine energy and delay in a single figure of merit is to use the product Et^2 since it is independent of the voltage to first approximation. Given two designs A and B, if the Et^2 of A is lower than that of B, then A is indeed a better design: for equal cycle time t, the energy of A is lower than that of B, and for equal energy, the cycle time of A is lower than that of B. In several papers, we have given experimental evidence of the range and limits of the assumptions under which the Et^2 metric is valid [8]. SPICE simulations also show that under normal operating conditions an Et^2-better circuit will give better energy performance and better cycle-time performance.

More sophisticated formulas than Et^2 are possible, but they do not necessarily work better in practice: the main practical deviations from our simple theory are the effect of the transistor threshold and the effect of velocity saturation on the speed of operation. Since these two effects largely counteract each other, Et^2 works very well around the design voltage of most

Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems (ASYNC'03)
1522-8681/03 $17.00 © 2003 **IEEE**

modern CMOS technologies. Furthermore, it is a much easier metric to handle than one that tries to incorporate the threshold or velocity saturation directly.

We have developed a theory of Et^2-optimal designs, the results of which have influenced the design of the Lutonium, particularly in the areas of high throughput, slack matching, transistor sizing, and conditional communication. (However, we do not claim that the Lutonium design is Et^2-optimal, as such a claim is beyond the current state of the art.)

One of the theoretical results is that an Et^2-optimal pipeline is short. For the Lutonium, we chose the highest throughput possible given nonspeculative execution, and a minimally speculative fetch loop. We realized that the common-case critical path in such a design would be the "fetch loop" (see below), and we chose our throughput target to be the fastest possible cycle time of this loop.

Optimizing the throughput of an asynchronous system involves *slack-matching* it. The simplest way of slack-matching a system is to treat it like a synchronous retiming problem: have the same number of pipeline stages on all paths. However, because the slack-matching buffers are almost always faster than the elements that perform computations, they can absorb timing variations, and it is not always necessary for all paths to go through an absolutely equal number of pipeline stages in order to optimize throughput. This has repercussions for Et^2-optimal design: whereas a designer who is only interested in t does not mind adding more buffers than strictly necessary, the designer that wants to optimize Et^2 must take into account the energy cost of the extra slack-matching buffers. Therefore, slack-matching has more subtleties for an Et^2-optimal system than for a t-optimal system; the optimized system has fewer slack-matching buffers in it. The MiniMIPS was over–slack-matched in this regard.

Transistor sizing for optimal Et^2 is achieved when the sum C of all gate capacitances is approximately $2P$ where P is the total parasitic capacitance [8]. *Sizing for optimal energy consumption is not minimal sizing!* (Optimal energy consumption is not minimal energy consumption.)

Extensive simulations of the MiniMIPS gave us the opportunity to gather invaluable information about the energy budget of an asynchronous microprocessor. In particular, it appears that only 10% of the total energy consumed in the processing of an instruction goes in the actual execution of the instruction—for instance the actual energy spent in the ALU during an `add` instruction. The remaining 90% of the energy is spent in communication: fetching the instruction and decoding it consumes 45% of the energy (this happens even for a `NOP`), and the final 45% goes into moving parameters and the result between execution units and registers. These results are relevant because the MiniMIPS design style is similar to the Lutonium design style.

We concluded that we should reduce communication at the expense of adding local computation. For example, we used a Huffman code based on instruction frequencies to control the main buses, which were segmented according to the corresponding Huffman tree.

III. Logic Family and Design Method

The logic family of the Lutonium is essentially the same as that of the MiniMIPS. Why a logic family chosen for high throughput should be optimal for Et^2 has not been established rigorously, and may not be true! That it would give good Et^2 performance compared to other asynchronous logic families can be argued and is supported by experimental evidence.

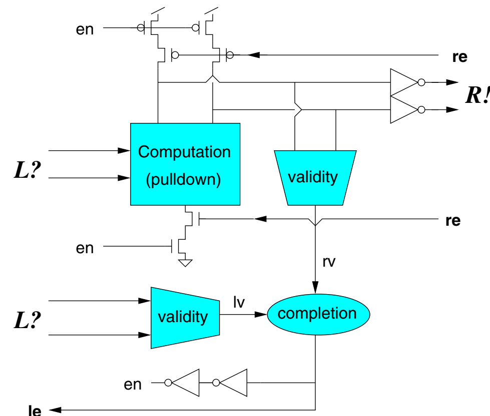

Figure 1. PCHB circuit template.

The basis of the logic family is the buffer described in CHP as:

$$*[L?x; R!f(x)]$$

The buffer may have several inputs and outputs including conditional ones. The above instance is the simplest one in the family. In traditional QDI design, the standard implementation consists of separating the control part from the datapath. This design style is simple and general and was used for the Caltech Asynchronous Microprocessor. But it puts high lower bounds on the cycle time, forward latency, and energy per cycle. First, in the control part the four-phase handshake on L and the four-phase handshake on R are totally ordered, putting all 8 transitions in sequence. Secondly, the completion-tree delay is included twice in the handshake cycle between two adjacent buffer stages, and it is proportional to the logarithm of the number of bits in the datapath. Finally, the explicit storing of the variable x in a register adds considerable overhead in terms of both energy and forward latency.

A solution to the problem was proposed for the design of the MiniMIPS, where we introduced a logic family based on very fine-grain pipeline stages implemented as "precharge half-buffers," or PCHB. First, in order to reduce the completion tree delay and keep the design QDI, the datapath has to be partitioned into independent portions, each with their own completion tree. The size of the portions is chosen in such a way that the completion-tree delays fit within the alloted cycle-time delays. Secondly, each partial buffer processes a portion of data small enough that control and datapath do not have to be separated. Thirdly, the explicit storing of the input data in a local register is eliminated by "reshuffling" the handshake sequence on L and R in such a way that the data can be processed directly from the input wires of L to the output wires of R. The term PCHB refers to the specific reshuffling chosen; another advantage of this reshuffling is that it has only two transitions

Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems (ASYNC'03)
1522-8681/03 $17.00 © 2003 **IEEE**

on the forward latency, i.e., between the data's being valid on L and the result's being valid on R. The basic CMOS implementation of a PCHB stage is shown in figure 1. Observe that all computation is done in the pulldown network with L as input.

