# VLSI design project
# Final report


| | |
|---|---|
| **Course**: | VLSI design project (ELEC 6027) |
| **Examiner**: | Iain McNally |
| **Group**: | G6 |
| **Group members:** | |
| | BawladyRaghavendra (rb905) |
| | Galatsopoulos Ploutarchos (pg205) |
| | Kouzis – Loukas Dimitris (dkl105) |
| | Stathousis Marios (ms605) |
| **Date**: | 26 May 2006 |


## Table of contents

# 1. Introduction

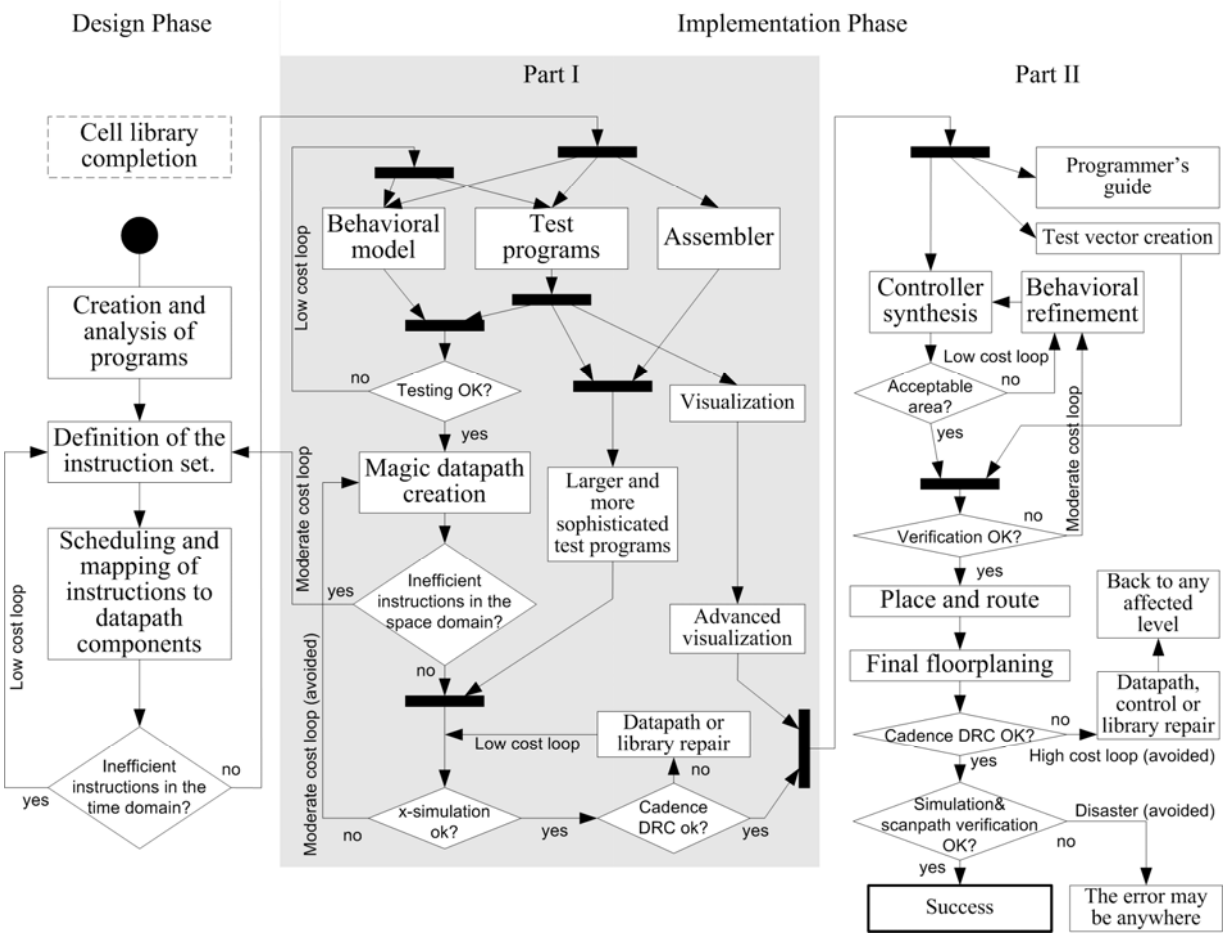
## *1.1 A success story*

This project is a success story. It delivered the thoroughly tested MOUFA processor with a very small footprint, able to fit loosely in the smallest possible pad ring. It supports interrupts and has a powerful instruction set that delivers maximum performance for the given memory interface. But no processor was ever a success only because of its performance. Our processor is supported by a complete software suite, consisting of an AT&T compatible assembler and a user friendly transistor-level simulation environment. A set of almost 2x400 test vectors and a utility for easily creating more can ease post-fabrication structural verification. Last but not least, the design is supported by a high quality 87-page programmers' guide structured in such a way that is both a complete reference guide and a kick start tutorial.

But why was it a success? Two are the main reasons:
1. Our modified design flow that helped us to differentiate over the competition.
2. The correct utilization of the members of the team in order to get the best result.

## *1.2 Our refined design flow*

An overview of our refined design flow can be seen in the following figure.



The steps of this design flow are the subject of the following chapters. The structure of this document is being based on it as well. This design flow isn't a result of random decisions nor, unfortunately, as a result of our long experience in this area. It is the result of careful thinking before each step and with the aim to minimize the risk of failure especially in the last phases of the development where the deadlines limit the number of feasible options and failures become very costly.

It must be also noted that the most important elements in the flowchart above are not the processes (steps)… The most important elements are the ones that most naïve project managers overlook. It is the number of loops (branches), their cost and the probability to take them! These define the risk of the project and lead to success or failure.

The existence of some decision (test) nodes early in the design flow, such as the Cadence DRC check on the datapath, vastly reduced the risk of failure. If we haven't done it at that early stage the next stage for this test would be just before the final deadline (as required from the original design flow). If an error exists in the library (as was the case) and it was not found until that late stage, there might be needed changes in the whole design, both in the synthesized controller and the automatically placed/routed control. Of course these should be followed by extensive re-testing. This would be a real disaster just a few hours before the final deadline!

In most cases our design flow was in accordance with the original project's design flow as dictated by the milestones and the deadlines. In some other cases this wasn't the case. This was unavoidable because different priorities are being set by each flow. The original design flow was oriented towards delivering a good product with relatively low risk of complete failure. Our flow was oriented towards delivering an extremely high quality and thoroughly tested product which of course results in higher level of risk.

## *1.3 Correct utilization of team members*

After a probing period of about two weeks, it was clear who was good on what and who was not. Because our primary goal was to deliver a high quality product, we had to allow a large amount of time on integration and testing. Also UI, compiler, test vector and documentation creation require vastly different skills.

Some of the team members thought that some tasks were required some others didn't agree. After the probing period it became clear that trying to force people do things they don't want to, results in amazingly poor quality results and deadline misses. Another problem was that here we don't have salaries, dismissals or other such ways to force people do things they don't like. On the other hand it's very difficult to motivate people who are expecting different things from a course like this. Someone may want to learn new things, someone may want to get an acceptable mark with minimum effort, someone may be dying to see his name printed on a piece of silicon. The point is that there is no right or wrong. All these approaches are acceptable, correct and MUST be well respected.

The solution that we gave to this problem was the following. In each meeting we announced a set of tasks that had to be done. Anyone who was willing to do some of them was required to do a mini-research and specify exactly what he is going to deliver and at which time. Enough time for integration, testing and managing unforeseen problems had to be allowed if something was a

deliverable for a deadline. Internal deadlines where set this way. After each delivery an integration and testing process took place and the final product was delivered to the next process.

An unfortunate result of this methodology is that you can't easily say who has done what as would be desired for easier marking. Every part of this project might have passed from one to up to four members of this team in a process of continuous refinement and improvement. This wasn't an unfortunate result of someone's laziness. This was ENFORCED internally from our team in order to guarantee the highest possible quality of the product. For example for the development of the programmers' guide, an internal deadline was set a long time before the delivery. Each chapter was delivered to somebody else for reviewing. The reviewer returned it to the original author with corrections and the author was responsible for accepting or rejecting them. Another example is this report. Different members have written their own paragraphs in their area of expertise. (For a list of who has worked in each chapter see Appendix H). These paragraphs where integrated with some "glue phrases" and the final result is of much better than if we created it by making anyone responsible to write a complete chapter on something he has partially worked on.

# 2. Design Phase

## *2.1 Cell library completion*

As the microprocessor design is a large design, it is necessary to buffer the Clock , nReset and Test signals. This will enable to avoid Clock skew and to improve the speed. Hence new leftbuf cell was added to the existing cell library as an alternative to the non-buffering leftend. The buffers must be non-inverting and as a result we use an even number of inverting stages.

The number of buffer stages and the size of the transistors used for the buffer cell were determined by considering the potential clock skew between a row containing a single D-type and a row containing a continuous run of D-types for 3000um length.

The length of the scandtype from the cell-library is 320um. To get a continuous run of scandtypes to the length of 3000um, 100 scandtypes were used. The buffer was designed to drive 100 scandtypes, thus taking care of the potential delay (Skew) of the clock signal.

Specification of the problem:

We want to have a buffer that will have the following specifications:

- Equal rise/fall time
- Give small skew compared to the setup time (?) maybe 1/10 of the setup time
- Examine dinstictly nreset, clock and Test

**For the Clock/Test:**

87 dtypes → 87 x 4 transistors ~ 87 x 2 = 174 inverters

ln(174) = 5.16 stages

Non-Inverting → 4 stages

Stage ratio (initial) $\sqrt[4]{184} = 3.68$

3 stages to the output buffer sized 129.8 / 2 = 64.9 times minimum inverter:
$\sqrt[3]{64.9} = 4.02$

| Isometric | Last stage boosted |
|---|---|
| 47.9 | 64.9 |
| 13.2 | 16.1 |
| 3.63 | 4.0 |
| 1 | 1 |

**For the nReset:**

87 dtypes → 87 x 3 transistors = 261 inverters

ln(261) = 5.6 stages

Non-Inverting → 4 stages

Stage ratio (initial) $\sqrt[4]{261} = 4.02$

3 stages to the output buffer sized 129.8 / 2 = 64.9 times minimum inverter:
$\sqrt[3]{89.6} = 4.47$

| Isometric | Last stage boosted |
|---|---|
| 64.9 | 89.6 |
| 16.2 | 20.0 |
| 4.0 | 4.5 |
| 1.0 | 1.0 |

**Final measurements on 83 Dtypes in a 3000um row**

| | fall_delay | rise_delay | prop_rise | prop_fall | dif_delay |
|---|---|---|---|---|---|
| Test | 0.42 | 0.40 | 0.94 | 0.98 | -0.02 |
| nReset | 0.36 | 0.35 | 0.97 | 1.01 | -0.02 |
| Clock | **0.44** | **0.40** | **0.95** | **1.00** | **-0.04** |
| | | Lightly loaded | | | |
| Test | 0.08 | 0.11 | 0.75 | 0.76 | 0.03 |
| nReset | 0.07 | 0.10 | 0.80 | 0.81 | 0.03 |
| Clock | **0.08** | **0.11** | **0.77** | **0.77** | **0.03** |
| | | Skew | | | |

| | | |
|---|---|---|
| Test | 0.19 | 0.23 |
| nReset | 0.17 | 0.20 |
| Clock | 0.19 | 0.23 |

We found that the analytical solutions where actually an overkill for our design. We could safely reduce the size of our buffers by large factors.

The leftbuf passed the test described on the web site. For a detailed discussion on testing the leftbuf, see Appendix O. You can see the final layout on Appendix D.

Possible improvements: We could save up to 20% of the total width by lowering the whole n-well instead of our laying-out technique.

## 2.2 Creation and analysis of programs

The first step in our Design face was to create the test programs in an abstract assembly language. We were able to use any instructions we wanted from every machine we wanted. You can see in Appendix A example code for random number generation created in that phase as retrieved from our archives. These programs were actually running in a Machine Of Unknown, Future Architecture which later was refined and formed our MOUFA processor. Of course throughout the project there was always the danger of becoming a Machine Of, Unknown Future, Architecture.

This was a very significant step. Yet before starting to model the processor we had a set of requirements on what we would like our machine to be able to do. This way we had a guideline of a minimum set of instructions that were needed to at least be able to be able to do something useful. You can see this minimum instruction set from that phase in Appendix A.

## 2.3 Definition of the instruction set

After having a minimum instruction set, we enrich it and completed it in order to form our initial instruction set our final instruction set created by this phase can be found in Appendix B.

## 2.4 Scheduling and mapping of the instructions to datapath components

This was a very important task. We had to map the instruction set to the minimum number of datapath components given the memory interface constrain. This included the creation of a 20-page document with a very strict specification on what each instruction will do in each step. The first page from this specification can be found in Appendix B. By using this specification it was easy latter to derive both the datapath components and the behavioral model for the control logic. By finishing this stage we where able to tell exactly how many cycles each instruction would take to complete and what datapath components would it use. As always datapath size – number of clock cycles trade-offs existed and they where resolved by considering performance first.

An initial instruction coding was also able to be delivered at this stage. Of course this would (and it did) lead to un-optimized controller and had to be improved many times until the final version.

A decision that was made from this stage was not to put the Interrupt Enable status to the flag because it would require to increase the bits for jump instructions from 2 to 3 (5 flags)

## 2.5 Inefficient instruction in the time domain?

At this stage we knew exactly how many cycles each instruction would take to execute. Expensive and not so useful instructions like push $immediate and interrupt_enable, interrupt_disable were discarded early. Instead of that A STIE/LDIE instruction was issued to store/retrieve the interrupt enable flag from the Carry flag.

## 2.6 RISC or CISC? MOUFA: The best of both worlds

**RISC features:**

- Instruction set is maximally orthogonal. Extremely compact memory size
- Advanced Short immediate architecture to maximize compactness of code.
- Huffman scheme
- Doesn't use register memory architecture

**CISC features:**

- Variable instruction lenght
- Complex instructions: push & pop
- All register are used as address registers!
- PC absolute and PC-relative instructions.
- Extremely compact code size due to complex instructions
- Most often used instructions support RISC-like immediate modes
- to make even more compact code with single instruction.

From the first stage, the design of Moufa's architecture was based on the idea of combining RISC and CISC approaches. The main idea was not to use a complete RISC because we would be obliged to have serious limitations to the size of our instruction set and the number of registers used, which wouldn't allow much flexibility to the programmer and would lead to poor performance of our microprocessor. On the other hand a fully CISC design approach would result in a complex instruction coding and this would have resulted in an inefficient design of the microprocessor's control logic. That is why our choice was to choose a relatively big instruction set but with careful and clever instruction coding.

