The iCORE^{tm} 520 MHz Synthesizable CPU Core

Nick Richardson, Lun Bin Huang, Razak Hossain, Tommy Zounes, Naresh Soni
STMicroelectronics, Inc
Advanced Designs, Central R&D
4690 Executive Dr. Ste. 200, San Diego, CA 92121
(858) 452-7715 x 270
nick.richardson@st.com; lun-bin.huang@st.com; razak.hossain@st.com; tommy.zounes@st.com; naresh.soni@st.com

Julian Lewis
STMicroelectronics, Ltd.
Advanced Designs, Central R&D
1000 Aztec West - Almondsbury
BRISTOL, BS32 4SQ - UK
+44 145 446 2647
julian.lewis@st.com

ABSTRACT
This paper describes a new implementation of the ST20-C2 CPU architecture. The design involves an eight-stage pipeline with hardware support to execute up to three instructions in a cycle. Branch prediction is based on a 2-bit predictor scheme with a 1024-entry Branch History Table and a 64 entry Branch Target Buffer and a 4-entry Return Stack. The implementation of all blocks in the processor was based on synthesized logic generation and automatic place and route. The full design of the CPU from microarchitectural investigations to layout required approximately 8-man years. The CPU core, without the caches, has an area of approximately 1.5 mm^2 in a 6-metal 0.18m CMOS process. The design operates up to 520 MHz at 1.8V, among the highest reported speeds for a synthesized CPU core [1].

Categories and Subject Descriptors
M.2.4 [**Design Methodologies**]: High performance ASIC design*: Optimizing microarchitecture to layout.*

General Terms: Performance, Design.

Keywords: Synthesis, ASIC, CPU, embedded, st20, pipeline, cache, branch-prediction, high-frequency, microarchitecture

1. INTRODUCTION
The goal of the iCORE project was to show that a synthesizable core could be capable of execution speeds typically only achievable by complex and time-consuming custom designs. This was to be done by using a customized ASIC design flow to create a very high performance version of STMicroelectronics' ST20-C2 embedded CPU core.

The ST20-C2 architecture is a superset of the acclaimed inmos transputer, which was first introduced in 1984 (inmos was acquired by STMicroelectronics in 1989). The ST20-C2 added an exception mechanism (interrupts and traps), extensive software debug capability and improved support for embedded systems. It is now used in high-volume consumer applications such as chips for set-top boxes.

DAC 2002, June 10-14, 2002, New Orleans, Louisiana, USA.

For the project to be considered successful, however, we had to demonstrate optimization in all the following areas, in approximate order of importance:

- Frequency
- Execution efficiency (IPC)
- Portability
- Core size
- Power consumption

2. OPTIMIZING THE MICROARCHITECTURE
To achieve good instruction-execution efficiency without excessive design complexity, previous implementations of the ST20-C2 architecture employed relatively short pipelines to reduce both branch-delays and operand/result feedback penalties, thus producing good IPC counts over a wide range of applications. In the case of iCORE, however, the aggressive frequency target dictated the use of a longer pipeline in order to minimize the stage-to-stage combinatorial delays. The problem was how to add the extra pipeline stages without decreasing instruction execution efficiency, since there would be little point in increasing raw clock frequency at the expense of IPC. Following analysis using a C-based performance model, a relatively conventional pipeline structure was chosen, but one that had some important variations targeted at optimizing instruction flow for the unique ST20-C2 architecture.

The pipeline microarchitecture is shown in Figure 1. It comprises four separate units of two pipeline stages each. The **IFU** (Instruction Fetch Unit) comprises the IF1 and IF2 pipeline stages and is responsible for fetching instructions from the instruction cache and performing branch predictions. The **IDU** (Instruction Decode Unit) comprises the ID1 and ID2 pipeline stages and is responsible for decoding instructions, generating operand addresses, renaming operand-stack registers and checking operand/result dependencies, as well as maintaining the program counter, known in this architecture as the Instruction Pointer (IPTR), and a Local Workspace Pointer (WPTR). The **OFU** (Operand Fetch Unit) comprises the OF1 and OF2 pipeline stages and is responsible for fetching operands from the data cache, detecting load/ store dependencies, and aligning and merging data supplied by the cache and/or buffered stores and data-forwarding buses. Finally, the **EXU** (Execute Unit) comprises the EXE and WBK pipeline stages and performs all arithmetic operations other than address generation, and stages results for writing back into the Register File Unit (RFU) or memory.