The choice of such a fine-grain pipeline stage has drastic consequences on the organization of the whole system. Because each stage can do only a modest amount of computation (essentially limited by the size of the pull-down circuitry), any non-trivial computation must be decomposed into a network of PCHB stages, with important repercussions on latency, throughput, and energy.

While the small size of a PCHB stage helps keep the cycle time low, it may also increase the global latency on computation cycles with feedback, in particular the so-called *fetch loop*, which is a critical part of the Lutonium pipeline. The relationship between a stage period p, a stage latency l, and a pipeline cycle-length n (in terms of the number of stages) is given by the equation $p = n*l$. A crucial step in the design process is choosing the individual stages and the length n of a pipeline in such a way that the equation is satisfied for p and l corresponding to the optimal cycle time and latency of each stage.

In the design of the Lutonium, we chose p to be equal to 22 elementary transitions (compared to 18 in the MiniMIPS). This choice was dictated by the complexity of the fetch loop, which we expected to have a length of 11 stages. Once this choice was made, all pipeline stages that we required to be able to run at the full throughput of the processor (i.e., at least all pipeline stages that have to operate once per instruction, and some others) had to be designed for a cycle period close to 22 transitions, and all cycles had to be *slack-matched* to be approximately 11 stages long.

IV. THE 8051 ISA

The 8051 microcontroller has 255 variable-length instructions, from one to three bytes. We have added a 256th instruction for writing instruction memory during bootstrapping. The opcode of an instruction is always encoded in the first byte. The second and third bytes are operands. They might specify a relative or absolute branch target, an indirect address, or an immediate operand. The 8051 is a Harvard architecture: instruction memory and data memory are separate.

The instruction set provides six addressing modes: (1) in direct addressing, the operand is specified by an 8-bit address field in the instruction representing an address in the internal data RAM or a special-function register (SFR); (2) in indirect addressing, the instruction specifies a register containing the address of the operand in either internal or external RAM; (3) in banked addressing, the register banks containing registers `R0` through `R7` can be addressed by a 3-bit field in the opcode; (4) some instructions operate on specific registers; (5) in immediate-constant mode, the constant operand value is part of the instruction; (6) the indexed addressing mode is used to read the program memory (the address is obtained by adding the accumulator to a base pointer).

The 8051 peripherals include logic ports, timers and counters, four I/O ports, and an interrupt controller.

V. THE LUTONIUM DESIGN

Instructions in the 8051 architecture may have implicit operands as well as explicit operands. An example of an 8051 instruction that involves many implicit operands is `ADDC A, @Ri`, or *add with carry accumulator to indirect* `Ri`. This instruction requests that the contents of the memory address pointed to by register `Ri` be added to the accumulator, with a carry-in, and stored in the accumulator. In the 8051 ISA, register `Ri` is itself indirect because the "register bank" is under software control. Therefore, `ADDC A, @Ri` involves the following actions: read the carry-in `PSW.C` out of the processor status word (PSW); read the register-bank selector `PSW.RS` out of the `PSW`; combine `PSW.RS` with `Ri` to make the memory address pointed to by `Ri`; read the contents of `Ri`; read the contents of the memory address pointed to by `Ri` (i.e., `@Ri`); read the accumulator; add the accumulator and `@Ri`; store the eight low-order bits of the result in the accumulator; store the carry-out in `PSW.C`; store the overflow bit of the add in `PSW.OV`; and finally store the carry-out of bit 3 in `PSW.AC`. (`PSW.AC` is involved in implementing base-ten arithmetic.)

The obvious approach to the nightmare of executing such an instruction is to microcode the instruction set and execute it on a conceptually simpler machine. Unfortunately, this would lead to a very slow implementation with many machine cycles per instruction execution. Therefore, rather than microcoding instructions with many implicit operands, we decided to have special-purpose channels for implicit operands. This allows such an instruction to execute in one cycle, and reduces dependence on general-purpose buses, which have large fanouts and hence high energy cost.

Most of the design effort in the Lutonium has gone into assuring good Et^2-efficiency. For instance, the Lutonium implementation is highly pipelined for speed, but it is still nonspeculative: the instruction-fetch unit only keeps filling the pipeline as long as it knows that the instructions are definitely going to be executed, and although branches are executed in one cycle, that cycle is "stretched." We took great pains to minimize switching activity: no register or execution unit receives control unless it will process data for a given instruction (hence it has no switching circuit nodes unless it is used on a particular instruction), and interrupts and pins only cause switching when accessed by software or an input pin switches. We also localize activity as much as possible: special registers (`SP`, `PSW`, `B`, `DPTR`) have their own channels and function units (instead of using the main buses and units) for energy and time savings; e.g., the 16-bit `DPTR` can be incremented in one cycle without using the main buses at all. Also, infrequently used units do not add to the fanin and fanout of the main buses (see section VIII), for energy and time savings. The time savings would be lost if a clock were used, since we have improved the average-case energy and delay at the expense of somewhat worsening the worst-case delays.

Now let us consider all communications caused by the one-byte instruction `ADDC A, @Ri`. $RuptArb$ first sends 0 to $Fetch$, indicating the absence of an interrupt request. Then $Fetch$ sends the instruction byte to $Decode$. The following actions are specifically required by the `ADDC A, @Ri` instruction: $Decode$ sends control messages to PSW, ALU, A, and $RegFile$ Then

Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems (ASYNC'03)
1522-8681/03 $17.00 © 2003 **IEEE**

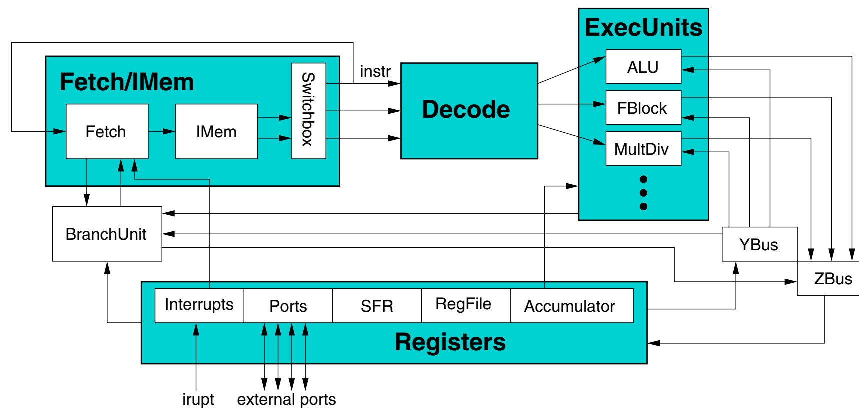

Fig. 2. Lutonium block diagram.