The addressing modes supported by our microprocessor, was a significant decision to be made. The more addressing modes are implemented the more flexible a microprocessor is. The problem again is the complexity imposed by the big variety of addressing modes. An innovative decision that was made which has its roots at the RISC microprocessors architecture and it is the use of short immediates, but first things first. The addressing modes supported by out microprocessor are register mode, immediate mode, register indirect mode, based with displacement and implied mode. These are all standard addressing modes used by every microprocessor. What is worth noting in that the immediate addressing mode can be using either a 16-bit immediate or a shorter immediate whose length depends on which is the actual instruction that uses the short immediate. What we gain using the short immediates when possible, is that we have an instruction which is one 16-bit word long in contrast to 16-bit immediate instructions which require two 16-bit words. In this way we save the time that would be required for an extra memory read when the program is executed

In our instruction set there is an instruction that is not one of the standard instructions used in every microprocessor. This is the DECIJNZ instruction. This instruction is used to implement loops very efficiently. Its execution requires the minimum number of cycles which is four and in this time it decreases the value of the I register by one and performs a conditional branch based on the result of the decrement. If the I register after being decreased has no zero value then the jump is executed. If the I register has a zero value then the jump is not executed and after that the next instruction is executed.

MOUFA supports 5 addressing modes,2 kinds of immediates and 3 kind of offsets and 5 instruction types. Let us analyze these numbers.

MOUFA supports 5 different addressing modes.

1. Register Addressing
2. Based with Displacement Addressing
3. Register Indirect Addressing
4. Immediate Addressing
5. Implied Addressing

In the Register Addressing mode the operation is performed between two registers. The one serves as a simple operand (source register). The second one serves both as an operand and as the register where the result is stored (target register).

In the Based with Displacement Addressing mode the operation is again performed between two registers, but in three different ways. In the first case, the instruction word contains except from the source and target register, a 6 bit offset. This offset will be sign extended and added to the source register. This source register now contains the address of a memory place that will be the second operand for the specified operation, the other being the target register. If this offset is zero then this instruction goes under the Register Indirect addressing mode.

In the second case, the instruction word contsains the opcode and a 12-bit offset. This offset will be sifgn extended and added to the current value of the Program counter in order to make a jump to another memory location.

The third case is very much alike the second,but this time the offset is 9 bit long. The 9 bit offset is supported for one specific instruction. The JUMP IF instruction allows the conditional short branch depending on the values of any flag. It is a common fact that most of the times the branch instruction doesn't need to branch that far.
This is the case also with the 12 bit offset.

In the Immediate Addressing mode we have 2 kinds of immediates. In the case of a long immediate (16-bit) the instruction consists of two words. The first word contains the opcode for the instruction and both the target and source register. The second word contsins the long immediate.
In the case of a short immediate ,the instruction consists of a single word which contains the opcode,the source and the target register and the short immediate.
The support of short immediates gives additional speed to the execution of programs because there is the capability of saving 4 clock cycles in the case that the operation needs a number between -256 and +255.

Under the Implied Addressing mode we have instructions that set or clear flags of the system, thus not using neither any of the registers nor the memory.

MOUFA has 5 types of instructions

1. Logical and Arithmetic
2. Bit Manipulation
3. Data transfer
4. Jump – Call – Return
5. Basic CPU control

The first category includes instructions that perform additions or subtractions and one of the NOT, XOR, AND, OR functions to the operands specified in the istrction word.

The second category includes instructions that set or clear system flags. The TEST instruction is also under this category as it tests if a register is equal to zero and sets the flags accordingly.

The third category includes instructions that either transfer data from the memory to the registers or from the registers to the memory. In addition, it includes instructions that move data from one register to another.

The fourth category includes all the jump,call and return instructions that allow the program to transfer the control to another memory location.

The fifth category includes instruction that set or reset the interrupt enable or the carry flag.

The big advantage of MOUFA microprocessor is that the architecture in combination with the very carefully selected instruction set gives the user the chance to write more efficient and compact code with the minimum memory usage, which increases significantly the speed.

Instructions like SET CARRY and CLEAR CARRY can set or clear the carry before the execution of the next instruction without having to perform any operation to any of the register that would result in a zero carry.

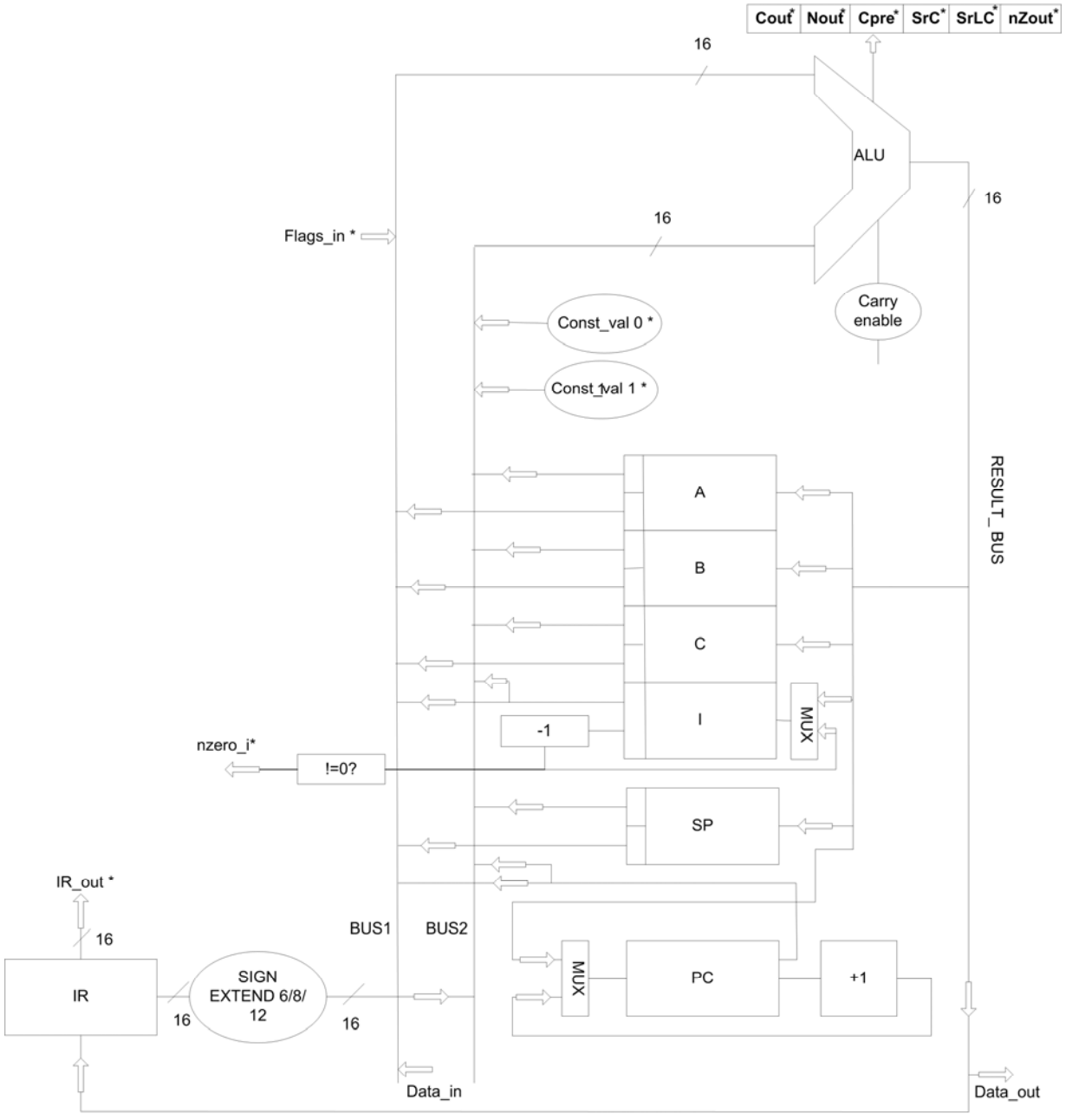
# 3. Implementation Phase: Part I

## *3.1 Behavioral model*

The behavioral model consisted of 2 parts. The model for the datapath which includes the ALU and the model for the controller. We will see them in the following chapters.

### 3.1.1 Datapath

A diagram of MOUFA datapath can be seen below.




* : from/to control unit

MOUFA microprocessor has 7 registers in total. This number allows the best combination of effective performance and relatively small size. There are 3 general-purpose registers A B and C. The I register operates as an inverter which checks if the content is zero. I is used a counter inside loops. For this reason we chose to accompany this register with a designated decrementor. This way we could have much faster functionality.
SP is MOUFA's stack pointer. The instruction set was modified so that PUSH and POP which are used before or after calling a subroutine, operate solely on this register. Of course MOUFA's instruction set allows to push and pop every register.

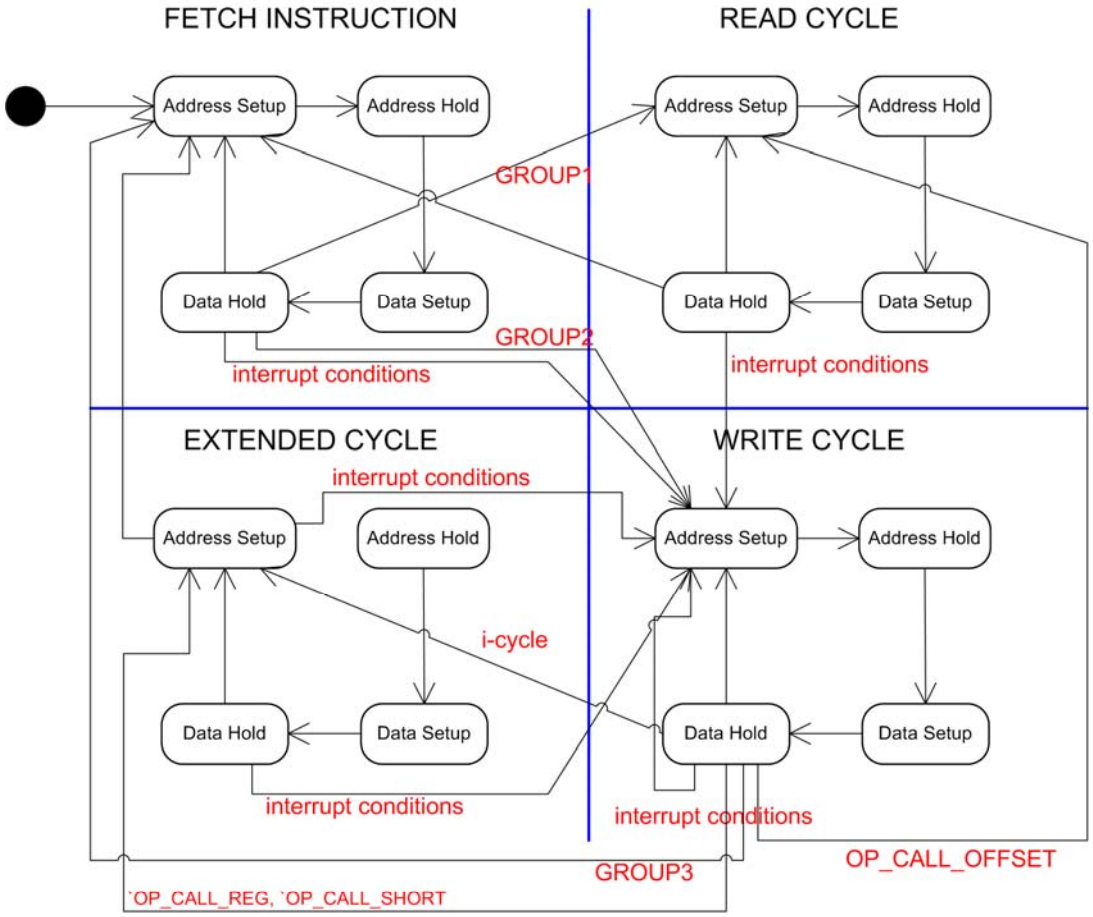
PC operates as the program counter. Again like in the case of I register, PC has a designated incrementor. Because in most cases in a program, the program counter is incremented by one, the choice of the incrementor will allow much faster speed.
Finally, IR is the register that stores the instruction word that comes from the memory. MOUFA supports 16-bit words. It also supports register – immediate instruction format. These immediate can be of 16 bits or 9 bits. In addition what makes MOUFA unique is that it operates with 3 kinds of offsets, a 12-bit, a 9-bit and a 6-bit. For all of the above reasons, IR has to be supported from a sign extension unit. The additional complexity for the variety of different operation it has to make is passed to the very inspired control unit.

The datapath components communicate with each other through 2 16-bit buses each one connected to the two inputs of the ALU. The output of the ALU is the RESULT_BUS, that connects to all the registers. The communication with the memory is done through the RESULT_BUS when we write something to the memory and through BUS1 when we read something from the memory. In addition to the 7 register there are the constant 0 and 1 that are fed from the control unit to the ALU when we have to move a value from one register to another or load a value from the memory to one of the registers. Another unique feature of MOUFA microprocessor is that the supported flags are not stored in a register but rather fed from the ALU to the control unit which does all the checking and all the calculations for their updated values.

ALU forms an important part of the data path in our MOUFA architecture, enabling us to perform both arithmetic and logical operations. The BUS1 and BUS2 of the data path are the inputs of the ALU. As each bus is of 16 bit, the ALU performs operations on 16 bit operands and throws the 16-bit result on the 16-bit Result Bus. The carry input to the ALU comes from the control logic based on the operation to be performed. In our MOUFA architecture, we have constant values either ‘0’ or ‘1’ fed from the control logic as one of the inputs of the ALU via BUS2. These constant values enable vast majority of the ALU functions like additions or subtractions of the content of the registers or memory by 1. The constant value ‘0’ is needed in the case of passing the value of a register or a value from memory via ALU to another register or another memory address. It is also used in comparing a register value with zero in order to produce the appropriate flags. In addition, those units are used in allowing a value from a register to drive the data out bus in order e.g. to give the address for a read: RD <- {A+offset}, or the PC in Instruction fetch phase: IR<-{PC}. ALU in our MOUFA architecture generates/updates six flags like zero flag, nZout, Arithmetic carry flag, Cout, Arithmetic Last carry flag, Cpre, Negative flag, Nout, Shift right carry flag, SrC, Shift right last carry flag, SrLC. For details on the operation of the ALU refer to see Appendix K.

### 3.1.2 Control logic

The control logic is a just a complex state machine. We can see in the following picture how complex it is even in a form where many details have been omitted (e.g. flags, interrupts)




GROUP1: `OP_MOV_REG_IMM, `OP_AND_REG_IMM, `OP_OR_REG_IMM,
`OP_XOR_REG_IMM, `OP_ADD_REG_IMM, `OP_ADDC_REG_IMM,
`OP_SUB_REG_IMM, `OP_SUBB_REG_IMM, `OP_CMP_REG_IMM,
`OP_MOV_REG_ADDR, `OP_POP_REG, `OP_POP_STATUS, OP_JIF (if jump required)

GROUP2: `OP_PUSH_REG, `OP_PUSH_STATUS, `OP_CALL_OFFSET,
`OP_CALL_REG, `OP_CALL_SHORT, OP_MOV_ADDR_REG

GROUP3: `OP_MOV_ADDR_REG, `OP_PUSH_REG, `OP_PUSH_STATUS

From the beginning and by using the experience of “Digital IC design course” the code was synthesizable. After some developing we turned from the “if based” coding to the “case based” coding which is a better practice for state machine coding because the synthesizer can choose to turn them in parallel or full parallel modes in order to reduce area. It can be seen in section 4.4 that these and even more where not enough.