The RFU contains the 3-deep register-stack, conceptually organized as a FIFO, comprising the *A*, *B*, and *C* registers (in top-to-bottom order). In practice, it is implemented using a high-speed multi-port register file. The memory-write interface is coupled with a store buffer (SBF) that temporarily holds data while waiting for access to the data cache.

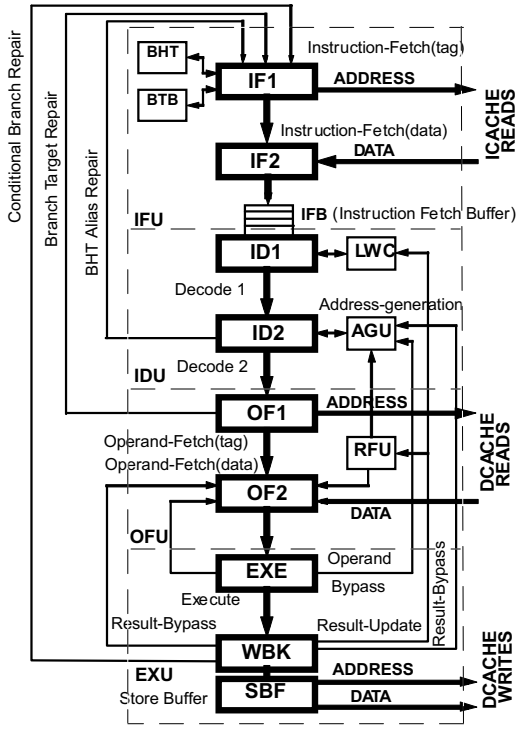

Figure 1. iCORE Pipeline Structure

The instruction-fetch portion of the pipeline (the IFU) is coupled to the execution portion of the pipeline (the IDU, OFU, and EXU) via a 12-byte Instruction Fetch Buffer (IFB).

To ensure that the pipeline is not continually stalled waiting for memory operands, two in-line operand caches are tightly integrated into the pipeline, both capable of being accessed in a single pipeline throw.

The first operand cache is for local variables, which are accessed relative to a Local Workspace Pointer (WPTR). It is called the Local Workspace Cache (LWC) and accelerates references to the first 16 words in the current local workspace, as pointed to by WPTR. The size of the LWC was determined by the results of running a number of benchmark programs on the C-based statistical performance model. It is placed early in the pipeline (in the ID1 stage) so that local variables used for non-local address calculations can be supplied to the Address Generation Unit (AGU) in the next pipeline stage (ID2) for early calculation of the full non-local address.

The second operand cache is an 8 KB in-line data cache that accelerates references to non-local variables, which can be anywhere in the 4 GB address space, as well as local variables that miss the LWC. It uses non-local addresses calculated by the AGU in the ID2 stage, and supplies operands to the Execute Unit, thus occupying the two pipeline stages before the EXU. For critical path optimization, its tag RAM and data RAM are placed in two different pipeline stages. Accessing the tag RAM before the data RAM eliminates the cache's set-associative output multiplexing that would have been needed if the tag and data RAMs were placed in the same stage, and also allows cache reads and writes to share the same control flow, thus simplifying the design.

Together, these two highly integrated operand caches are complementary to the ST20-C2 architecture's *A*, *B*, and *C* registers, effectively acting like a very large register file.