PSW sends C to ALU and RS to $RegFile$ (on special channels.) Then $RegFile$ accesses Ri and @Ri in sequence (without communicating outside of $RegFile$ in the middle of the sequence). Finally ALU receives its regfile operand through the general-purpose $DRBY$ bus, and its accumulator operand through a dedicated accumulator channel, and computes results for the accumulator and PSW, which are written back through special-purpose channels.

VI. Instruction Fetch Loop

The performance of the Lutonium is limited by that of the instruction fetch. We therefore carefully designed the fetch loop to optimize throughput. In this spirit, while 8051 instructions have variable length (one to three bytes), our first design decision was to allow two consecutive bytes of code to be fetched at a time from program memory, which is aligned along even addresses. Hence, if the last byte of a basic block happens to fall on an even address, we introduce some speculation in the fetching. The speed advantage of doing this more than compensates for the extra energy of fetching the unwanted odd byte; this scheme also reduces the average instruction-memory overhead since only one address need be decoded for each pair of bytes fetched. (Similarly, the preceding even byte is thrown away when a basic block starts on an odd address.)

The entire fetch loop is shown in figure 3. Along with the instruction memory ($IMem$), it comprises units that compute the next program counter ($Fetch$) and route the instruction bytes to other parts of the microcontroller as needed ($SwitchBox$). Another unit ($Decode$) is not on the critical loop but decodes instructions to send specialized information to each of the execution units. The instruction memory is described in section VII.

A. Instruction SwitchBox

$SwitchBox$, at the outputs of $IMem$, serves as both filter and router for the retrieved two bytes of program code. The first byte of every instruction, its opcode, is not necessarily stored at an even-byte address. An instruction's length, as well as whether it can change the program counter (i.e., whether it is a branch type of instruction) is encoded here. Often, one of the two bytes fetched from $IMem$ is an opcode, and it must be forwarded by $SwitchBox$ for decoding before the second byte can be either routed or discarded. $SwitchBox$ can either discard instruction bytes or forward them to $Fetch$, $Decode$ (on three channels, one for each of the possible byte positions in an 8051 instruction), or A (the accumulator; for code-read instructions). The $Fetch$ unit controls $Switchbox$. $Switchbox$ can route both bytes simultaneously as long as they go to different destinations.

B. Fetch Unit

To keep the main fetch loop as short as possible, the information from the current instruction required to compute the next program counter is decoded within the $Fetch$ unit itself; this information is the length of the instruction and whether or not it can change the program counter (i.e., whether or not it is a branch or jump)—it is enough to decode the first byte of the instruction (the opcode) to find out these facts. Information required by the execution units (including information encoded in the second and third bytes of an instruction) is not needed as quickly and is therefore left to the $Decode$. There is no feedback from the $Decode$ and execution units except on branches and interrupt guesses (see section XI-A).

The 16-bit program counter is incremented by two during every cycle of the $Fetch$; this incrementing takes place speculatively, whether or not the next sequential address is needed. It turns out that the Et^2 of the processor is thus improved over a design which would wait to check whether an interrupt or branch is disturbing the sequential program flow before starting the increment.

$Fetch$ also checks for interrupts after every instruction. If an interrupt occurs, the program counter pointing to the next sequential instruction is sent off for storage on the stack, and then $Fetch$ reads in the address of the interrupt handler and starts fetching instructions from there. There is no branch prediction in the Lutonium.

C. Decode Unit

The instruction decoder of the Lutonium ($Decode$) is much more complicated than that of the MiniMIPS. Part of the reason for this is that the 8051 instruction set is quirkier and more complicated. Also, early in the design process, we made the

Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems (ASYNC'03)
1522-8681/03 $17.00 © 2003 **IEEE**

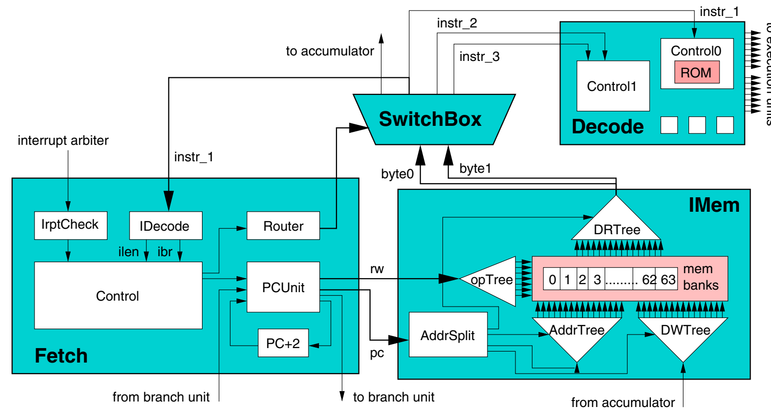

Fig. 3. Instruction fetch loop.

conscious decision to centralize all of the microcontroller's instruction decoding in one unit so as to reduce the size and number of communication channels traversing the chip. The only exception to this centralization is the above-mentioned special decoding that occurs in the *Fetch*.

Decode is decomposed into many processes, most of which only consume dynamic energy when their computations (decodings) are required. At a high level, one large process (*Control0*) decodes the opcode (the first byte) of every instruction while another large process (*Control1*) performs most of its computations only when the second and third bytes of an instruction are available (if they exist). *Control1* receives its inputs significantly later (about one cycle later) than *Control0*, and the design of *Decode* takes advantage of this fact to distribute computations and optimize performance.

For energy efficiency, nothing is computed or routed unless necessary. Applying this principle to *Decode*, we obtain the following property: not only does a `NOP` take less energy to decode than, for example, an `ADD` or an interrupt-register access, but an `ADD` also takes less energy to decode than an interrupt-register access.