For the initial behavioral model of the controller we used extensively functions. These where easy to be derived from the early specification document. By using these functions the coding of the instructions was really a Software Matter of coding in a micro-code like way.

A critical point for the control logic was interrupts. These where issued form the beginning as they should and form the first delivery of the behavioral model, we where able to serve interrupts from the timer module. Detecting when we must check for an interrupt is easy. We just check when next_state is Fetch_IR, next_sub_state is Data_SETUP and the extended cycle is 0. If

interrupt conditions hold true, instead of executing the next instruction, we initialize an i-cycle (which is considered through this text as a separate instruction with higher priority than the one on the IR). This i-cycle is responsible for pushing the program counter to the stack, disabling the interrupt and redirecting execution's flow to the address 0x0002 where the Interrupt Service Routine (ISR) must exist. An important detail is that after the STIE instruction we have to check for interrupts according to the NEW interrupt enable's status.

## *3.2 Test programs*

The algorithm used for the multiplication program is very fast especially when one of the numbers is small. One of the most common algorithms used for multiplication is the one that takes the multiplier, shifts it to the right, takes the bit that overflows, if it is one the multiplicant is put in an accumulator and the accumulator is then shifted to the left (multiplied by two). If the bit that overflows is zero the accumulator is just shifted to the left without any value being added to it. The algorithm can be seen below

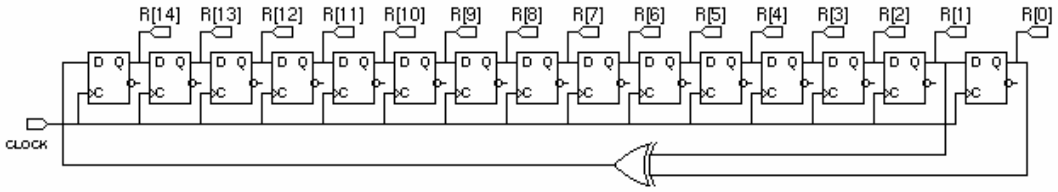
```
A = Multiplicand; B = Multiplier; P = Product register = 0
P = 0;
 for( i = 16; i > 0; i--)
  {
     if (B[0] != 0)
     {
       P = P + A;
     }
     B >> 1;
     A << 1;
  }
```

This is the algorithm that we use except for a small detail that makes it much faster. When we have two numbers to multiply, first of all we compare them and use the smallest one for right shifting and the largest one to be added in the accumulator. After each right shift we check if the number that occurred is zero or not. If the remaining number is zero there is no point to continue the algorithm as we already have the final result. This small detail can increase the speed of multiplication amazingly if one of the operands is small.

The factorial program program is presented to demonstrate the use of stack for recursive routines. It could be implemented in a much easier way by using a single loop where a counter has the value of the number whose factorial we want to calculate and a variable is initialized to this number too. Then the only thing that we have to do is to decrease the counter and multiply it with the variable. We repeat this process until the counter gets the value one. In our recursive implementation we consider that the factorial of a number N is the product of N with the factorial of N-1. What we actually calculate isn't N! but N! mod 65536 because the result is 16 bits long. In order to calculate the products we call the multiply subroutine that we have previously implemented.

The algorithm we are using in order to implement the random number generator uses a 15-bit LFSR and produces 15-bit random numbers. The LFSR that is going to be used takes the zero

and first bits of the shift register and puts them as inputs to a xor gate. The output of the xor gate is fed back to the 15th bit of the shift register as we can see below



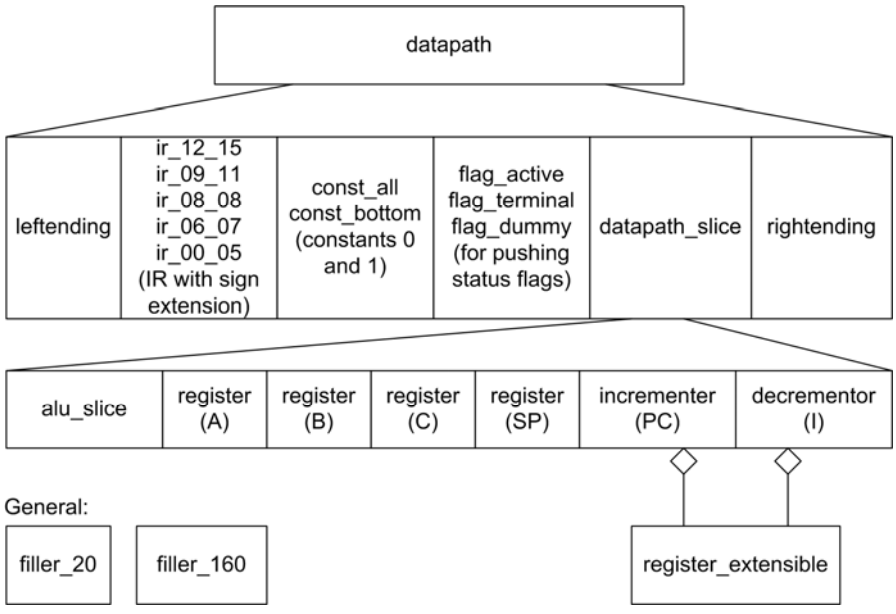

The algorithm used produces a 15-bit result every 15 clock cycles using the LFSR that we have just presented and has a maximal length sequence which means that it produces 32767 values before repetition. The subroutine is an implementation of the LFSR's function for 15 clock cycles. This means that we just have to implement the behavior of the LFSR for one clock and repeat it for 15 times.

## 3.3 Testing OK?

The testing of behavioral model took place with the initial test programs that where either hand-coded to numbers or created by the appropriate compiler-like program. This was a very tight test-development process as the development of the processor was taking place on the same time with the tests. Multiple iterations of this process took place till we had a completely working system. It must be noted that from this initial test phase, interrupts were fully supported.

## 3.4 Magic datapath creation

In order to reduce the effort of routing a highly hierarchical design of the cells – building blocks of the datapath got applied. In the following figure you can see the hierarchy of the cells. The schematics can be found in Appendix N and the final layouts can be find in Appendix D.




The ALU was designed with the lowest cost cells that we had in terms of area, for our library.

| | Solution proposed in[1] | | Our solution | | |
|---|---|---|---|---|---|
| mux2 | 3 | 480 | 1 | 160 | 160 |
| fa | 1 | 180 | 1 | 180 | 180 |
| and2 | 4 | 320 | 1 | 80 | 80 |
| or2 | 1 | 100 | 1 | 100 | 100 |
| xor2 | 2 | 240 | 1 | 120 | 120 |
| trisbuf | 1 | 80 | 5 | 400 | 80 |
| Total area: | | 1400 | | 1040 | |

An important issue in the design of the datapath was minimizing the critical path that was as always the carry path. The lines that involve the carry have been routed carefully in order to ensure minimum distance.

The design of the cells that where responsible for the sign extension was quite tricky because of the many different modes that had to be supported. Advanced bit-slice design techniques where applied.

## *3.5 Inefficient instructions in the space domain?*

Now we had sufficient information to see how much area in terms of datapath each instruction consumes. At this level we re-evaluated our instruction set. Everything was in accordance with our initial planning, so no instructions were actually removed. Some of them where actually modified.

What we did was to remove a top row from the datapath that used to create the flags out of datapath's signals. This row was very efficiently laid out but most of its expressions would just disappear if they got integrated within the controller. This changed somehow the semantics of our architecture. For example instead of storing the Zero Flag we are now actually storing the NotZero flag and the instruction jz/jnz instruction gets actually encoded with reversed way in respect to the others.

It also became clear that storing flags inside the datapath would be an extreme waste of area. The 4 flags were designed to be integrated to the controller. Driving of the sign extend bit from the IR has also been moved to the controller. The push/pop status semantics got finalized as we know new that the flags where going to be pushed/popped from the 4 MSB's of the memory because these are closer to the top of the datapath where the interface with the controller was expected to be.

In a similar manner the semantics of the sign extension modes got finalized. There were some discussions about putting the IR in the control logic in order to be more tightly coupled with the decode logic. We also gave up this idea because of the very efficient layout of the IR and sign-extension module. Shifting/rotating operations also got finalized at this stage. The idea of increasing/decreasing the stack pointer via datapath was also thrown away. The idea of sharing the half adder between the PC and the I register was also thrown away as it required multiplexers with more area than the extra full adder.

[1] http://www.ecs.soton.ac.uk/~bim/notes/ups/pdf/ups02.pdf

After this evaluation process we had a finalized very compact layout of our datapath.

## 3.6 Visualization

Our original visualization was actually based on a modification of the original xl_graphics.v of previous years. This visualization worked with Verilog-xl and required modifications in the monitor.v file. The most important restriction that Verilog-xl imposed was the usage of an older Verilog standard. This way we couldn't write e.g. module control (output reg a); but we had to write it in three different lines:

```
module control(a);
output a;
reg a;
```

Due to late integration on the project converting the original syntax to this became a very stressful task. In order to reproduce this simulations see at Appendix F.

## 3.7 Assembler

I (dkl105) wrote the assembler. This assembler was never meant to be simple. From the very early stage of development, the goal was to be able to compile the (AT&T syntax) .s assembly files that gcc (See Apendix J) creates when we use the –S flag.

In order to achieve this I used two very powerful compiler compiling tools; Lex and Yacc. Even by using them, the source code is about 850 lines long. If these tools haven't been used, the assembler would never have been completed at the quality I wanted.

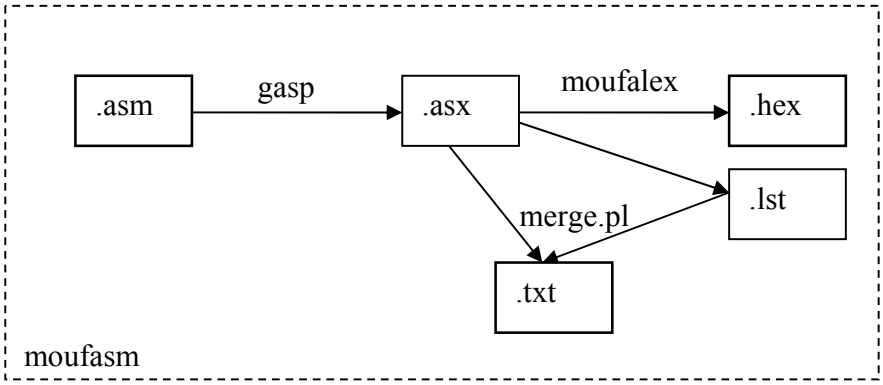
The assembler does two passes. In the first pass it tries to identify values for all the labels and in the second pass it does the actual parsing and code generation. At the beginning it had a feature of auto-detecting the appropriate instruction by its operands in the case of instructions with short immediate and full immediate option (Jumps, adds(-subs) and movs). The problem with jumps is that if at the first pass we don't know if the jump instruction is one or two words long, we can't figure out the offset of the labels that has to be used in order to decide if that instruction is one or two words long. Later we used labels for variables as well and the same problem occurred for moves and adds (this could be solved by creating data-program sections but it would be unnecessary complicated). Of course these problems are classical compiler building problems and they could be solved relatively easily with some of the solutions proposed in the bibliography. Addressing these problems was beyond the scope of this project. gcc can do it for us anyway, by just including the length of each instruction in the appropriate file.

One the interesting problem was passing the number of lastly parsed line from lex to yacc. This was necessary in order to provide meaningful error messages and in order to create a listing file that would be latter used to create the .txt file with the description of code. By looking at gcc's source code I was inspired to use the technique that was finally used.

One good fact of being GNU-compatible is that we can re-use components. We set up GASP[2], the GNU assembler pre-processor, on the top of our assembler and we get (almost) for free macros, conditionals and other useful facilities.

[2] http://www.ia.pw.edu.pl/~wujek/dokumentacja/gnu/gas/gasp_toc.html

The whole assembler data flow is as shown bellow:




A component that we haven't commented yet is merge.pl. This is a perl script that takes the list (.lst) file created from the assembler and the (.asx) file which was input on the assembler and merges them to the .txt file with the listing of the program. This functionality could be integrated inside our assembler but it would be more time consuming to code it in C instead of perl.

I would also like to note that in the .txt file apart from the things required by the specification, the program's counter value is also listed. This is useful for understanding the program but is also essential in order to make the advanced visualization plug-in described bellow to work.

The source code can be found in the verilog/assembler/src directory. There also exists the "compile" script that easies its compilation. Note that in order to compile successfully it needs lex & yacc to be installed on the system. For information on using the assembler refer to programmer's guide.

## *3.8 Larger and more sophisticated test programs*

By using the assembler we were able to create larger and more sophisticated programs. They were also used to test the assembler itself. full_test.asm which tests all the instructions of the system (including the interrupts via the system timer) and shuffling.asm that creates high-quality random numbers were created. The final test programs are the following: factorial.asm, full_test.asm, multiply.asm, random.asm, shuffling.asm.

One bad thing about LFSR is that it has a small period of $n^2$-1 which in our case is 32767. This can be easily and without much performance overhead get extended by using suffling[3]. This technique has been used on top of the LFSR in shuffling.asm and gives and astonishing period of $O(32767^{128})$. It also destroys "correlations concerned with hyperplanes"[3] which means that we produce random enough numbers for most applications. It must be noted that the shuffling program creates at the beginning SAM_NUM (in our case 128) + 1 random numbers for state initialization. This translates to about 65000 cycles. This increases initialization overhead but after that it's almost as fast as the simple version (521 cycles/number instead of the original 477 cycles/number = 44 cycles/number overhead).

[3] "Monte Carlo Methods in Statistical Physics", M.E.J. Newman, G.T. Barkema, Oxford University Press Inc. 1999, New York, pp. 391

## 3.9 Cross simulation

The datapath.v and datapath.vnet were extracted form the layout (a process that could cost up to 5 mintues). Then a lot of signals had to be bind from the .vnet file to actual datapath's variables to allow probing of its internal state. This was achieved by modifying datapath.v. Magic's :getnode command greatly accelerated thisprocess.

The initial cross-simulation attempt included having both datapath modules (behavioral and extracted) in core_cpu.v and driving them with the same input signals and observing their output signals. This way I had a fail-proof testbench that would demonstrate inconsistencies between the two the models.

Because of the careful (time consuming) magic laying out, the cross simulation worked immediately. There where only slight inconsistencies that were resolved within an hour. At the end the differences where only glitches.

Then we split the project to two folders, the behavioral and the new mixed folder. We attempted simulating the mixed model on its own. As a great surprise, the the system didn't work at all. The problem was that the values of the registers instead of changing where oscillating all the time. After some careful debugging the cause of this was found. The behavioral model had 0 propagation delay on its registers which caused simultaneous change on the data and the clock lines on each clock's rising edge. This was a violation of the hold time of the registers of the datapath that caused these oscillations. The problem was solved by adding propagation delay on controller's flip flops. But this created another problem. Memory timing violations existed on the beginning of the simulation by having the registers in an undefined 'x' state for some time. These violations exist on the actual system and we will see them again on the section 4.11. The solution was to set 0 delay on the response of registers to the nReset signal. Of course these registers are not realistic.