2.1 Instruction Folding

An important effect of the in-line operand caches is to increase the opportunities for instruction folding. This is an instruction decoding technique that combines two or more machine instructions into a single-throw pipeline operation. Since the ST20-C2 has a stack-based architecture, most of its instructions specify very simple operations such as loading and storing operands to and from the top of the 3-deep register-stack, or performing arithmetic operations on operands already loaded onto the register-stack, but for simplicity, memory and ALU operations are never combined in the same instruction. Without such compound instructions that allow memory and arithmetic operations to occur as single-slot pipeline operations, it would not be possible to take advantage of iCORE's in-line memory structure. With folding, however, up to three successive instructions can be merged into one operation that occupies a single execution slot and which fully uses iCORE's pipeline resources.

As an example, take the following very common three-instruction sequence:

ldl <n>
ldnl <m>
add

The first instruction, **load local** (*ldl*) loads from memory a variable that is addressed by an offset <n> from the current value of the Local Workspace Pointer (WPTR) and pushes it onto the top of the register-stack. The second instruction, **load non-local** (*ldnl*) loads from memory a variable that is addressed by an offset <m> from the value on top of the register-stack (in this case the value that was just placed there by the *ldl* instruction). Finally the *add* instruction adds the top two values on the register-stack (*A*, *B*) and pushes the result back onto it.

Without folding, this instruction sequence would occupy three distinct pipeline slots, and could be subject to stalling due to data-dependency between the *ldl* and the *ldnl* instructions. However, with folding and iCORE's in-line memory structure they can be executed in a single pipeline slot, even though they require two memory reads. To understand how this is done, consider the stage- by-stage execution:

IF1: Instruction fetch (instruction cache tag access)

IF2: Instruction fetch (instruction cache data RAM access, load IFB)

ID1: execute the *ldl* by reading local variable from LWC at offset <n>; issue *ldl*, *ldnl*, *add* from IFB as one operation if *ldl* hits the LWC

ID2: Using the AGU, generate address to be used by the *ldnl* instruction by adding the local operand obtained by the *ldl* in ID1 to the offset <m>.

OF1: Fetch data requested by *ldnl* from data cache (tag RAM)

OF2: Fetch data requested by *ldnl* from the data cache (data RAM); fetch second data operand from register file unit (pop top of stack)

EXE: Add the data obtained by the *ldnl* in OF2 to the second operand from register-stack.

WBK: Push result onto the top of the register-stack.

Note how the three folded instructions are executed separately by different pipeline stages (the *ldl* in ID1, the *ldnl* in ID2, OF1, OF2, and the *add* in EXE, WBK), but because only one pipeline stage at a time is occupied, the three instructions are apparently executed in a single clock cycle.

There are many such folding sequences supported by iCORE, but they all follow the same principle as described above.

2.2 Data Forwarding

In a deeply pipelined design, it is important to mitigate the effects of delays caused by pipeline latencies. One significant source of latency-based performance degradation is caused by data dependency between successive instructions. If the dependency is a true one (such as when an arithmetic operation uses the result of another instruction immediately preceding it) then there is no way of eliminating it, although microarchitecture techniques can be used to reduce the pipeline stalls it causes. This is generally done by providing data forwarding paths between potentially inter-dependent pipeline stages. In iCORE this includes data bypass paths from WBK to OF2, EXE to OF2, WBK to ID2, and EXE to ID2, as shown in Figure 1.

2.3 Branch Prediction

iCORE's relatively long pipeline gives rise to the danger of branches causing significant performance degradation compared to previous ST20-C2 implementations. After performance simulations showed that the effect of the longer pipeline on some benchmarks was significant, a low-cost branch prediction scheme was incorporated into iCORE's microarchitecture.

In a machine without branch prediction, the source of most branch penalties is two-fold:

Firstly, the target address of a taken branch or other control-transfer instruction must be determined. This cannot be done until after the branch has been decoded, since it must first be identified and then (usually) an arithmetic operation performed, such as adding an offset to the current Instruction Pointer value, to obtain the address of the next valid instruction in the program flow. By the time its target address can be applied to the instruction cache to make it start fetching instructions from the new destination, several instructions have been fetched from the wrong address. These must be cancelled, creating a penalty of several clock cycles.