D. Instruction Flow

As we have seen, when an instruction is fetched from program memory, one cycle is taken to decode the opcode before the rest of the instruction can be handled. This can cause hiccoughs in the instruction flow when two bytes have been fetched from memory but only one can be handled per cycle. The second byte is not thrown away and does not need to be re-fetched: as this is an asynchronous microcontroller, the data is simply not acknowledged and thus not allowed to disappear from the wires until the next cycle. (Of course, in the case of a branch or interrupt, it may be acknowledged only to be discarded.) As an example, table 4 illustrates fetching of two consecutive three-byte instructions, $I0$ and $I1$, where $Ix.y$ is the yth byte of instruction Ix. The columns cycle0, cycle1, etc., indicate the cycle in which the byte is finally used and acknowledged.

	cycle0	cycle1	cycle2	cycle3
IMemByte0	...	$I0.2$	$I1.1$	$I1.3$
IMemByte1	$I0.1$	$I0.3$	—	$I1.2$

Table 4. Lutonium fetch loop: Fetching two three-byte instructions in a row.

In summary, the instruction fetch loop of the Lutonium can fetch sequential instructions with the following throughputs:

- a sequence of one-byte instructions at 1 byte/cycle
- a sequence of two-byte instructions at 2 bytes/cycle
- a sequence of three-byte instructions at 1.5 bytes/cycle

The average throughput on a program consisting of random instructions, including branches, is 1.37 bytes/cycle.

VII. Memories

The Lutonium program memory holds a maximum of 8 kB of code and comprises 64 banks of 128 bytes each. The banks are arranged in a two-level tree with eight ways of branching per level. The leaves of the tree are 2 bytes (16 bits) by 64 rows. The memory is interleaved by using the three least significant bits of the address to control the highest level of branching and the three next least significant bits to control the second level of branching. This means that consecutive two-byte blocks of memory are stored in different banks, in order to maximize throughput when executing straightline code. The banks themselves can thus be optimized for forward latency, with little regard to cycle time: the slow cycle time of the banks will only matter when the two-byte blocks are accessed with a 128-byte stride. This design is similar to (but larger than) the I- and D-cache designs used in the MiniMIPS [7].

We used SPICE simulations to select dimensions for the distribution trees (whose leaves are the 64 banks) that result in the lowest possible forward latency for *IMem*: 1.8 ns. Meanwhile, the energy consumption of *IMem* is 80 pJ for every two bytes of code retrieved. (All figures are at 1.8 V in the TSMC 0.18-μm technology.)

VIII. Segmented Buses

The Lutonium buses must select and distribute data from and to many sources and destinations. For instance, $DRBY$, the main bus that sends direct (non-implicit) operands to execution units, must select from nine inputs. A circuit that selects one

1522-8681/03 $17.00 © 2003 IEEE

of nine inputs will be slower than one that selects one of two inputs. Now suppose that inputs 0..7 are selected with probability 1/16 each, but input 8 is selected with probability 1/2. One could make a circuit that selects input 8 faster than the others, but the speed advantage will be lost in a clocked circuit. This is because if one is using a clock, the difference in speed of the two types of inputs will typically be less than one quarter of the cycle time, so one is forced to make the decision of either ignoring the difference altogether, or adding an entire clock cycle's worth of latency for inputs 0..7 that only does 1/4 of its potential work, just so that the clock can run a little faster for input 8.

We are not using clocks, so the total delay for a tree of fixed-fanin selection stages is proportional to the number of stages. If each selection stage is binary, then each input can be associated with a binary string corresponding to the sequence of control signals needed to select it. Notice that the length of this string is proportional to the delay. To minimize expected delay, therefore, one minimizes expected string length. This problem has been studied extensively, and its solution is known as Huffman coding. The solution to our introductory example is that inputs 0..7 have binary strings 0000..0111 and input 8 has binary string 1. To select input 8, one sends just one control bit (1) to the main selection stage. To select inputs 0..7, one sends one bit (0) to the main selection stage, and three bits to subsequent stages. We can call this type of protocol a *segmented-bus control protocol*. Notice that only the control bit to the main stage is sent unconditionally.

In practice, we make two modifications to Huffman coding. First, there is no reason to impose the restriction of fixed fanin. Rather, we typically can have fanin between two and ten, with a different cost for each fanin. Secondly, we sometimes want to follow selection by immediate distribution, and we wish to combine the last selection stage with the first distribution stage.

For example, $DRBY$ is a process with multiple inputs and outputs. It sends an operand from a selected register to a selected execution unit. Control input $DRBY.I$ receives the instruction. An array $DRB.In[]$ ("Direct Read Bus", DRB) of input channels is needed to select the source register, and an array $Unit[].Y$ is needed to select the destination execution unit:

$DRBY \equiv$
$*[DRBY.I?i; DRB.In[yBank(i)]?y; Unit[where(i)].Y!y]$
$yBank(i) \in$ {"RegFile", "A", "B", "PSW", "DPL", "DPH", "SP", "RuptRegs", "PRDM"}
$where(i) \in$ {"Exchange", "FBlock", "ALU", "BitUnit", "PCL", "PCH", "DMem"}

In the $DRBY$ bus described above, the frequencies of the various inputs and outputs depend on which program the Lutonium is running; we used benchmarks (8051 programs found on the World Wide Web) to obtain these frequencies. $DRBY$ is decomposed into a tree of processes as shown in figure 5. Each of these processes requires at least one control channel. These control channels together implement $DRBY.I$. (A segmented-bus control protocol is used for $DRBY.I$.)

Since $RegFile$ is the most frequently used input and $Exchange$ is the most frequently used output, process $Main$ has been placed in the tree such that sending from $RegFile$ to $Exchange$ takes one stage. $Main$ is implementable as a simple stage of logic, and has the following specification:

$Main \equiv$
$*[DRBY.I.DRBMain?src, DRBY.I.YMain?dest;$
$\quad [src =$ "RegFile" $\longrightarrow DRB.RegFile?y$
$\quad \| src =$ "Other" $\longrightarrow Main.RegMerge?y$
$\quad];$
$\quad [dest =$ "Exchange" $\longrightarrow Exchange.Y!y$
$\quad \| dest =$ "Other" $\longrightarrow ExecSplit.Y!y$
$]]$

This gives us only one unconditionally required stage of latency; an additional stage is needed only if the input is uncommon or if the output is uncommon. Extremely uncommon inputs need yet another stage:

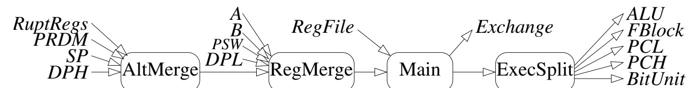

Figure 5. $DRBY$ bus decomposition, block diagram (data channels only).