For reproducing these cross simulations see Appendix F.

## 3.10 Cadence DRC

At this level we had our first piece of layout completed. The cadence DRC process (do_cmos05_cellin, DRC with drc_shape? not_pads?) that was used in the previous semester for the cell library was applied to the datapath. This was the right time to do it, even if it looked early because afterwards the complexity would greatly increase and much of the work may had to be repeated. It was proved that errors existed and were repaired as described in the next section.

At this early stage we also created the minimum pad ring (including interrupt pin) in order to see how much space was available for the datapath. The minimum pad ring allowed 1436 x 1436 nm on its inside. Our datapath was 783 x 852 nm large so we had filled already more than 50% of our space.

## 3.11 Datapath or library repair

Our library had actually some problems. More specifically some taps were conflicting with our active regions. Actually our library was a little bit over-designed in respect to the number of taps (for the given process) so we resolved this issue by simply removing them. Another problem was that some (small) cells like tielow, tiehigh and rowcrosser didn't have an N-well. This was also easily resolved. Although simple, these mistakes could lead to panic if they where issued a few

hours before the deadline. Even worse, if we haven't thoroughly tested the library in the previous semester, there could be more serious and hard to solve problems. It is important to use as strict as possible design check as soon as a piece of layout becomes available. There is no good testing its behavioral functionality if it can't be actually used.

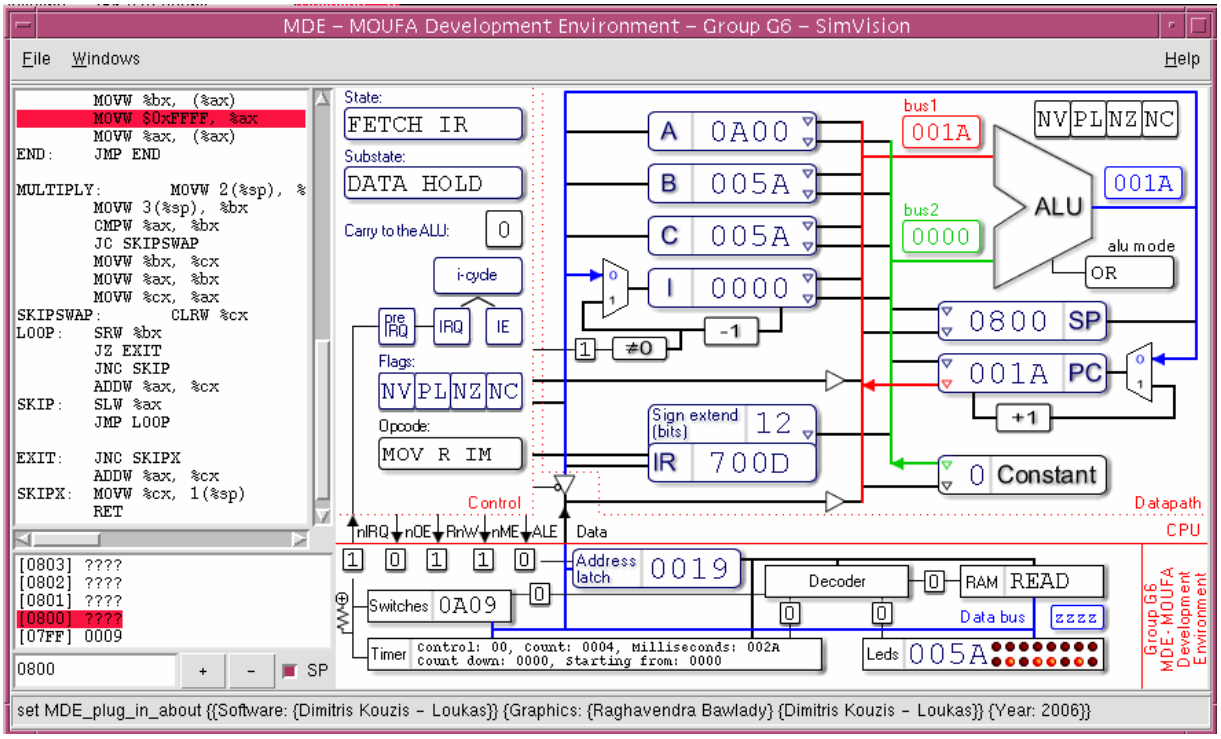
## *3.12 Advanced visualization [dkl105]*

We were the first "customers" of our product and we suffered from the lack of an appropriate User Interface able to give us quickly all the information we need to debug the processor and our programs. Based on this remark and with the willingness to make our (customer's) life easier, we derived a set of requirements for building a better user interface. The most important of them were:

1. To be at least as good as the old one which means:
    a. To include all the information we are able to see with Verilog XL.
    b. To be able to navigate quickly
2. To be similar to other simulators which means:
    a. To be able to show the program and the current execution position.
    b. To be able to show the contents of the memory and the stack
    c. To be able to show the state of system's peripherals
3. To be attractive and with high educational value

Obviously these goals could not be met with register window's functionality of Simvision. We needed something more. We used SimVision's convenience classes to create our MOUFA Development Environment (MDE).

The language for its development Simvision plug-ins is TCL. Of course I (dkl105) have only used TCL in the previous semester for first time and there were a lot of things to learn. I have some more details on the development process in Appendix I because a demonstrator expressed his interest in learning TCL/TK and maybe some more are willing to do the same. The result of the development effort is MDE. We can see its main window in the following figure.




All the design goals have been met with this User Interface. On the top left side the source code of the currently running program appears. The currently executing line of code (retrieved by the value of program counter) gets highlighted with a red frame. Under that frame exists the memory view frame. By it entering a value on the field or by pressing the +/- buttons we can view the contents of any memory location. With a single click on the SP check-box, the memory view follows the values of the stack pointer showing the contents of the stack. The memory view shows also a few addresses before and one address after current stack position.

The main frame of the plug-in Shows the state of the processor and the complete system. The view of the system is extremely detailed and accurate as extracted by the verilog models

The high educational value originates from the appropriate design of the main frame. All the sequential elements of the system are colored with dark blue color. The combinational and complex (system level) elements are indicated by black color. The three CPU busses result_bus, bus1 and bus2 are colored with blue, red and green respectively. The active three state buffers get highlighted with the color of the bus that they drive. True Led indications present the current status of the Leds and all the information available for the peripherals is presented on the screen. The flags are named with the familiar x86 notion and appear in the same order as in any x86 debugger. The user doesn't have to learn the user interface. Everything is organized in a way that allows intuitive use of the it.

A lot of effort has been put into laying out all these elements of this complex system with a way that makes it look simple. At the same time a constrain was that the whole layout should fit in the screen and allow free space for waveform windows even if we use a 1024x768 resolution screen.
A problem that we faced during development was that the program became a little bit slow mainly because of the network overhead that any UNIX UI system requires. In order to anticipate this, we split the UI update routine to two parts. The one responds immediately to SimVision's

cursor's movement and updates the current position on the code view. If you don't move the cursor for half a second, the complete UI updates. During its update a wait cursor prevents any further update of SimVision's cursor.

MDE was ready at about the middle of the project (first days of Easter). We believed that it was one step beyond any competition and in order to prevent unnecessary "questions" from others we weren't using it publicly until the last week. For more information on how to perform simulations refer to Appendix F.

# 4. Implementation Phase: Part II

The Part II of the implementation phase is mainly distinguished from the Part I from the chronological order. This part took place during and after Easter. We received from Part I a complete design specification, a working behavioral model, a great simulation environment an assembler and the layout of the datapath. Now we have to complete the design and document it.

## *4.1 Programmer's guide*

MOUFA's programmer guide is a very helpful accompanying document of our microprocessor. It addresses all main aspects that a programmer needs to know to get accustomed to MOUFA's full capabilities.

First of all we present the architecture of the microprocessor in all details so that the programmer will have a deep understanding of the operations supported by the ALU, the route of data in the datapath and the supported addressing modes.

The main purpose of each programmer's guide is to present the instruction set that it supports. In our instruction set chapter all the instructions are presented in detail and in a user friendly format so that the programmer will be able to see all instructions supported, the addressing modes supported by each instruction, the syntax of each instruction as it is supported by our assembler, the exact operations that it performs when executed and the flags affected by each instruction. Furthermore the binary representation of each instruction is provided along with explanation of the information coded in each group of bits. In this way the programmer is able to actually write hex code for MOUFA processor. The number of cycles needed for each instruction's execution is also provided so that the programmer will be able to calculate how fast his programs are executed.

Then our programmer's guide goes one step forward and presents some useful and ready to use subroutines. In this way the new programmer using MOUFA will have some working MOUFA's assembly examples which can be used as reference as they perform some very useful functions using the full capabilities provided by our instruction set and addressing modes. These subroutines extend the issues that can be dealt by our processor with no effort made by the programmer as they enable him to perform 32-bit arithmetic operations, N- bit right shifts and rotates, if-then-else statements, for loops, while loops, array manipulation operations which are very useful for string handling. Even the core for the implementation of an FIR filter and the bubble sorting algorithm are provided. Apart from these subroutines three complete programs are presented these are the multiplication program, the factorial program and the Random generator program.

In the final chapter of our programmer's guide we provide simulation results of the three example programs. We describe how to use the assembler to generate hex code from the assembly program and we also show the graphical user interfaces provided for the programmer to carry out the simulation. The assembler designed for our microprocessor uses the AT & T syntax or GAS (GNU assembler) Assembly Syntax for writing the assembly level language program. In this way we don't impose our own rules about the assembly syntax, which would make the new programmer of MOUFA microprocessor feel uncomfortable. The programmer just has to be familiar with the AT & T standard syntax.

## 4.2 Controller synthesis

This started as a manual process and ended up to be a highly optimized automated process. At the beginning we were using the synth_custom manually to synthesize controller's file, play with the parameters, see the schematic and identify things that looked irrational on them.

A simple script (count_instances) was developed later to count the number of library cells that our controller has (See Appendix L). Later another simple script was developed that was executing the synth_custom in batch mode and counting the instances as well (See Appendix L). We also created a backup utility that allows us to backup the current version of control.v and all relevant files in a new folder tagged with the time/date by just typing backup (See Appendix L)

It must be noted that the time of synthesize ranged from 7 minutes down to less than 30 seconds depending on server's load, network's bandwidth (which is a bottleneck when you work from Greece) because of the many messages (that should be considered – couldn't just /dev/null them) and the behavioral abstraction level of the Verilog file (the least sequential, the least the time).

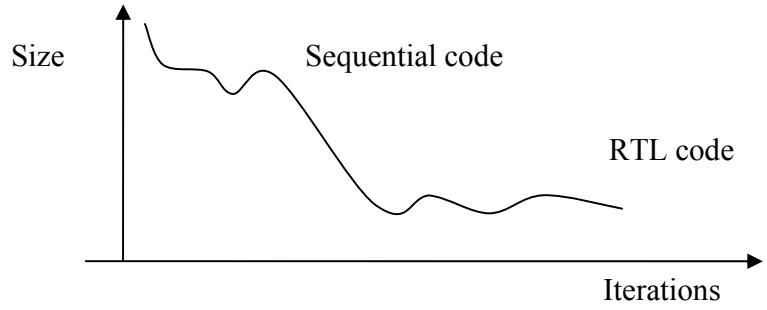
It must be noted that synthesis process is highly non-deterministic (at least with the level of insight we have on Cadence's BuildGates tool via the manual). You may modify a line of Verilog and expect to get a major decrease of the logic count and end up with a hundred more gates. This is extremely true if you have large sequential blocks where there are many things implied and it's up to the synthesizer to make decisions that minimize the area.

## 4.3 Acceptable area?

This question is not as simple as it sounds. What you want as an area is the minimum for the given functionality. You can't just tell a number like 3000 and go for it because it may not be achievable. You can't also apply the usual engineering rule "the project is ready when the deadline comes" because there are many things to do after synthesis. Every moment you spend seeking less area you cut it for other things that may be more beneficial for the project.

In order not to make this question cause an infinite loop, a methodology had to be derived. What we used was a Simulated Annealing[4] – like approach. We start with a solution. Then we apply as many deterministic modifications as we can to minimize the area. When get stuck we try to think something else. Sometimes trying something that seems completely irrelevant may lead to unexpected improvements. With our backup utility we can backtrack to any previous good solution.

[4] http://en.wikipedia.org/wiki/Simulated_annealing




As it can be seen in the figure above, the size of the design after each iteration follows some specific patterns. When you start optimizing un-optimized code, you get a rapid decrease of the size. Then the size stabilizes and more ideas applied don't contribute significantly to better solutions. We are stack in a (local?) minimum. If we remain in this range of solutions more than a certain time we can be sure that we are close enough to the global minimum or just unlucky! In either case we should quit.

## *4.4 Behavioral refinement*

The aim of this step was to reduce the number of cells needed from the datapath. Our initial design had 688 (!!) cells and an error that would result to about 700 cells. Obviously it had to be optimized.

The very important step towards optimizing was the change on the coding of the arithmetic instructions. All the instructions got merged to a single mega-isntruction. This was achieved with the following manner:

The 7 control signals of the ALU got compressed to 4 signals this way:

Compression from 7 to 4 bits :

```
NegEn = ~C3 & ~C2
SubEn = ~C3 & C1
AritEn = ~C3 & C2
XorEn = ~C3 & C0
AndEn = C3 & C0
OrEn  = C3 & C1
SrEn = C3 & C2
```

| ALUmode | C3 | C2 | C1 | C0 | NegEn | SubEn | AritEn | XorEn | AndEn | OrEn | SrEn |
|---|---|---|---|---|---|---|---|---|---|---|---|
| FN_ADD | **0** | **1** | **0** | **0** | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| FN_SUB | **0** | **1** | **1** | **0** | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| FN_XOR | **0** | **0** | **1** | **1** | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| FN_NOT | **0** | **0** | **1** | **0** | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| FN_AND | **1** | **0** | **0** | **1** | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| FN_OR | **1** | **0** | **1** | **0** | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| FN_SRC | **1** | **1** | **0** | **0** | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

Then the arithmetic mega-instruction was formed with 16 bits of the following format:

01 [C1] [C3] [tgt3] [src3] [C2] [C0] [srcZero] [crcImm] [CarryEn] [save target]

The CX signals are as described above. The srcZero forces the second operand of the instruction to zero (useful for mov and test instructions). The crcImm defined if a long immediate followed or it was a register-register instruction. The CarryEn flag defined if the Carry flag should be considered in the instruction or 0 should be assumed. The save target flag indicates if the result

should me saved on the target register or the instruction is just a test. The tgt3 and xrc3 are the ids of the target and the source registers respectively (0:A, 1:B, 2:C, 3:I, 4:SP, 5:PC).

The signals that control the flags were hand-coded with the following simple expressions:

store_zero_sign_flag; AND, OR, XOR, NOT,
sotre_all_flags: ADD, ADDC, SUB, SUBB, SR, SRC, CMP, TEST
NO: MOV

```
assign store_no_flags = C3 & (srcZero);
assign store_all_flags = C2 & ~ store_no_flags;
assign store_zero_sign_flag = ~(store_no_flags | store_all_flags);
```

By applying this modification 114 cells were saved an the count got down to 574. With more optimizations on the coding of STIE, LDIE, STC, ADD/MOV short, PUSH and POP we got down to 529. Not important improvements. The next important change was setting enable_carry's signal by default to 0. This reduced the count to 484, an important improvement for a single change. By playing with the default values we ended up with 458 cells. We also tired some tricks like forcing full parallel and changing the order of cases but there were no improvements. Obviously we where stuck.