Secondly, conditional branches, i.e. those that are dependent on the results of previous instructions to determine their direction, usually must progress all the way to the end of the pipeline before the condition on which their action depends is resolved. Only then can instruction fetches from a new target address proceed, causing an even greater branch penalty than that incurred by the target address calculation alone.

iCORE implements a branch prediction mechanism that reduces the penalties in both these cases. A two-bit predictor scheme is used to predict branch and subroutine-call instruction behaviour (taken or not taken), while a Branch Target Buffer is used to predict target addresses for taken branch and call instructions. Also, a 4-deep return stack is used to predict the special case of *ret* instruction return addresses.

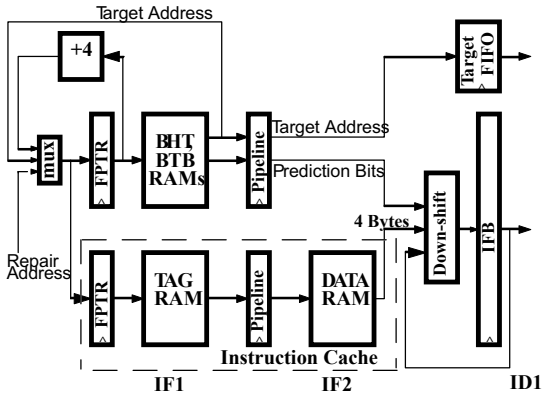

Figure 2. Branch Prediction Logic

The prediction scheme used by iCORE is shown in Figure 2. It employs a 1024-entry Branch History Table (BHT), each entry of which contains a two-bit predictor that implements a branch-hysteresis algorithm (weakly/strongly taken/not-taken). The BHT keeps the most current branching history of a set of instruction cache words. Predicted fetch-addresses (i.e. the target addresses of predicted-taken branches) are kept in a 64-entry Branch Target Buffer (BTB).

Whenever a new 4-byte word is fetched from the instruction cache, the instruction address that is used to index the cache also indexes the BHT and BTB. The retrieved information from the BHT is used to predict whether there is a taken branch at the current instruction address. If a taken branch is predicted, the address retrieved from the BTB is loaded into the Instruction- Fetch Pointer and used to fetch the next instruction word. Otherwise, the sequential value of the Instruction-Fetch Pointer is used. All this takes place in the very first pipeline stage, ID1, so that predicted target addresses can be applied immediately to the instruction cache, and the performance impact of successfully predicted branches is negligible.

As branches are predicted, mispredictions may occur. One type of misprediction is caused by the cost-saving measures of using only the low-order bits of the instruction-fetch address to index BHT and BTB entries while ignoring the upper address bits, and by providing only one BHT/BTB entry per 4-byte cache word, even though there could be more than one branch in a word. These factors can cause a single BHT or BTB entry to be shared by many fetched instructions, even though the entry is likely to be correct only for the one instruction that was originally used to update it; the instructions at other fetch addresses that share the entry may get wrong predictions or wrong target addresses, or both.

Other sources of misprediction include mispredicted direction of conditional branches, and incorrect return addresses. Mispredictions are detected and resolved in later pipeline stages. The instruction decoder in the ID1 stage can detect many mispredictions caused by BHT aliasing. For instance, if a *taken* condition is predicted for an instruction that is not a branch, or an *untaken* condition is predicted for an unconditional branch, the prediction is clearly in error and a repair operation can start immediately with only a small penalty. To check predicted targets, an adder in the OF1 stage calculates the correct target address of a taken branch and compares it with the

predicted one (which is fed forward from the BTB). If there is a mismatch due to a missing aliased BTB entry, then recovery can start from OF1. Finally, a conditional branch must wait until it reaches the EXE stage before the condition on which its action was determined can be checked, and thus has the longest repair time when incorrectly predicted.