Most of the time, only one or two stages are used, even though the worst case is four. Thus in the common case there are fewer data stages (and fewer control messages) than in any other case, leading to time and energy savings (compared to a traditional design, which would require all cases to finish in the same amount of time).

IX. Decode Design

The high-level behavior of $Decode$ is simple: it sends each instruction to each unit that needs it. However, for efficiency, it is not necessary to send the complete instruction (which may be many bytes) to every unit. Rather, there is a different protocol for sending the instruction to each unit. Each of these protocols was designed *after* designing the corresponding unit; this leads to energy efficiency because the resulting protocols do not require additional recoding or eventual discarding of information.

For each datapath process (such as $DRBY$), we determined a protocol (such as the segmented bus control protocols described above) for sending it an instruction and implement the protocol as outputs of whichever $Decode$ subprocesses have the necessary information.

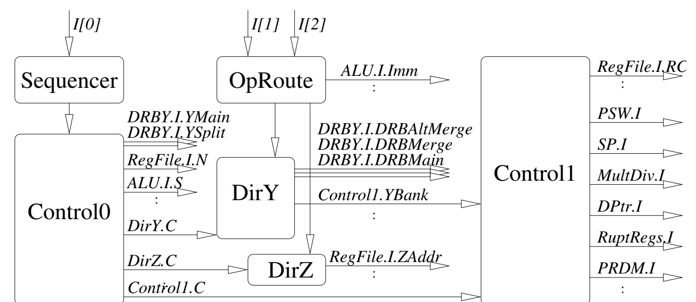

Figure 6. $Decode$ Block Diagram.

$Decode$ comprises $Sequencer$, $Control0$, $Control1$, and $OpRoute$. Only the operand router ($OpRoute$) has byte-wide data; the other units operate on smaller data (typically single-digit control information). Separating the large-data operations into $OpRoute$ in this way is energy efficient, because $OpRoute$ is very simple.

1522-8681/03 $17.00 © 2003 IEEE

A. Sequencer

Our original idea was that all instructions would run in one cycle; however, the `CALL` and `RETURN` instructions both use the main buses, stack pointer, and the register file twice. Rather than adding special sequencing hardware, which would be used rarely, in all these units, we chose to centralize this sequencing in the *Decode*'s *Sequencer* unit. *Sequencer* also inserts interrupt pseudo-calls and the interrupt guesses (see section XI-A) into the instruction stream. *Sequencer* consumes very little energy and adds minimal delay in the common case, which is when it simply passes along byte 0 of the instruction.

B. Decode.Control0

Opcode fields (i.e., information available from instruction byte 0 alone) are unconditionally computed (i.e., extracted, but not necessarily trivially) and sent to other units in a circuit called $MegaROM$, which has 18 single-digit output channels. The next stage consists of processes (*Filter*, *ExecSplit*, *BusSplit*, $Control1$) that forward a message to an execution-unit controller when and only when that unit is involved in executing an instruction. Each execution-unit controller (e.g., $ALU0$) does unit-specific decoding only in the case when it receives a message. Note that only certain units (those whose operations can be determined from instruction byte 0 alone) can be controlled directly from Control0.

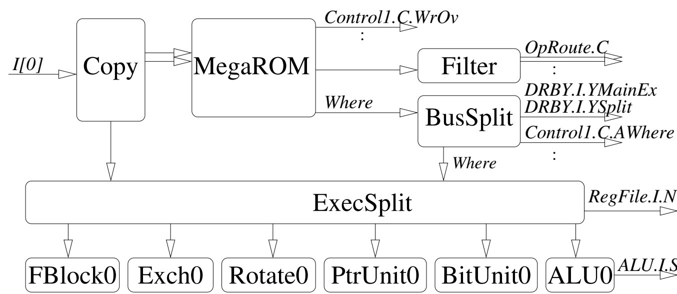

Figure 7. Decode.Control0 Block Diagram.

C. Decode.Control1

The decoding problem is complicated by the fact that the registers must receive instructions (not just addresses) because special instructions implicitly access registers, and we do not wish to use general-purpose buses to handle these types of accesses. Since some special registers (like PSW) are used on every cycle in some kinds of code (in, e.g., a sequence of `ADD` instructions), it is important to make sure that they can be read concurrently with the reads of other instruction operands. Each special-register–control function of $Control1$ can be activated in one of three ways:

1) direct y access
2) direct z access
3) special operation

Information about the direct accesses is received on the channels $Control1.YBank$ and $Control1.ZBank$, from the processes $Decode.DirY$ and $Decode.DirZ$. Ideally, we would like a "magic process" with 12 inputs that determines which special units to activate and only activates those units. Unfortunately, this is too large for a single PCHB stage. Therefore we must decompose $Decode.Control1$. We can only reduce the number of inputs per decoding process by having a process dedicated to each special function. These processes must receive all their inputs unconditionally: making these inputs conditional is difficult because the bit determining whether PSW (for example) is used by an instruction cannot be computed from just part of the instruction; it depends on y, z, and opcode fields.

Thus the y and z register identifiers must be copied to all control blocks. For instance, $PSWCon1$, the control block for PSW, must be told whether or not every instruction has a direct PSW access. When PSW is not used, $PSWCon1$ must still be sent NOPs as placeholders; this is unfortunate from the point of view of energy efficiency, but the other solution would be to un-pipeline the design, which would cost even more because of the loss in throughput.

Fortunately, $RegFile$, PSW, and A are the only registers and special function units that are used with considerable frequency in compiled code. Hence we can make the common case more efficient by making only three copies of the y/z operand identifiers, and a fourth "special" reserve copy. $RegFileCon1$, $PSWCon1$, and $ACon1$ generate control for their respective units if necessary, while *Special* forwards the control on to further stages if necessary (which it rarely is). Thus we keep the number of copies small in the common case. SP is much more commonly used than other registers controlled by *Special*; hence the commands for these other registers are further filtered by the process *ReallySpecial*. Figure 8 illustrates the $Control1$ decomposition.

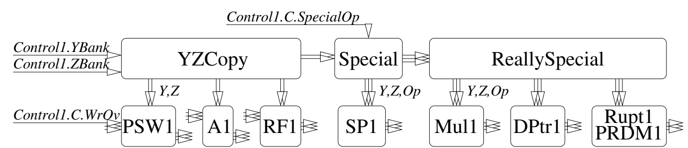

Figure 8. Decode.Control1 Block Diagram.