We had another option that we wanted to avoid in any cost… To write down the Karnaugh maps and optimize them. Unfortunately we had to do it. These tables can be found in Appendix M. Then we started from a clear control.v and added the signals one by one. This way we where able to know exactly how many gates where spend for each expression. These values still exist in the control.v as comments even though they are not accurate anymore. Some times we had to go down to schematic level (synth_custom) to understand where the problem was in cases that we found cell's increase very high.

After all this process, we ended up with control logic with 346 cells which means more than 50% improvement which was a great success. Behavioral refinement was a "low cost loop". The problem was that it got executed too many times.

## *4.5 An unfortunate event*

At this level of development an unfortunate event happed! We received an e-mail that was telling that we had to deliver the processor and programmer's manual in one week.

Somebody claimed before Easter that we had 3 weeks after we return to complete the processor and nobody cross-checked that (so it's everybody's fault). As result we thought that we should deliver the processor at the 26$^{th}$ of May.

You can imagine our shock when we realized that we had one instead of three weeks. We tried not to panic and we achieved that. We rescheduled the tasks in order to minimize the just astonishingly increased risk and we entered a Mission Impossible – Rambo mode by increasing our working hours per day and halting all other tasks that were running in parallel.

At the end of the week we delivered the processor and the programmer's guide as we should.

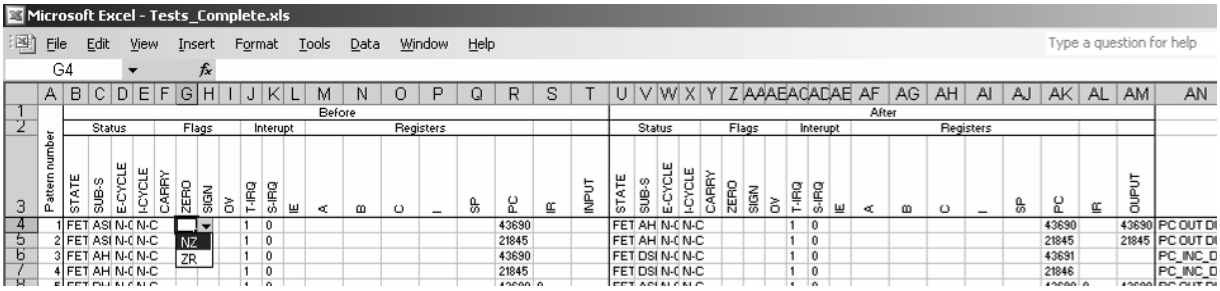
## *4.6 Test vector creation*

The need for test vectors was created from the fact that we needed a way to verify that the microprocessor worked properly for all possible combinations of operation that it supports (including interrupts). The test vectors would be applied to the design before the final placement, route and floorplaning to verify the controller as well as after it.

The philosophy behind the test vector creation is that instead of an exhaustive checking, a smart check was adequate enough to ensure the right-beings of MOUFA microprocessor core. Thanks to our highly vertical instruction set we could reduce highly the number of test needed for verification.

For example, we checked that an arithmetic instruction worked correctly with all the registers and we assumed that all the other arithmetic instructions will do the same (a reasonable claim, since all the arithmetic instructions are actually one). The same principle was applied for conditional jumps that have vertical functionality in respect of the flags.

The tool that was used for the development of the test vectors is an Excel spreadsheet with a large number of Macros and Validation constrains. As we can see in the following picture with our tool you can define what the original state is before the clock cycle and verify that after the clock cycle is what you expect it to be.




The state of the processor before and after the clock includes the following options:

- State (Fetch , Read , Write)
- Sub-state (Asetup , Ahold , Dsetup , Dhold)
- Extra Cycle
- Interrupt cycle
- Flag Values (Zero, Carry , Sign , Overflow)
- T-IRQ (Temporary Intertupt Flag)
- S-IRQ (Safe Interrupt Flag)
- IE (Interrupt enable flag)
- Values of all registers (A, B, C, I, PC, SP ,IR)
- Input of the I/Os (Before) - output to the I/Os (After)

Through these test vectors we had to check whether our registers were updated with the correct values after each instruction and more importantly that all the others remain unchanged. We had to ensure that all flags were updated in a proper manner. It had to be verified that all our conditional jumps controlled by the flags were right from both aspects: a) correct destination b) correct identification of the condition.

There are two different sets of test vectors. For the behavioral model we set all the undefined state variables to Xs (unknowns). If there is an error in the control logic, some of these errors will propagate to the registers we verify.

(Un)fortunately in the real hardware there are no unknowns and as a result for our Scanpath tests we had to create another two sets of test vectors where the undefined state variables are filled with 1 and with 0 respectively and of course are expected not to change. In this case also the output bus is expected to be in 'Z' state when undefined.

The Excel tool can be found in the verilog\core_test\Tests_Complete.zip where also exist the 3 resulting test vector files tests_1.txt, tests_0.txt, tests_1.txt

## *4.7 Verification*

We used these test vectors in order to verify the RTL model of the processor before placement and routing. By extensively testing at this level we avoided the extremely expensive post-routing modification of our control logic.

In order to test the control logic we used a testbench Verilog module (core_test/core_test.v) and we modified the control.v, cpu_core.v, datapath.v slightly to provide debugging ports. This was done with a very elegant way (by using Verilog's conditional statements `ifdef, `else, `endif) and as a result there was no need to change anything else on the rest of the design and tools or re-testing. core_test.v applies the required states on the debug ports, clocks the cpu in order to load the states, then applies the value in the input and then clocks again. After that it takes the new states, the output values on the Data_out bus and compares them with the expected providing detailed error messages in case of mismatch.

It is very important to say that ERRORS EXISTED and where found using this methodology. These errors where quite hard and couldn't be found easily with software test programs. The four errors found where:

1. A typographic mistake: A || instead a && in an expression of the RTL model.

2. A bug that Interrupt Enable was cleared after a pop without IE. In order to use the pop instruction as retie (return with interrupt enable) there is a bit controlling if we will set the interrupt enable. If this bit is 0 the IE flag must remain the same while in our RTL design it was cleared. This error didn't exist in the original behavioral model.

3. A bug that Carry Enable was disabled during Push Status. The push status instruction should push the carry flag as well. The default value of the carry_en signal is 0 and probing for push status wasn't included in its expression. This was an extremely hard mistake.

4. Another bug with carry enable and push status. When the carry flag was one, the value pushed in to the stack was increased by 1 because we used the add operation for moving the flags to the memory. Of course this is a very un-important mistake that could possibly never be found as far as pop status instruction is being used, which discard the LSB's. However it's good to be formal and so it got repaired by using the OR instead of the ADD function.

One case where this could result in a bug would be this:

```
pushw status
movw %ax, (%sp)
addsw $1, %sp
testw $0xA000, %ax
jz FLAGS_OK
```

This is an extreme but still valid way to check that all flags are in a certain state. You don't want this to be the reason your space ship will fail, right?

Having reached that level of insight of our processor we could now proceed with confidence to the layout with small risk of looping back to the behavioral model.

## *4.8 Place and route*

We used L-Edit for placement and routing of the controller. The settings and the process that we have used can be found in Appendix C. We had to modify the labels in our library slightly to fit L-Edit's requirements. Placement and routing with L-Edit is a highly automated process and the margin of mistakes was limited. Despite that it was a process where we had to carefully choose the parameters to have an efficient layout.

Several different floorplaning techniques where considered specially during our initial draft design where we still had a huge 700 gate controller and we also thought that the final pad-ring had to be square and not rectangular. One of them can be seen in the following picture (Image 1) where the (huge) controller is divided in two sub-modules; the main controller module and a decompression module that minimizes the number of wires required to connect the Controller and Datapath (to minimize height overhead). Of course all these problems where finally solved by minimizing the size of the controller and by accepting a non-square shape (Image 2).

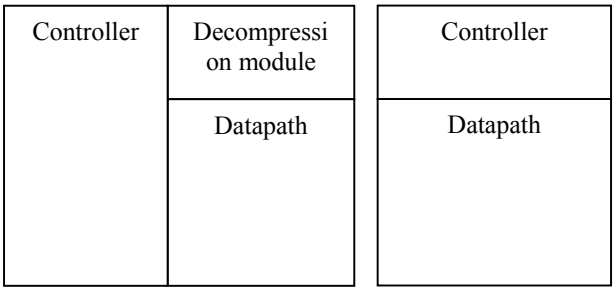
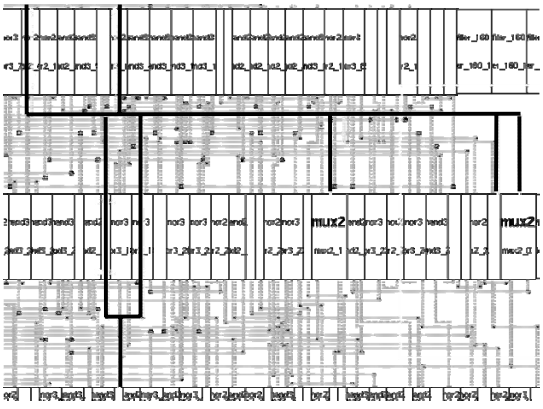



Image 1




Image 2

Image 3



It must be noted that L-Edit't routing is far from perfect. As it can be seen in the above figure (Image 3) it tends to add row-crossers in places that they could be possibly avoided. After careful inspection we concluded that one full row couldn't be saved by correcting these mistakes so we didn't waste our time in these issues.

After going from L-edit back to magic I added the leftbuf and the rightending cells did the inter-row connections of them and polished the design. I modified controller's bottom wiring to match the fingerprints of the datapath as can be seen in the figure bellow.


This way the two modules got electrically connected by just putting the one over the other. No routing was needed in the cpu_core for the datapath control signals. This was a time-consuming process but it was better than auto-routing with magic and then repairing all the errors and inefficiencies. Additionally by routing these signals on controller's area we saved really a lot of space.

The layout of the controller(.mag) can be found in Appendix D.

## *4.9 Final floorplaning*

This section describes the steps followed for routing 'control' and 'data path' to form cpu_core. The obtained cpu_core is then placed and routed in the suitable pad ring to form a complete CPU.

The magic file cpu_core.mag is created. This magic file consists of control and data path hardware obtained using 'getcell' command.

% magic cpu_core

: getcell control

: getcell datapath

Given our control layout, very few extra routing is being needed to complete cpu_core. This was easily hand-made to ensure efficient and faster layout. As we said before, magic's auto routing usually requires a lot of extra effort to repair and optimize it. The final layout of the cpu_core(.mag) can be found in Appendix D.

To complete the microprocessor/CPU design, we create a pad ring using the following command:

create_pad_ring -interrupt <xsize> <ysize>

where xsize and ysize are the x and y dimensions (in um) of the pad ring. The option 'interrupt' creates pad ring with interrupt signal pin nIRQ. The minimum pad ring of 1.4 x 1.4 mm was more than enough for our needs. The obtained pad ring file is renamed to cpu.mag. In the magic file cpu.mag, the cpu_core is place using getcell command;

: getcell cpu_core

The Magic netlist file 'cpu.net' used for routing the cpu_core with the pins provided in the pad ring is edited to ensure that all the ports of the cpu_core is connected to suitable pins provided in pad ring. The magic box tool is used to define the routing area in the magic file cpu.mag. The area defined should extend significantly around the inner free space provided in the pad ring to ensure that all the pins are connected to the ports of the cpu_core with no routing errors. Then on the executing the routing command within magic as shown below,

: route

we obtain the completely routed CPU. To ensure that wide metal tracks are provided for power and ground, we need to first wire the power and ground pads with wide enough metal tracks before carryout routing.

Some hours were spent repairing Magic's errors as usual and making them more efficient mainly by reducing vias. The appropriate selection of positioning of I/Os during L-Edit place and route allowed us to have short wires. We could make it slightly better by putting the ENB pin at the right side of the control so that it has equal distance from all the pins. The final layout of the cpu(.mag) can be found in Appendix D.

## 4.10 Cadence DRC

Now that we have the complete microprocessor layout within the pad ring, it has to pass through the cadence design rule check. You can find a complete step by step guide on how to do cadence DRC on a magic design on Appendix E.

We created and imported to Cadence two versions of the cpu. The one is without the pad-ring (cpu_cadence.mag) and the other is the complete (cpu.mag). Both layouts can be found in Appendix D. A complete view of cpu's layout from within Virtuoso can be found in Appendix D (Cadence view of MOUFA).

The cpu_cadence got DRC'd with all the switches enabled and gave no errors at all. The cpu got DRC'd and gave the following errors (with float? switch on):

| # errors | Violated Rules |
|---|---|
| 32 | r2304: maximum metal1 width: 25u |
| 32 | r2704: maximum metal1 width: 25u |
| 32 | r3404: maximum metal1 width: 25u |
| 4722 | rxxx: floating metal 1 |
| 2459 | rxxx: floating metal 2 |

All of these errors exist inside the 32 I/O pin buffers. The first three and are not very important anyway. We hope that the reset don't exist in Alcatel's I/O pin buffers used for manufacturing. DRC-ing without float? doesn't mention the last two errors.

It must be noted that a small layout error was found during Cadence DRC on the datapath. The flags lines where floating from a level and so on. It got easily repaired by adding a flag_terminal cell between flag_all and flag_dummy that grounds these lines. These errors weren't detected during the previous Cadence DRC on the datapath because the float? switch was not set.

The important fact is that by checking early for DRC errors in our design flow we avoided last minute's surprises. This rule has been taught to me (dkl105) the hard way, in last semester's "Digital IC Design" course.

## 4.11 Simulation & Scanpath verification

We had to simulate all the programs again after this verification. This included modifying the cpu.v file by binding the appropriate net names from the .vnet file to publicly available Verilog

variables. This was a lot of work but we have done it again many times and there were no surprises. Magic's :getnode instruction helped a lot again.

Then coreview.tcl, monitor.v and system.sv, xl_graphics.v had to be modified a little to support the Visualization framework. It is important that NO structural modifications where made. The names of the variables that get probed were just changed. The probing and visualization of the extracted CPU IS COMPLETE which means that we can see all the variables and system parameters we could see in all other cases.

The way one can run the simulations including extracted level can be found in Appendix F. All programs where run again and their results were as expected. Even the most demanding testing program (full_test.asm) was verified to run correctly.

Of course there are some timing violation warnings during the first few ns because of the propagation delays of the gates of the CPU. These are the same delays that we have seen in our cross-simulation on the beginning.

Even though having the programs running is a good sign, there was a lot more testing to be done. A stimulus.v file was written to ensure that the scan path works. It provides a 1 to the SDI and counts the number of clocks till it sees getting out from the SDO. This is the length of our scanpath. Refer to Appendix F to see how to reproduce this simulation.