The steps needed to repair a branch depend on the type of misprediction. In the case where BHT aliasing causes a non-branch instruction to be taken, then all instructions fetched after that instruction are flushed from the pipeline, and instruction fetching is restarted from the *source* address of that instruction to ensure that it is correctly fetched, while the BHT is updated to indicate *not taken.* Conversely, in the case where a target address of a taken branch is incorrectly predicted, then instructions from the bad target are flushed from the pipeline and instruction fetching is restarted from the correct *target* address computed in the OF1 stage, while the BTB is updated with the new target address.

Finally, repairs needed to correct a mispredicted *conditional* branch flush all subsequent instructions from the pipeline and begin fetching again either from the target address of the branch or the next-sequential address of the branch, depending on whether the branch is taken or not.

2.4 Cache Subsystem

Features of the cache controller's design were focused on the need for simplicity (thereby enabling high frequency) and the requirement for very tight coupling to the CPU pipeline. These requirements resulted in a two-stage pipelined approach, the first stage of which handled the tag access and tag comparison, and the second stage that handled the data access and data alignment. This gave a very good balance of delays, and had the advantage of saving both power and delay by eliminating the need for physically separate data RAMs for each associative cache-bank (Way), since the data RAM's Way-number could be pre-determined in the tag cycle and then simply encoded into extra data RAM address bits. To further reduce delays, integrated pipeline input registers were built into both the tag and data RAMs to eliminate input wire delays from their access times.

A simplified block diagram of the IFU and OFU cache controllers is shown in Figure 3. One controller occupies the IF1 and IF2 pipeline stages in the IFU and the other occupies the OF1 and OF2 pipeline stages in the OFU. The same cache controller design is used in each case, so its main pipeline stages corresponding to IF1/IF2 in one case and OF1/OF2 in the other are generically known as CH1 and CH2.

Before the CH1 phase, the cache controller receives the request (read or write), address, and data from the CPU and loads them into its input pipeline registers.

In the CH1 phase, the tag RAMs are accessed and the tag comparison is performed. The first version of iCORE has 8 KB caches and a 16-byte line size, so 512 tag entries are required. At the end of CH1, the result of the tag comparison (hit/miss), and the CH1 address and data fields from the original CPU request are forwarded to the CH2-phase pipeline input registers.

In CH2, the data RAMs are accessed and the data is returned to the CPU's Instruction Fetch Buffer (IFB) in the case of the instruction

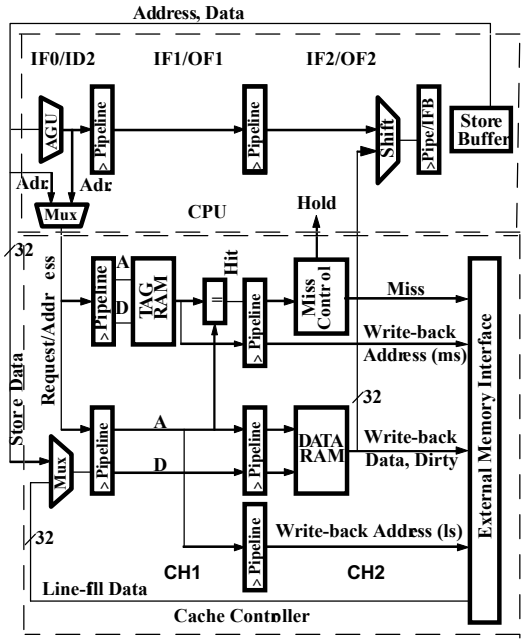

Figure 3. Cache Controller

cache, or the data aligner on the inputs to the EXE stage in the case of the data cache.

As well as returning data on a cache hit, the CH2 stage also initiates a cache-miss sequence if the requested data is not in the cache, asserting the *hold* signal to freeze the CPU pipeline until data fetched from external memory is returned.