X. Peripherals

The 8051 architecture specifies mechanisms that permit the programmer to set and examine individual pins; this allows a wide variety of protocols to be implemented in software. The architecture also allows pins to be hardware-configured to be used in an external memory interface, to be used for gathering interrupts from external sources, to be used as counter inputs, or to be used as a serial interface. We implemented all of these except the serial interface, since that can be straightforwardly implemented in software.

There is a special counter input "CLOCK" that cripples a standard 8051, in the sense that "CLOCK" cannot be stopped or slowed down without seriously affecting the performance of the CPU core. Our design does not have this problem. In fact the peripheral interface is decoupled from the CPU to the extent that we obtain a list of desirable properties that would be very hard to achieve in a clocked system.

Our handling of the CLOCK input has repercussions in two areas: first, the "deep sleep" mode of the Lutonium is much improved over that of clocked 8051s; and secondly, the switching activity in the chip is much less dependent on the CLOCK rate.

1522-8681/03 $17.00 © 2003 **IEEE**

The Lutonium has a deep sleep mode in which almost all switching activity ceases. The important benefit of the asynchronous implementation is that the Lutonium can wake up out of the deep sleep mode *instantly*—there is no wakeup time during which crystal oscillators must be started, etc.[1] The only part of the processor that can operate during deep sleep is the counters; the counters can continue to count even while the CPU is sleeping; the CPU only wakes up if the counter overflows and the interrupt checking for such an overflow is enabled. The software SLEEP sequence (see section XI-B), by which the processor enters deep sleep, is the only way that peripheral activity can delay the CPU for an arbitrary amount of time.

The weaker synchronization between the CPU core and the pins has further advantages: the peripheral interface never has switching activity if pins are not switching and the CPU is not accessing it; CPU execution speed is affected by the peripheral interface only when peripheral functions are executed by the CPU; and if a single counter is enabled and it does not overflow, the CLOCK input does not cause any switching activity besides the incrementing of that counter.

In addition, we made several improvements to the peripheral interface that would benefit either synchronous or asyncronous designs. Had we not improved these interfaces, their bad effects in practical applications might have dwarfed the improvements enabled by our asynchronous design. For example, passive pullups can burn enough power to make all other power optimization pointless; therefore we added direction registers, so that the external-memory–and–pin interface does not require passive pullups. We also made several changes to the SRAM interface: we added a demultiplexed SRAM mode, so that an external SRAM can be added without also adding an external register chip, and we added a Fast Read mode, so that an external SRAM read does not require three cycles. Finally, we added some miscellaneous features: we exposed internal timer registers and interrupt priority registers to help with debugging, and we added an internal oscillator (some commercial 8051s have this). The extra features are disabled by default, so that the programmer that does not expect them gets standard 8051 behavior.

Each peripheral feature is specified (in the 8051 architecture) as a sequence of actions on pins. Pins are wires, and our circuit design style is glitch free; hence it does not allow arbitrary actions on wires. Hence we separate the "special" circuits that must connect to wires, and require nondeterministic synchronizer circuits; it turns out we need several instances of only a small number of such circuits:

1) pulse synchronizers (send "0" for each detected pulse)
2) port modules (read and write current wire value)

These circuits are specially designed so that they are able to handle arbitrary input waveforms safely.

The CPU interface with each of the above circuits consists of standard four-phase channels; therefore each peripheral function can be implemented as a sequence of commands, using our standard design style. The modules in the CPU that communicate with the peripheral interface are $RuptRegs$ (interrupts) and $PRDM$ (PortRegs/DMem).

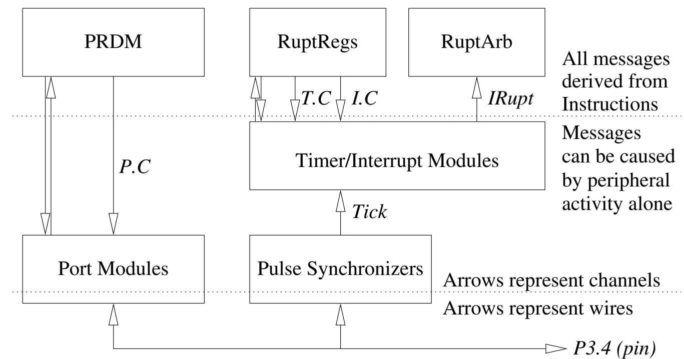

Figure 9. Lutonium Peripherals Block Diagram.

XI. Sleep Mode and Interrupt mechanisms

A. Independence of Execution from Interrupts

The 8051 architecture suggests that registers holding the interrupt request condition be evaluated after every instruction. In fact the result of the evaluation is almost always that there is no active interrupt.

The Lutonium uses a distributed interrupt mechanism. It would be too expensive to check all the interrupt-enable registers on every cycle. Therefore, we implement part of the interrupt mechanism in the $RuptArb$, which is part of the instruction-fetch unit and fetch loop. The $RuptArb$ generates an "interrupt guess" (IG), which is an approximation (see below) to the true interrupt-request status. If IG is **true**, the interrupt-control registers in $RuptRegs$ will be queried as to whether the interrupt should be taken or not.

IG can be **true** only in response to a message on the $IRUPT$ channel, which is only generated by an interrupt module if the corresponding interrupt is enabled. All we need to do on every instruction cycle is to probe the $IRUPT$ channel so that $IMem$ knows whether to compute an ordinary PC or an interrupt PC. The process accomplishing this is as follows in CHP:

$$RuptArb \equiv *[[\ \overline{IRUPT} \rightarrow IRUPT, IMem.IG!\mathbf{true} \ | \neg\overline{IRUPT} \rightarrow IMem.IG!\mathbf{false} \ ]]$$

Thus we avoid evaluating complicated interrupt conditions on every instruction cycle and reduce overall power consumption.

There are hazards when interrupts are enabled or disabled. The former hazard (which is the need to take an interrupt right after it is enabled, even if the request happened a long time ago and was discarded) is solved by regenerating $IRUPT$ messages when interrupts are enabled. The latter hazard is that interrupt requests that were generated before an interrupt-disable instruction may not be discovered in the $IRUPT$ channel until after an interrupt-disable instruction was already executed; hence they must semantically follow *after* the interrupt-disable instruction (i.e. there should be no interrupt request). Extra $IRUPT$ messages caused by the solution to the first hazard, and by the latter hazard, are masked by the fact that IG is, after all, only a guess: interrupt-disable actions and taken interrupts are sequenced at the $RuptRegs$, which leads to an implementation with the proper sequential semantics.