Even more testing was to be applied on the CPU. extracted_test.v (that can be found in the core_test directory) was developed that feeds the test vectors to CPU via the scanpath and compares the results with the expected ones.

Scanpath's chain was extracted from the layout and verified by the extracted simple scan-path test (stimulus.v) by inspecting the variable changes in each clock cycle. Everything was as expected. Scanpath's structure can be found in Appendix G.

In order to minimize bugs and the effort put in developing extracted_test.v it is heavily based on the core_test.v (which was designed with portability in mind from the beginning). There have been developed some functions that convert from scanpath format to the format used by core_test.v. In order to advance performance, last test's state is being scanned-out at the same time that the new test vector is being scanned in.

The extracted CPU passed successfully all the test vectors. Refer to Appendix F to see how to reproduce this simulation. Obviously these test vectors could be used to test a fabricated processor using a procedure similar to the simulated one.

After all this testing, we can guarantee that our CPU is working according to the specifications. The VLSI design project is a success story.

# 5. Possible improvements

## 6.1 Instruction set improvements

There are very many ways to improve the instruction set. An important consideration while coding the instruction set was to allow spaces for future expansions. There exists free space for instructions that use short immediate form and of course many more for general purpose instruction.

It was apparent when writing the assembler that handling of 8 bit variables was extremely slow. Of course it is an 16-bit architecture so this is not needed and there is no need for the overhead of an 8-bit compatibility mode, but the existence of an operation like this:

```
%ax <= {%ah, 0}
%ax <= {0, %al}
```

This could reduce memory usage for char operations by up to 50%. This was not a priority in a 16-bit processor like ours and that's why we didn't do it.

Another useful thing to support would be rotate left/right so instead of writing:

```
 sr %ax
 jnc SKIP
 or $8000, %ax
SKIP:
```

we could simply write:

```
 rr %ax
```

This could accelerate some communication protocol implementations and could be easily implemented with no overhead on the datapath and just a little bit more complex control. Perhaps in a future version such an addition be considered.

## 6.2 Clock speed improvements

Why MOUFA can't run in 100MHz? There are many reasons. These and possible solutions are discussed in this section.

1. Critical Path

The critical path includes the Carry Chain of the adder. Unfortunately it includes memory's data setup time as well (for operations like addw $CONST, %REG) and the control logic that drives these instructions. Obviously this is a large limiting factor for the total speed of the system. Using a memory with faster setup time will immediately affect the maximum speed that can be achieved by the system but unfortunately just with an additive fashion (If we save 100ns in data setup time we end up with 100ns larger smaller minimum preriod). The use of pipelining could increase the maximum clock frequency but this would conflict for some instructions with the operation of memory resulting in a high ratio of stalls. Of course this would also increase amazingly system's complexity and surly duplicate the area of our processor. The easiest and more area efficient way to improve the maximum frequency for this architecture is to add carry lookahead to the adder.

2. Datapath Control Signal Buffers

The datapath control signals get driven from simple gates. These have a large fanout because of the large number of transistors (XorEN and LoadX signals drive 64) and the large lines they

drive. Buffers should be included to some of these signals if it's identified that they transit too slowly and delay the processor.

3. I/O Buffers and ENB

I/O lines are in the critical path via the memory interface. Although I'm sure I/O pin buffers have the minimum load in their inputs, I/O lines are quite large and some of them with a large number of vias. The ENB signal drives the longest line of the system and at least 32 transistors on the I/O pin buffers. Inserting buffers on these datapath and control signals will be beneficial for the speed of the system.

# 6. Conclusion

The VLSI design project was the most challenging course of the MSc up to now. It gave us the opportunity to use all the knowledge we have gained in the first semester and apply it in a practical problem.

Despite of what we expected, most of our problems existed because we had more freedom and increased responsibilities. Our decisions were going to affect the final result of the project. Wrong decisions of the beginning where appearing as problems throughout the project. Then we had to go back and solve them by repeating the same processes again and again which was a tremendous waste of time and energy. It became apparent that a more forward-looking approach should be applied. The results of our new viewpoint are apparent after the first half of the first part of the implementation phase where most early mistakes have been repaired and important mistakes have been avoided.

The experience gained from this project will help us to take better decisions on future projects and manage better our team's time and resources.

At the end of the day, what we have is a very high quality microprocessor design, a great set of tools and documentation to support its developer and the very important experience of a VLSI design project.

# Appendixes

## *Appendix A. Brainstorming*

The random number generation program implemented with arbitrary instructions of a Machine of Unknown, Future Architecture.

```
Random
         mov            SP, #65535  ; Initialize SP to max address

         mov            P, #2048    ; Address of switches
         mov            A, [P]
main:    push           A
         push           #0
         call           random
         pop            A
         mov            P, #2560    ; Address of leds
         mov            [P], A      ; Show the LSB on the leds
         pop            B

         jmp            main        ; Infinite loop

// the stack will be like this when entering this
// function:
// SP+3  ->             Seed
// SP+2  ->             Result placeholder LSB
// SP+1  ->             Return address
// SP+0  ->             Unknown
random:
         // PUSH EVERYTHING USED HERE (A,B,I,SP,Status)
         add            SP, #3
         mov            A, [SP]     ; A <- Seed
         mov            I, #16      ; 15 loops
         clrc

rlo:     rrc            A           ; Rotate right with carry
         mov            B,A         ; Backup A in B – Doesn't affect C
         and            A,#1        ; LSB zero?     – Doesn't affect C

         jmpcz          rcz
                                    ; X-LSB non-zero
         jnz            A, rxor0    ; Xor result is 0
         jmp            rxor1       ; Xor result is 1

rcz:                                ; X-LSB zero
         jnz            A,rxor1     ; Xor result is 1

rxor0:   mov            A, B        ; Xor result is 0 - Restore A
         jmp            endl        ; Goto end of loop

rxor1:   mov            A, B
         or             A, # 16384  ; Set the 15th bit
```

```
endl:    decijnz        rlo         ; End of loop
         sub            SP, #1
         mov            [SP], A     ; Return A
         // POP (A,B,I,SP,Status)
      ret
```

**MOUFA: Machine Of Unknown Future Architecture**

Minimum instruction set:

```
mov      A|C|I|P|SP, #immediate     ; load immediate to registers
mov      A, [P|SP]                  ; load from stack to A or B
mov      [P|SP], A                  ; store A to pointer location
mov      (A,B,C)|(A,B,C)            ; inter-register move

and      A, #immediate              ; A<-A and immediate
or       A, #immediate              ; A<-A or immediate
sub      A, #immediate              ; A<-A-#i
add      A, B                       ; unsigned addition A<-A+B
add      SP, #immediate             ; unsigned subtraction SP<-SP+#i
sub      SP, #immediate             ; unsigned subtraction SP<-SP-#i

rrc      A|C                        ; rotate right with carry
clrc                                ; clear carry flag

push     A|B|C|I|SP|Status          ; push value of A to the stack
push     #immediate                 ; push immediate value
pop      A|B|C|I|SP|Status          ; pop value of A

decijnz  offset                     ; decrease I jump while not zero
jmp      offset                     ; unconditional jump
jnz      A, offset                  ; jump if A non-zero
jmpcz    offset                     ; jump if carry is zero
call     offset                     ; call function
ret                                 ; return from subroutine

offset -> 16 bit signed offset
immediate -> 16 bit constant
```

Possible additions:

```
call     A|B|C                      ; Indirect call useful for cases
retie                               ; Return with interrupt enable
mov      A, [#imediate]             ; Direct address mem
mov      [#imediate], A
mov      A, [#imediate+I]           ; Direct address with offset mem
mov      [#imediate+I], A
mov      A, [(B|C)+I]               ; Indirect address with offset mem
mov      [(B|C)+I], A
pop      #imediate                  ; Discard #i words from stack
mov      B|C|I, […]                 ; mem load to registers
mov      […], B|C|I
rrl      A|C                        ; Rotate left through carry
```

## *Appendix B. Instruction set and datapath early specification. Instruction scheduling.*

**INSTRUCTION SET**

| | | |
|---|---|---|
| 1. MOV reg-immediate | 14. addc reg – immediate | 26. push immediate |
| 2. MOV reg-reg | 15. add_small reg – immediate | 27. push reg |
| 3. MOV reg-{reg} | 16. sub_small reg – immediate | 28. pop |
| 4. MOV {reg} – reg | | 29. decijnz offset |
| 5. AND reg-immediate | 17. sub reg – reg | 30. jump offset |
| 6. AND reg –reg | 18. sub reg – immediate | 31. jnz reg,offset |
| 7. OR reg – immediate | 19. subb reg –reg | 32. jz reg,offset |
| 8. OR reg- reg | 20. subb reg – immediate | |
| 9. xor reg-immediate | 21. rrc reg | 33. call offset |
| 10. xor reg – reg | 22. rlc reg | 34. call reg |
| | 23. rol reg | 35. jump reg |
| 11. add reg – reg | 24. ror reg | 36. return |
| 12. add reg – immediate | 25. not reg | 37. retie |
| 13. addc reg – reg | | 38. enable_interrupt |
| | | 39. disable_interrupt |

**MOUFA INSTRUCTION SET DETAILS**

Datapath building blocks:

```
load_bus_1_from(DATA_BUS);
load_bus_1_from(IR[2:0]);
load_bus_1_from(IR[5:3]);
load_bus_1_from(PC);
load_bus_1_from(SP);
load_bus_1_from(sign_extend(IR[13:6]));
load_bus_1_from(STATUS_BITS);

load_bus_2_from(IR[5:3]);
load_bus_2_from(sign_extend(IR[13:6]));
load_bus_2_from(CONSTANT_1);
load_bus_2_from(CONSTANT_0);

store_alu_res_to(IR[5:3]);
store_alu_res_to(DATA_BUS);
store_alu_res_to(STATUS_BITS);
store_alu_res_to(SP);
store_alu_res_to(PC);

alu_mode(ADD_SIGNED);
alu_mode(AND);
alu_mode(OR);
alu_mode(XOR);
alu_mode(ADD);
alu_mode(SUB);
alu_mode(SR);
alu_mode(NOT);
```

```
carry_en(1); // An and gate
carry_en(0);

PC++;
--i; // With Zero Flag Output

ie=1/0;
C=1/0;

store_all_flags();
store_zero_flag();
store_zero_and_carry_flag();
```

**Instruction set implementation**

**40. MOV reg-immediate**

```
word 1:
       ICODE   OP2   OP1
[0000000001] [010] [000] (0x0050)
word 2:
               Immediate
                   42405

example: 0050 A5A5
```

```
if (is_mov_reg_imm(IR)) {
      mem_read_cycle {
            // In address setup step
            load_bus_1_from(PC);
            load_bus_2_from(CONSTANT_0);
            alu_mode(OR);
            store_alu_res_to(DATA_BUS);
            PC++;
            // In data setup step
            load_bus_1_from(DATA_BUS);
            load_bus_2_from(CONSTANT_0);
            alu_mode(OR);
            store_alu_res_to(IR[5:3]);
      }
}
```

**41. MOV reg-reg**

```
If (is_mov_reg_reg(IR)) {
      load_bus_1_from(IR[2:0]);
      load_bus_2_from(CONSTANT_0);
      alu_mode(OR)
      store_alu_res_to(IR[5:3]);
      store_zero_flag();
}
```

**42. MOV reg-{reg}, offset**

```
If (is_mov_reg_addr_reg(IR)) {
      mem_read_cycle {
            // In address setup step;
            load_bus_1_from(IR[2:0]);
            load_bus_2_from(sign_extend(IR[13:6]));
            alu_mode(ADD_SIGNED);
            store_alu_res_to(DATA_BUS);
            // In data setup step
            load_bus_1_from_(DATA_BUS);
            load_bus_2_from(CONSTANT_0);
            alu_mode(OR);
            store_alu_res_to(IR[5:3]);
            store_zero_flag();
      }
}
```

…

**The final instruction coding**

| | OPCODE | ATE | SIMPLE | ADDER\|OR | LOGnARI | Target (OP2) | | Source (OP1) | TargetNull | CarryEn | SUB_MODE\|AND | Sr\|not | ForceBus2To0 | IMMnREG | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CALL short_offset | **1** | **0** | **1** | **1** | | | | offset12 | | | | | | | |
| ADD reg – short_immediate | **1** | **1** | **1** | **0** | tgt3 | | | | imm9 | | | | | | |
| MOV reg – short_immediate | 1 | 1 | 1 | 1 | tgt3 | | | | imm9 | | | | | | |
| JIF [Z,C,OF,?], ne, short_offset | **1** | **1** | **0** | **-** | fl2 | n e | | | offset9 | | | | | | |
| DECIJNQ short_offset | 1 | 0 | 1 | 0 | | | | offset12 | | | | | | | |
| MOV reg-{reg}, short_offset | 1 | 0 | 0 | 0 | tgt3 | | src3 | | | off6 | | | | | |
| MOV {reg} – reg, short_offset | 1 | 0 | 0 | 1 | tgt3 | | src3 | | | off6 | | | | | |
| **MOV reg-immediate** | 0 | 1 | 1 | 1 | tgt3 | | - | 0 | 0 | 1 | 1 | 0 | 1 | **imm16** | |
| AND reg-immediate | 0 | 1 | 0 | 1 | tgt3 | | - | 0 | 1 | 0 | 1 | 0 | 1 | imm16 | |
| OR reg – immediate | 0 | 1 | 1 | 1 | tgt3 | | - | 0 | 0 | 0 | 1 | 0 | 1 | imm16 | |
| XOR reg-immediate | 0 | 1 | 1 | 0 | tgt3 | | - | 0 | 1 | 0 | 1 | 0 | 1 | imm16 | |
| ADD reg – immediate | 0 | 1 | **0** | **0** | tgt3 | | - | **1** | **0** | 0 | 1 | 0 | 1 | imm16 | |
| ADDC reg – immediate | 0 | 1 | **0** | **0** | tgt3 | | - | **1** | **0** | 0 | 1 | 1 | 1 | imm16 | |
| SUB reg – immediate | 0 | 1 | **1** | **0** | tgt3 | | - | **1** | **0** | 0 | 1 | 0 | 1 | imm16 | |
| SUBB reg – immediate | 0 | 1 | **1** | **0** | tgt3 | | - | **1** | **0** | 0 | 1 | 1 | 1 | imm16 | |
| CMP reg-immediate | 0 | 1 | **1** | **0** | tgt3 | | - | **1** | **0** | 0 | 1 | 0 | 0 | imm16 | |