2.5 Effects of Microarchitecture on IPC

Most of the microarchitectural enhancements' effect on IPC were first studied through the use of a C-based statistical performance model, which "executes" various benchmark program traces produced by an ST20-C2 Instruction Set Simulator (ISS). They include basic instruction folding, the Local Workspace Cache and its alternative designs, enhanced instruction folding enabled by the use of LWC, and branch prediction schemes and designs. The architecture of some of the features was fine-tuned during RTL implementation, as more accurate performance and area trade-off analyses were made possible. A summary of the effects of these microarchitecture features on IPC is listed in Table1.

Table 1: IPC Enhancements on Dhrystone 2.1

Features	IPC Effect	Overall IPC Improvements
Add basic fold	10%	10%
Add LWC	10%	20%
Enhance instruction folding	7%	27%
Add LWC rotating index	8%	35%
Add branch prediction	5%	40%
Enhance branch prediction transition algorithm	1%	41%
Add Return Stack	3%	44%
Add LWC dynamic disable	4%	48%

3. OPTIMIZING THE IMPLEMENTATION

Implementation of the design was based on a synthesis strategy for the control and datapath logic. Only the RAMs used custom design methods.

3.1 Synthesis Strategy

The synthesis process was divided into two steps. The first step employed a bottom-up approach using Design Compiler™ in which all top-level blocks were synthesized using estimated constraints. Blocks were defined as logic components generally comprising a single pipeline stage. The execute unit of the processor included a large multiplier, for which Module Compiler™ was used as it was found to provide better results than Design Compiler™. As the blocks were merged into larger modules, which were finally merged into the full processor, several incremental compilations of the design were run to fine tune the performance across the different design boundaries.

The only region of the design where synthesis provided unacceptable path delays was in the OF2 and the IF2 stages. In both cases the problems involved the use of complex multiplexer functions. Custom multiplexer gates were designed for these cases and logic was manually inserted with a *dont_touch* attribute to achieve the desired results. Other complex multiplexer problems were resolved by re-writing the HDL to guide the synthesis tool towards the preferred implementation.

The second step of the synthesis strategy employed a top-down approach, with cell placement using Synopsys' Physical Compiler™, which optimized gates and drivers based on actual cell placement. Physical Compiler™ proved to be effective in eliminating the design iterations often encountered in an ASIC flow due to the discrepancy between the gate load assumed by the wire-load model and that produced by the final routed netlist. The delays based on the placement came within 10% of those obtained after final Place and Route. Physical Compiler™ can be run with either an "RTL to placed gates", or a "gates to placed gates" flow. The "gates to placed gates" flow was found to provide better results for this design.

3.2 Standard Cell Library

A higher performance standard cell library (H12) was developed for iCORE to improve circuit speed compared to a supplied generic standard library. Speed improvement of more than 20% was observed in simulation and on silicon when the H12 library was used.

The conventional CMOS static designs used in the generic library were used in the H12 library, but with the following differences:

- Reduced P/N ratio
- Larger logic gates (vs. buffered)
- More efficient layout
- Increased cell height

The P/N ratio for each cell was determined by the best speed obtained when the gate was driving a light load. It was observed that when the gate-load increased, the P/N ratio needed to be increased to get optimal speed. However, the physical-synthesis tool minimizes the fan-out and line loading on the most critical paths, so the H12 library was optimized for the typical output loading of those paths.

Delays were also reduced by creating larger, single level cells for high-drive gates. The original library added an extra level of buffering to its high-drive logic gates to keep their input capacitance low and their layout smaller, but at the expense of increasing their delay. Eliminating that buffer by increasing the cell size also increased the input capacitance, but still the net delay was significantly reduced. These cells were only used in the most critical paths, so their size did not have much impact on the overall layout size.

The H12 cells generally had larger transistors that required the layouts to be taller. However, by using more efficient layout techniques, many of the cells widths were reduced compared to the generic library. Also, the larger height of the cells allowed extra metal tracks to be used by the router that in turn increased utilization of the cell placement.