[1] At least one other 8051 implementation, the Dallas DS89C420 "ultra high-speed microcontroller," has a special wakeup system with two "clock gears": the DS89C420 wakes up out of deep sleep by using a "low gear" ring oscillator with a speed nominally about one third the normal clock speed for some microseconds; after that, it switches to the off-chip–clock "high gear" [9].

1522-8681/03 $17.00 © 2003 IEEE

B. Deep Sleep

If one uses a clock, the time required to start the clock generator on wake-up from deep sleep (a mode with no switching nodes) is often so large that the deep sleep mode cannot be used, or programmers resort to using a pseudo-sleep mode which does not halt all switching activity (e.g., an oscillator and a few gates continue to run).

The lack of a usable deep sleep means that there are long idle periods during which the clock (and everything driven by it) are consuming power. Even if this is a small amount of power it could dominate the overall energy use, owing to its duration. To solve this problem, we present the design of *deep sleep with instant wakeup* for our asynchronous 8051.

First, we observe that if the $RuptArb$ process goes to sleep, instruction fetching stops, and hence execution stops. This is what sleep mode should do. Counters, if enabled, can continue to run without instruction execution, and this is desired—it should be possible to exit sleep mode through counter overflow. Other than the desired enabled counters, there will be no switching activity if instruction execution is stopped. We can easily add a message to the $IRUPT$ channel that makes $RuptArb$ go to sleep:

$$
\begin{array}{l}
RuptArb \equiv \\
*[[\overline{IRUPT} \longrightarrow \\
\quad [\overline{IRUPT = \text{“sleep”}} \longrightarrow IRUPT; \ \ [\overline{IRUPT}] \\
\quad \square \overline{IRUPT = \text{“other”}} \longrightarrow IRUPT; \ \ IMem.IG!\mathbf{true} \\
\quad] \\
\ | \neg\overline{IRUPT} \longrightarrow IMem.IG! \ \mathbf{false} \\
\ ]]
\end{array}
$$

We expose to the programmer a “sleep” instruction which enters sleep mode, and a SLEEP sequence, which atomically enables interrupts on entering sleep and disabling them on exiting sleep. This allows the proper implementation of condition variables (which is hard to do on many microcontrolllers).

To design the SLEEP sequence properly, we must deal with a problem that results from the CPU's being pipelined. Since we do not have a mechanism for discarding speculative instructions, we do not know how many instructions *after* a “sleep”-generating instruction will have executed when the CPU goes to sleep. This is a problem, because a test-and-set loop (e.g., to implement mutex locking) might evaluate its exit condition prematurely, before actually going to sleep, and then incorrectly return to sleep (possibly forever) as soon as it is awakened.

We solve the problem of post-“sleep” instruction execution by requiring the instructions containing the sleep instruction to make up an infinite loop. The loop is exited only after an interrupt request executes and returns. In summary, the “sleep” instruction must perform the following actions atomically:

1) enable interrupts (because they must be disabled before “sleep,” to avoid falling asleep immediately after returning from the interrupt handler that handles the interrupt that is supposed to wake up the processor).
2) notify the interrupt-return-address-saving-unit that it should modify the saved PC so that the return from the interrupt handler that executes when the processor wakes up jumps past the infinite loop.
3) send “sleep” message to RuptArb.

We chose `MOV SLP,A` (with `SLP=CFH`) as the “sleep” instruction. We give the following sequence the mnemonic `SLEEP`:

```
{precondition: interrupts disabled here}
        MOV SLP,A
loop:   SJMP loop
        CLR IE.EA
{postcondition: interrupts disabled here}
```

The SLEEP sequence results in the following sequence of actions: (1) atomically enable interrupts and go to sleep; (2) wake up as soon as at least one interrupt has been handled; and (3) disable interrupts. In fact our design guarantees that the instruction after an interrupt return is executed (as demanded by the 8051 architecture), so SLEEP will catch exactly one interrupt (assuming only one level of interrupts is enabled).

C. Spinning on a variable efficiently using Deep Sleep

Suppose an interrupt routine sets a bit `myFlag` on success. The following code waits for the interrupt routine to run successfully, while consuming zero dynamic power in the meantime:

```
        CLR IE.EA
        SJMP test
sleep:  SLEEP
test:   JNB myFlag, sleep
        SETB IE.EA
```

XII. Performance Estimates and Comparisons

The performances are for the Lutonium-18 (0.18-μm) prototype implementation. This implementation will use the TSMC SCN018 process offered by MOSIS. It is a 0.18-μm CMOS process with a nominal voltage of 1.8 V and threshold voltages of 0.4 V and 0.5 V. The performance estimates given here have been obtained by two methods. First, we simulated several standard instructions (both electrical simulation in SPICE for the memories and digital simulations augmented with estimates of energy and delay per transition). Secondly, we extrapolated the MIPS performance to the Lutonium taking into account (1) the architecture differences (22 transitions/cycle vs. 18 transitions/cycle, fetch-loop differences), (2) technology scaling, (3) transistor-sizing differences, (4) the datapath width difference (32 vs. 8 bits), (5) the cache/memory difference (32 vs. 16 bits). The results of the two approaches agree closely.

The performances are summarized in figure 11. Energy breakdowns for an ADD and a NOP are shown in figure 10. NOP is interesting because it exercises the most energy-intensive functions (fetching and operand decoding) that are required of any instruction. (There are no special optimizations for NOP, since it is not used in ordinary programs.) Most noticeable is the high energy consumption of instruction decoding compared to the MIPS. As in the MIPS, we observe that the execution of the instruction proper accounts for only a small fraction of the total energy consumption.

In order to compare our design to three existing implementations of the 8051 which were all three fabricated in 0.5-μm

1522-8681/03 $17.00 © 2003 IEEE

CMOS technology, we estimated the performance of the Lutonium for a hypothetical Lutonium-50 implementation in a 0.5-μm CMOS process. At the nominal 3.3 V, the Lutonium-50 would run at 100 MIPS and would consume energy corresponding to 600 MIPS/W.

In a comparable technology, a synchronous implementation by Philips runs at 4 MIPS and 100 MIPS/W. An asynchronous implementation by Philips runs at 4 MIPS and 444 MIPS/W. The synchronous Dallas DS89C420, called "ultra-high-speed," runs at 33–50 MIPS and 100 MIPS/W.