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CMPC reg-immediate | 0 | 1 | **1** | **0** | tgt3 | | - | | **1** | **0** | 0 | 1 | 1 | 0 | imm16 | | |
| SR reg-reg | 0 | 1 | 0 | 1 | tgt3 | | src3 | | 1 | 0 | 0 | 0 | 0 | 1 | | | |
| SRC reg-reg | 0 | 1 | 0 | 1 | tgt3 | | src3 | | 1 | 0 | 0 | 0 | 1 | 1 | | | |
| NOT reg - reg | 0 | 1 | **1** | **0** | tgt3 | | src3 | | **0** | **0** | 0 | 0 | 0 | 1 | | | |
| **MOV reg-reg** | 0 | 1 | 1 | 1 | tgt3 | | src3 | | 0 | 0 | 1 | 0 | 0 | 1 | | | |
| AND reg –reg | 0 | 1 | 0 | 1 | tgt3 | | src3 | | 0 | 1 | 0 | 0 | 0 | 1 | | | |
| OR reg- reg | 0 | 1 | 1 | 1 | tgt3 | | src3 | | 0 | 0 | 0 | 0 | 0 | 1 | | | |
| XOR reg – reg | 0 | 1 | 1 | 0 | tgt3 | | src3 | | 0 | 1 | 0 | 0 | 0 | 1 | | | |
| ADD reg – reg | 0 | 1 | **0** | **0** | tgt3 | | src3 | | **1** | **0** | 0 | 0 | 0 | 1 | | | |
| ADDC reg – reg | 0 | 1 | **0** | **0** | tgt3 | | src3 | | **1** | **0** | 0 | 0 | 1 | 1 | | | |
| SUB reg – reg | 0 | 1 | **1** | **0** | tgt3 | | src3 | | **1** | **0** | 0 | 0 | 0 | 1 | | | |
| SUBB reg –reg | 0 | 1 | **1** | **0** | tgt3 | | src3 | | **1** | **0** | 0 | 0 | 1 | 1 | | | |
| CMP reg-reg | 0 | 1 | **1** | **0** | tgt3 | | src3 | | **1** | **0** | 0 | 0 | 0 | 0 | | | |
| CMPC reg-reg | 0 | 1 | **1** | **0** | tgt3 | | src3 | | **1** | **0** | 0 | 0 | 1 | 0 | | | |
| TEST reg | 0 | 1 | **1** | **0** | - | | src3 | | **1** | **0** | 1 | 0 | 0 | 0 | | | |
| | | | | | FLAG_ID | | NEW_FLAG | | SET_FLAG | POP | POP | PUSH | REG_USE | | | | |
| PUSH reg | 0 | 0 | 1 | 1 | - | | src3 | | - | 0 | **0** | **0** | 0 | 0 | | | |
| PUSH status | 0 | 0 | 1 | 1 | | - | | | - | 0 | **0** | **0** | 0 | 1 | | | |
| CALL reg | 0 | 0 | 1 | 1 | - | | src3 | | - | 0 | **0** | **0** | 1 | 0 | | | |
| CALL offset | 0 | 0 | 1 | 1 | | - | | | - | 0 | **0** | **0** | 1 | 1 | imm16 | | |
| POP reg | 0 | 0 | - | 0 | tgt3 | | - | | SET_IE | 1 | **0** | **0** | - | 1 | | | |
| POP status | 0 | 0 | - | 0 | | - | | | - | 1 | **0** | **0** | - | 0 | | | |
| JIFAR [Z,C,OF,?], ne, offset | 0 | 0 | - | 0 | fl2 | ne | - | | - | 0 | **1** | **0** | 0 | 0 | imm16 | | |
| SETC | 0 | 0 | - | 0 | | - | | | - | 0 | **0** | **1** | 0 | ? | | | |
| STIE | 0 | 0 | - | 0 | | - | | | - | 0 | **0** | **1** | 1 | 1 | | | |
| LDIE | 0 | 0 | - | 0 | | - | | | - | 0 | **0** | **1** | 1 | 0 | | | |
| NOP | 0 | 0 | 0 | 0 | - | - | - | - | - | - | - | 0 | **0** | **0** | - | - | |

## *Appendix C. L-Edit place & route*

**L-edit use case:**

File -> Import mask data -> Cif...
From file...
Z:\design\fcde\verilog\place\cell_lib.cif
Generate new layers...
Import

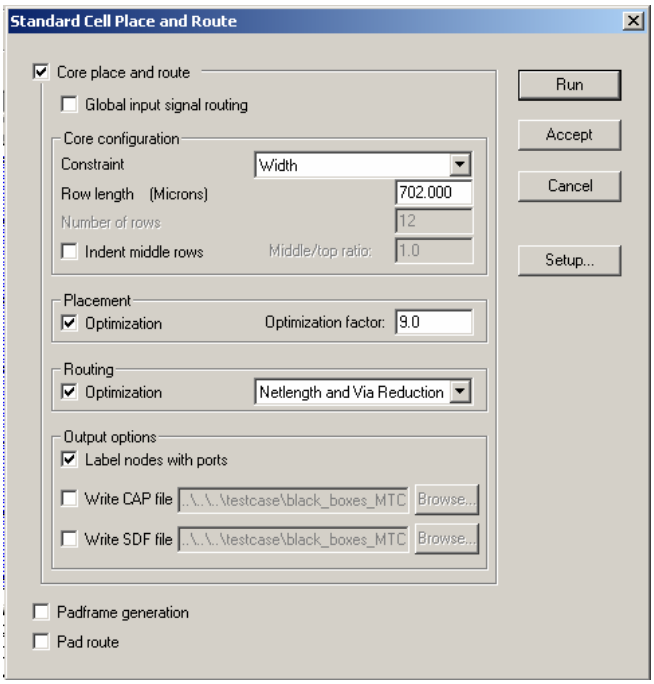
Save... "cell_lib.tdb" -> OK

Open: cmos05.tdb
Tools->SPR->Setup
Standard cell library file: cell_lib.tdb
NEtlist file: control.tpr
Overwrite existing pad route specifications?.. Yes OK

Tools->SPR->Place and route [For options see bellow]
File -> Export mask data -> CIF…-> Export -> OK -> OK

Then we use the extract_cif_library to go from the cif file to a magic layout.

**Final place and route settings:**

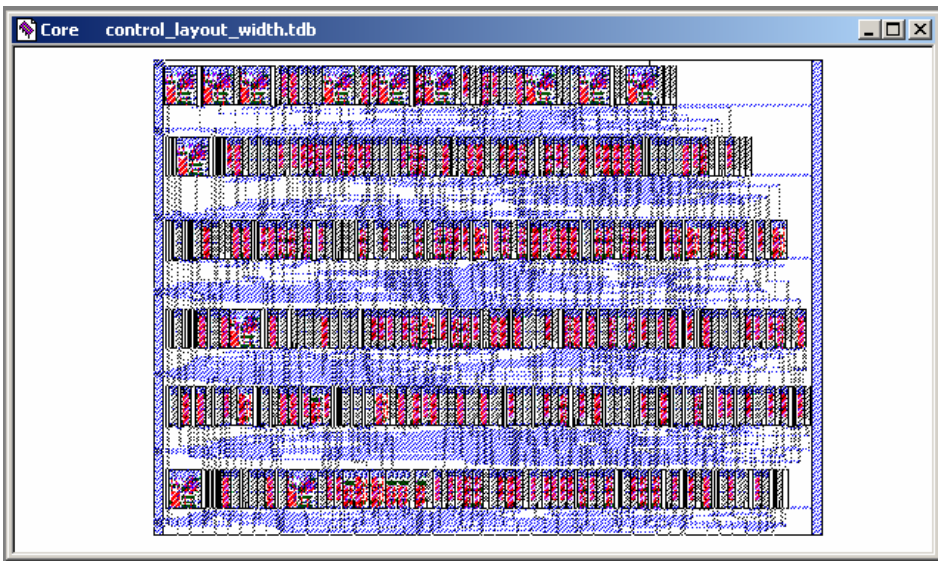



**Pins arrangement:**

| # | NAME | SIDE | # | NAME | SIDE | # | NAME | SIDE |
|---|---|---|---|---|---|---|---|---|
| 1 | Test | BOT | 26 | ALUmode_1_ | BOT | 4 | flags_from_pop_0_ | LEFT |
| 2 | SDI | BOT | 27 | ALUmode_2_ | BOT | 3 | flags_from_pop_1_ | LEFT |
| 3 | LoadIR | BOT | 28 | nZout | BOT | 2 | flags_from_pop_2_ | LEFT |
| 4 | se_8_6 | BOT | 29 | LoadA | BOT | 1 | flags_from_pop_3_ | LEFT |
| 5 | se_sign | BOT | 30 | EnA2 | BOT | 16 | IR_0_ | LEFT |
| 6 | se_11_9 | BOT | 31 | EnA1 | BOT | 15 | IR_1_ | LEFT |
| 7 | en2SE | BOT | 32 | LoadB | BOT | 6 | IR_10_ | LEFT |
| 8 | const_val | BOT | 33 | EnB2 | BOT | 5 | IR_11_ | LEFT |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 9 | en2_cont | BOT | 34 | EnB1 | BOT | 4 | IR_12_ | LEFT |
| 10 | en1_const | BOT | 35 | LoadC | BOT | 3 | IR_13_ | LEFT |
| 11 | flags_out_1_ | BOT | 36 | EnC2 | BOT | 2 | IR_14_ | LEFT |
| 12 | flags_out_2_ | BOT | 37 | EnC1 | BOT | 1 | IR_15_ | LEFT |
| 13 | flags_out_3_ | BOT | 38 | LoadSP | BOT | 14 | IR_2_ | LEFT |
| 14 | en1ST | BOT | 39 | EnSP2 | BOT | 13 | IR_3_ | LEFT |
| 15 | SrLC | BOT | 40 | EnSP1 | BOT | 12 | IR_4_ | LEFT |
| 16 | ALUmode_0_ | BOT | 41 | IncPC | BOT | 11 | IR_5_ | LEFT |
| 17 | SrC | BOT | 42 | LoadPC | BOT | 10 | IR_6_ | LEFT |
| 18 | flags_out_0_ | BOT | 43 | EnPC2 | BOT | 9 | IR_7_ | LEFT |
| 19 | Cpre | BOT | 44 | EnPC1 | BOT | 8 | IR_8_ | LEFT |
| 20 | Cout | BOT | 45 | zero_i | BOT | 7 | IR_9_ | LEFT |
| 21 | ALUmode_4_ | BOT | 46 | decI | BOT | 19 | NIRQ | LEFT |
| 22 | ALUmode_6_ | BOT | 47 | LoadI | BOT | 17 | nOE | LEFT |
| 23 | ALUmode_5_ | BOT | 48 | EnI2 | BOT | 18 | nRW | LEFT |
| 24 | ALUmode_3_ | BOT | 49 | EnI1 | BOT | 2 | ALE | TOP |
| 25 | Nout | BOT | 50 | en1DB | BOT | 1 | ENB | TOP |