4. PHYSICAL DESIGN STRATEGY

Physical Compiler was used to generate the placement for the cells. The data cache and instruction cache locations were frozen during the placement process and the floorplanning was data-flow driven. Clock lines were shielded to reduce delay uncertainty by using Wroute in Silicon Ensemble™ by Cadence. To minimize the power-drops inside the core, a dense metal5/Metal6 power grid mesh was designed.

The balanced clock tree was generated with CTGen™. The maximum simulated skew across the core was 120 ps under the worst-case process and environmental conditions. Typically this would be significantly less. The final routed design was formally compared to the provided netlist using Formality™. The Arcadia™ extracted netlist delay was calculated using Primetime™.

5. RESULTS

The iCORE processor was fabricated in STMicroelectronics' 0.18m HCMOS8D process. A GDSII plot of the test-chip is shown in Figure 5. Excluding the memories, the entire chip is seen to have the distinctive random layout of synthesized logic. The large memories on the right and on the top and bottom of the plot are the data and instruction RAMs. The other small memories seen in the plot are the Local Workspace Cache, the Branch History Table and Branch Target Buffer.

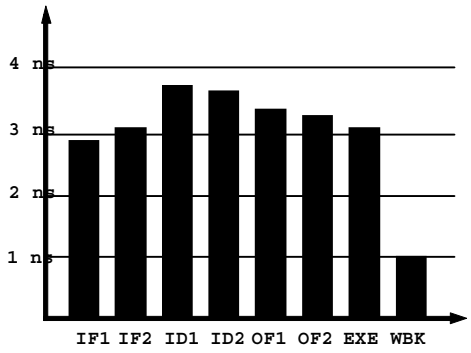

Figure 4. Static Analysis: Worst-case Pipeline Paths

Timing simulation indicated that a good balance of delays between the pipeline stages was achieved, as shown in Figure 4.

Silicon testing showed functional performance of the design from 475 MHz at 1.7 V to 612 MHz at 2.2 V and 25 degrees Celsius, ambient.

Analysis of the Dhrystone 2.1 benchmark showed that iCORE achieved an IPC (Instructions Per Cycle) count of about 0.7, which met the goal of being the same or greater than that of previous ST20-C2 implementations, indicating that the various pipeline optimizations were functioning correctly.

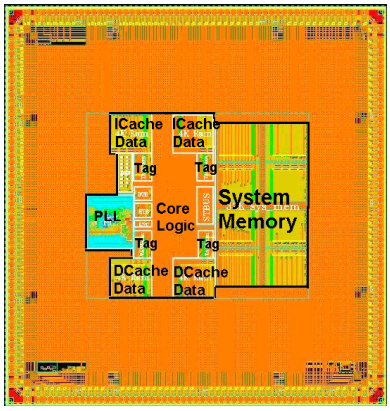

Figure 5. iCORE Test-chip GDSII Plot

6. CONCLUSIONS

The performance gap between custom and synthesized embedded cores can be closed by using deep, well-balanced pipelines, coupled with careful partitioning, and mechanisms that compensate for increased pipeline latencies. This enables the use of simple and well-structured logic functions in each pipeline stage that are amenable to highly optimal synthesis[3]. The performance gains are consolidated by use of placement-driven synthesis, and careful clock-tree design.

This paper explored the use of advanced logic design to implement a high performance CPU core using an ASIC methodology. By avoiding custom circuit design, we have been able to reduce the design time and high costs traditionally associated with it, while still achieving excellent performance.

The final chip has been demonstrated to be functional in silicon at up to 520 MHz under typical conditions.

7. REFERENCES

[1] C. D. Snyder, "Synthesizable core makeover: Is Lexra's seven-stage pipeline core the speed king?," *Cahner's Microprocessor Report*, July 2, 2001.

[2] "ST20-C2 Instruction Set Reference Manual", *STMicroelectronics, 72-TRN-273-02,* November 1997

[3] D. G. Chinnery and K. Keutzer, "Closing the gap between ASIC and custom: An ASIC perspective," *Proceedings of the 38th Design Automation Conference*, Las Vegas, NV, pp. 420-425, June 2001.