In terms of Et^2, let us rank the four designs according to increasing Et^2 figures. The Lutonium-50 ranks first with $1.7 \cdot 10^{-25}$ Js^2, followed by the Dallas at $4.0 \cdot 10^{-24}$ Js^2, the Philips asynchronous 8051 at $1.4 \cdot 10^{-22}$ Js^2, and finally the Philips synchronous 8051 at $6.3 \cdot 10^{-22}$ Js^2. Hence, the Lutonium outperforms its best competitor by almost a factor of 30 in Et^2.

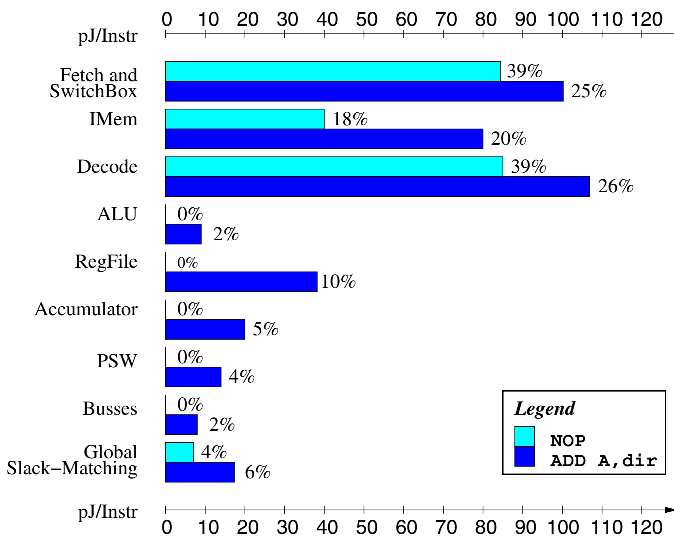

Figure 10. Energy breakdowns for instructions.

1.8 V	200 MIPS	100.0 mW	500 pJ/in	1800 MIPS/W
1.1 V	100 MIPS	20.7 mW	207 pJ/in	4830 MIPS/W
0.9 V	66 MIPS	9.2 mW	139 pJ/in	7200 MIPS/W
0.8 V	48 MIPS	4.4 mW	92 pJ/in	10900 MIPS/W
0.5 V	4 MIPS	170 μW	43 pJ/in	23000 MIPS/W

Table 11. Performance from low-level simulation (conservative!).

XIII. Conclusion

Our past experience in designing complex asynchronous microprocessors gives us confidence that the performance figures of the fabricated prototype will be close to our estimate, if not better.

If such is the case, this experiment will demonstrate that the Caltech fine-grain–pipeline design-style can indeed produce energy-efficient systems, provided that a number of parameters are adjusted carefully: transistor sizing, slack-matching buffers, and process decomposition. In particular, the decomposition of an initial sequential process into a network of PCHB-implementable modules is still a trial-and-error procedure leading to vastly different results in the hands of different designers. We are developing systematic algorithms to improve the method [11].

As the supply voltage keeps decreasing with feature size, the designer's freedom to exchange throughput against energy, which is at the core of the Et^2 approach, is reduced. Soon, the designer may end up with an excess throughput that cannot be converted into needed energy savings. Nevertheless, the Lutonium experiment confirmed our conviction that the Et^2 metric currently provides the best approach to energy-efficient designs.

Acknowledgments

The research described in this paper was sponsored by the Defense Advanced Research Projects Agency, and monitored by the Air Force Office of Scientific Research. Acknowledgment is due to Abe Ankumah, whose Caltech Senior Thesis on the design of an energy-efficient asynchronous 8051 provided us with valuable information at the start of the project [12].

References

[1] Alain J. Martin. Towards an Energy Complexity of Computation. *Information Processing Letters*, 77, 2001.
[2] Alain J. Martin, Steven M. Burns, Tak-Kwan Lee, Drazen Borkovic, and Pieter J. Hazewindus. The design of an asynchronous microprocessor. In Charles L. Seitz, ed., *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pp. 351–373. Cambridge, Mass.: MIT Press, 1991.
[3] Paul I. Pénzes - *Energy-delay efficiency in Asynchronous Circuits*, Ph.D. Thesis, California Institute of Technology, June 2002
[4] Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nyström, Paul Penzes, Robert Southworth, Uri Cummings and Tak Kwan Lee. The Design of an Asynchronous MIPS R3000 Microprocessor. *Proceedings of the 17th Conference on Advanced Research in VLSI.* Los Alamitos, Calif.: IEEE Computer Society Press, pp. 164–181, 1997.
[5] Alain J. Martin. Synthesis of Asynchronous VLSI Circuits. *Formal Methods for VLSI Design*, ed. J. Staunstrup, North-Holland, 1990.
[6] José A. Tierno, A.J. Martin. Low-Energy Asynchronous Memory Design. *Proceedings of International Symposium on Advanced Research in Asynchronous Circuits and Systems*, IEEE Computer Society Press, 176–185, 1994.
[7] Mika Nyström, Andrew Lines, and Alain J. Martin. A Pipelined Asynchronous Cache System. In preparation.
[8] Alain J. Martin, Mika Nyström, and Paul I. Pénzes. Et^2: A Metric for Time and Energy Efficiency of Computation. In *Power-Aware Computing,* R. Melhem and R. Graybill, eds. Boston, Mass.: Kluwer Academic Publishers, 2002.
[9] *DS89C420 Ultra-High-Speed Microcontroller.* Maxim Integrated Products, 2000–2002.
[10] Hans van Gageldonk, Kees van Berkel, Ad Peeters, Daniel Baumann, Daniel Gloor, and Gerhard Stegmann. An Asynchronous Low-Power 80C51 Microcontroller. In *Proceedings of the Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC'98),* San Diego, Calif., 1998. Los Alamitos, Calif.: IEEE Computer Society Press, 1998.
[11] Catherine G. Wong and Alain J. Martin. High-Level Synthesis of Asynchronous Systems for Energy Efficiency. Submitted for publication, 2002.
[12] Abraham Ankumah. Designing an Energy-Efficient Asynchronous 80C51 Microcontroller. Caltech Senior Thesis, 2001.

1522-8681/03 $17.00 © 2003 IEEE