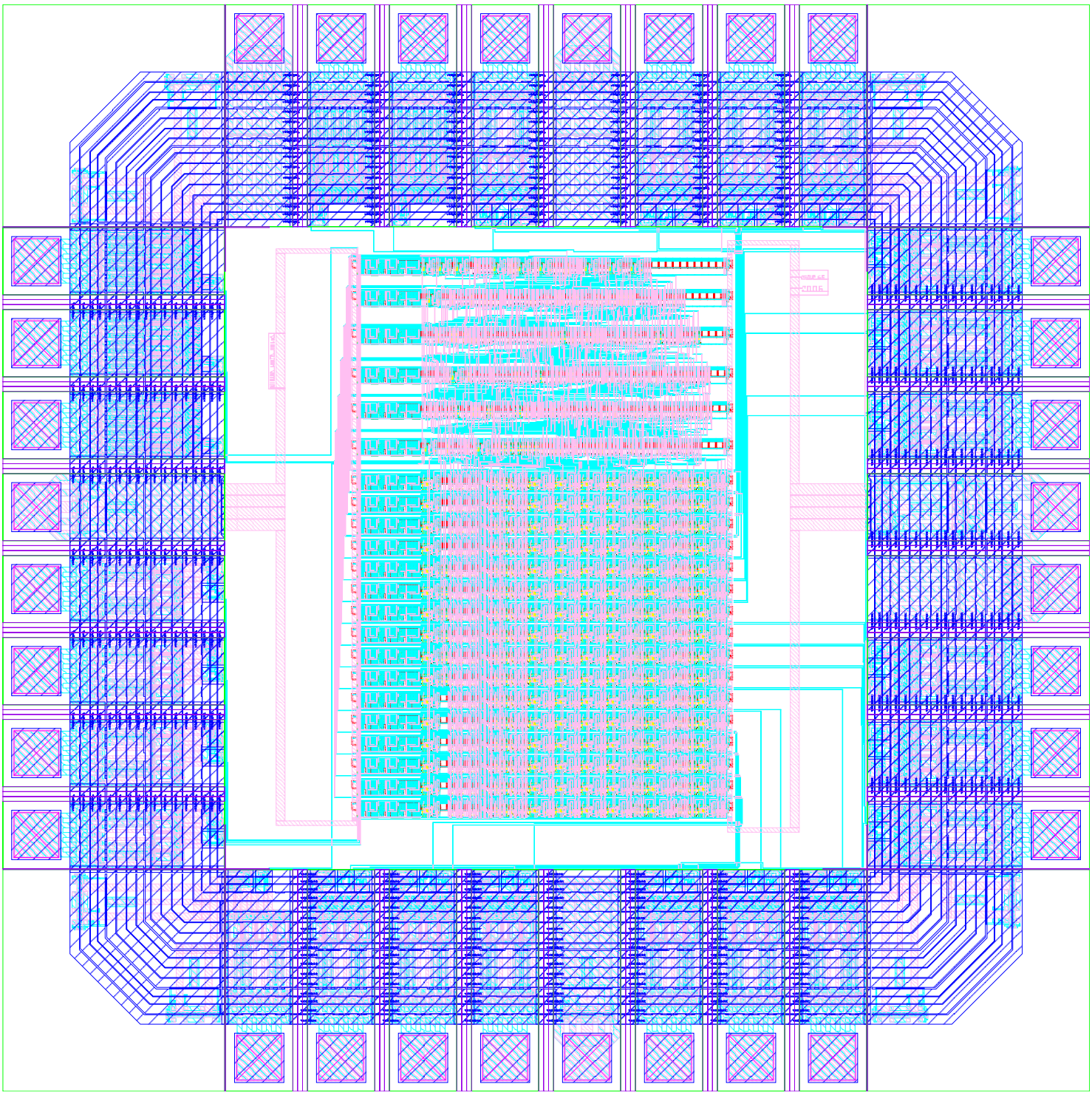
**Resulting layout:**

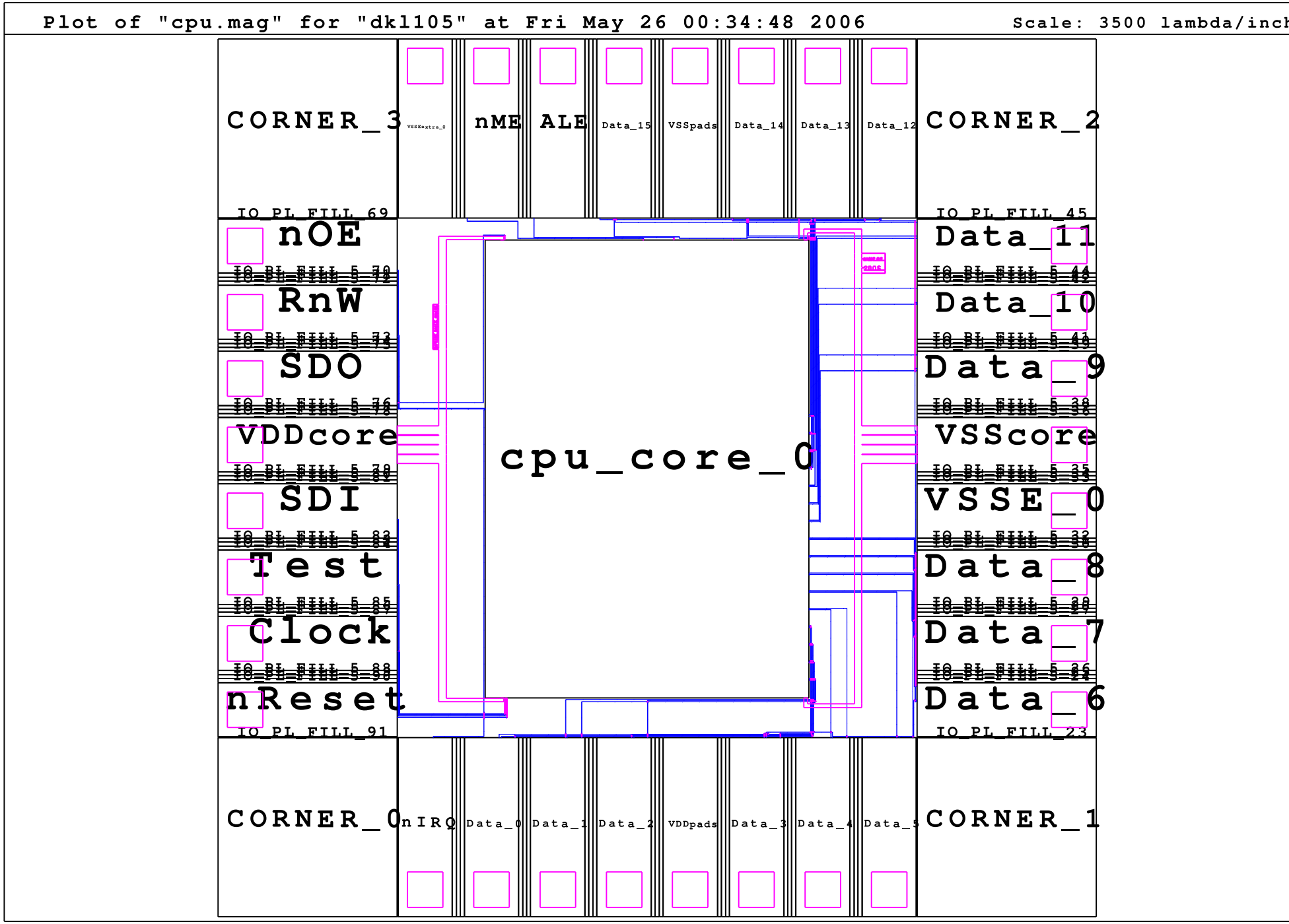



**Area and statistics:**

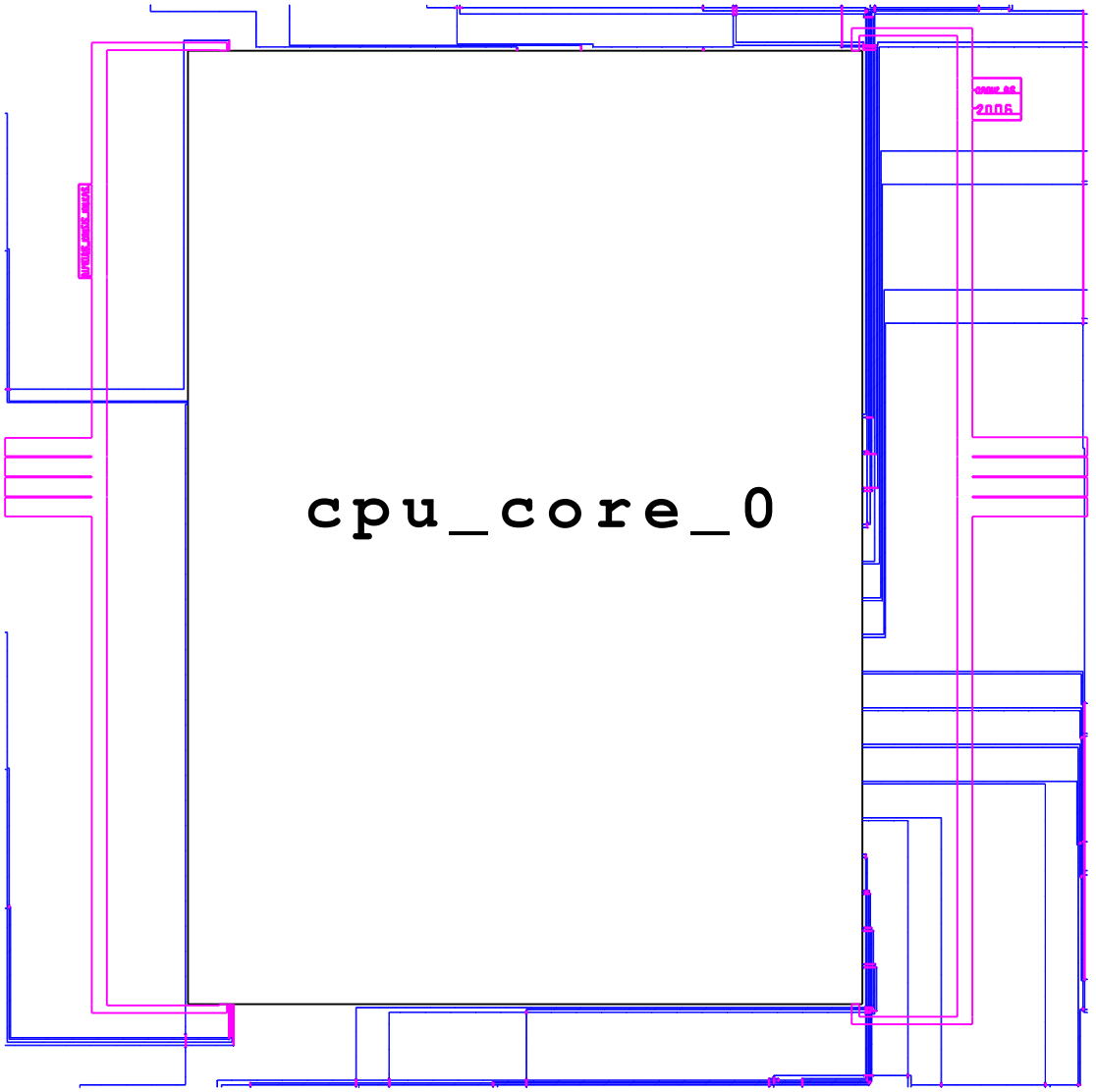
Area = 9947.01 Microns^2
MBB = 674.05 Microns x 476.1 Microns
Density = 3.09938%

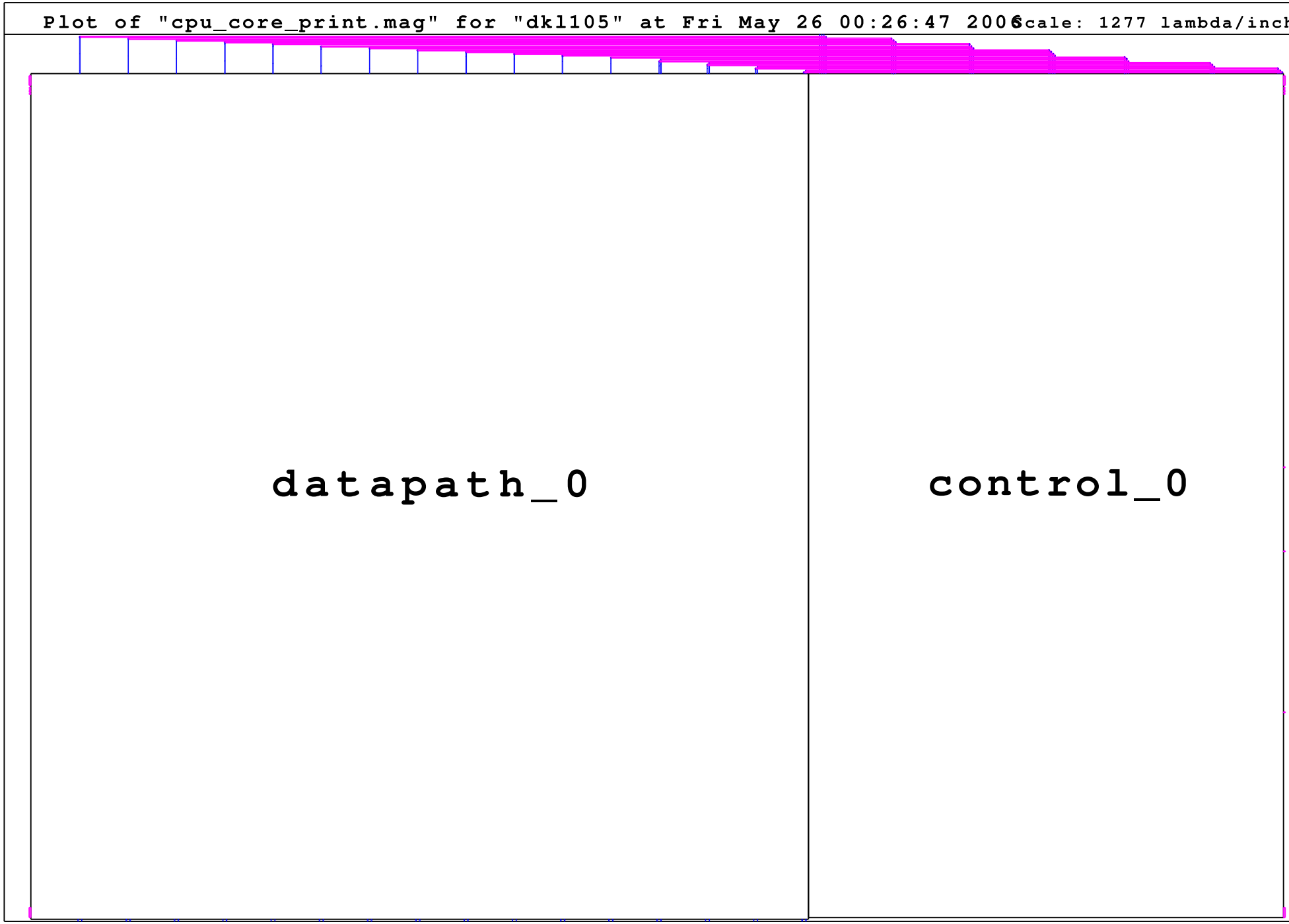
## *Appendix D. Layouts*



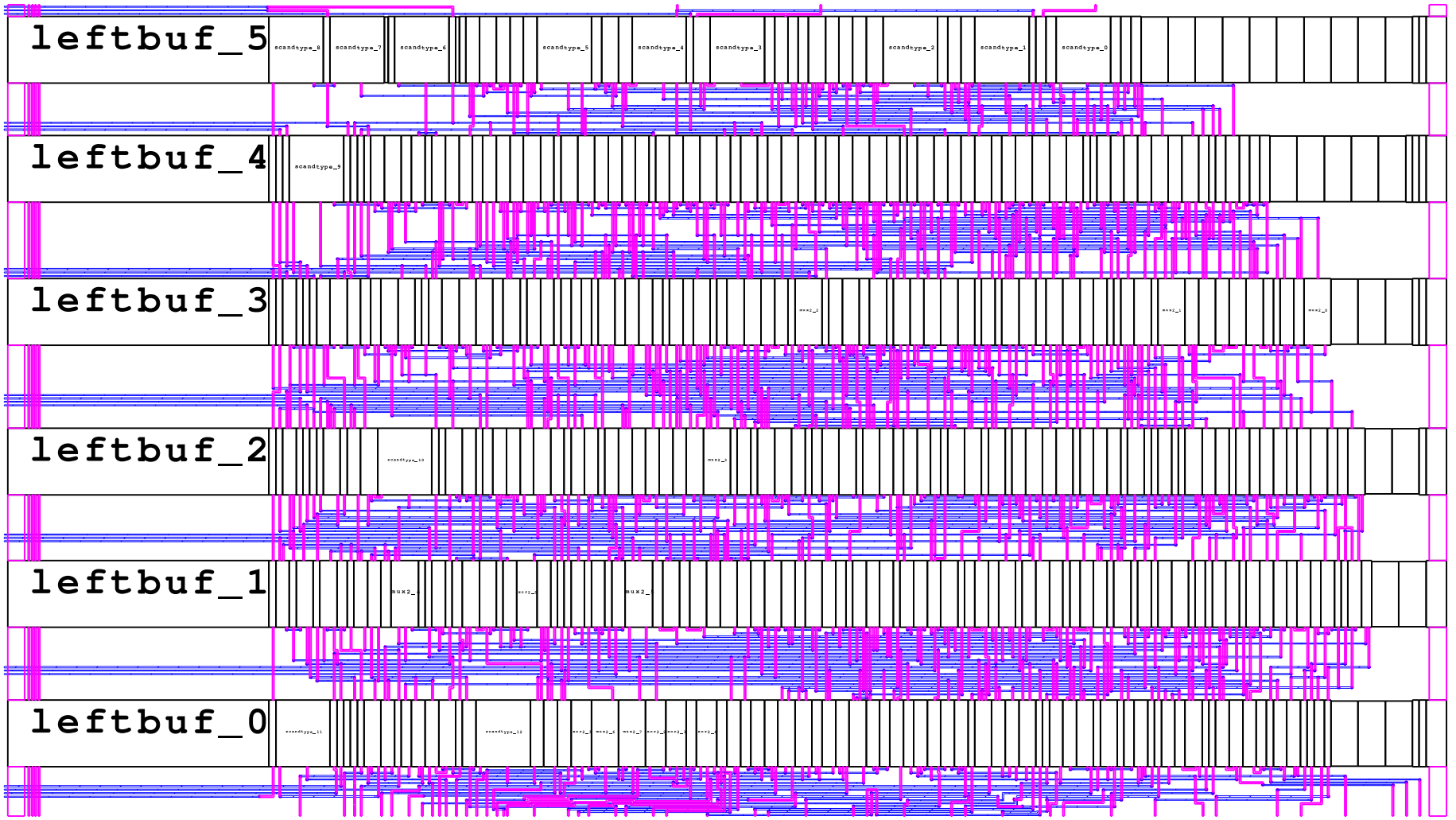

Plot of "cpu.mag" for "dkl105" at Fri May 26 00:34:48 2006
Scale: 3500 lambda/inch
CORNER_3
nME
ALE
Data_15
VSSpads
Data_14
Data_13
Data_12
CORNER_2
IO_PL_FILL_69
IO_PL_FILL_45
nOE
Data_11
RnW
Data_10
SDO
Data_9
VDDcore
cpu_core_0
VSScore
SDI
VSSE_0
Test
Data_8
Clock
Data_7
nReset
Data_6
IO_PL_FILL_91
IO_PL_FILL_23
CORNER_0
nIRQ
Data_0
Data_1
Data_2
VDDpads
Data_3
Data_4
Data_5
CORNER_1

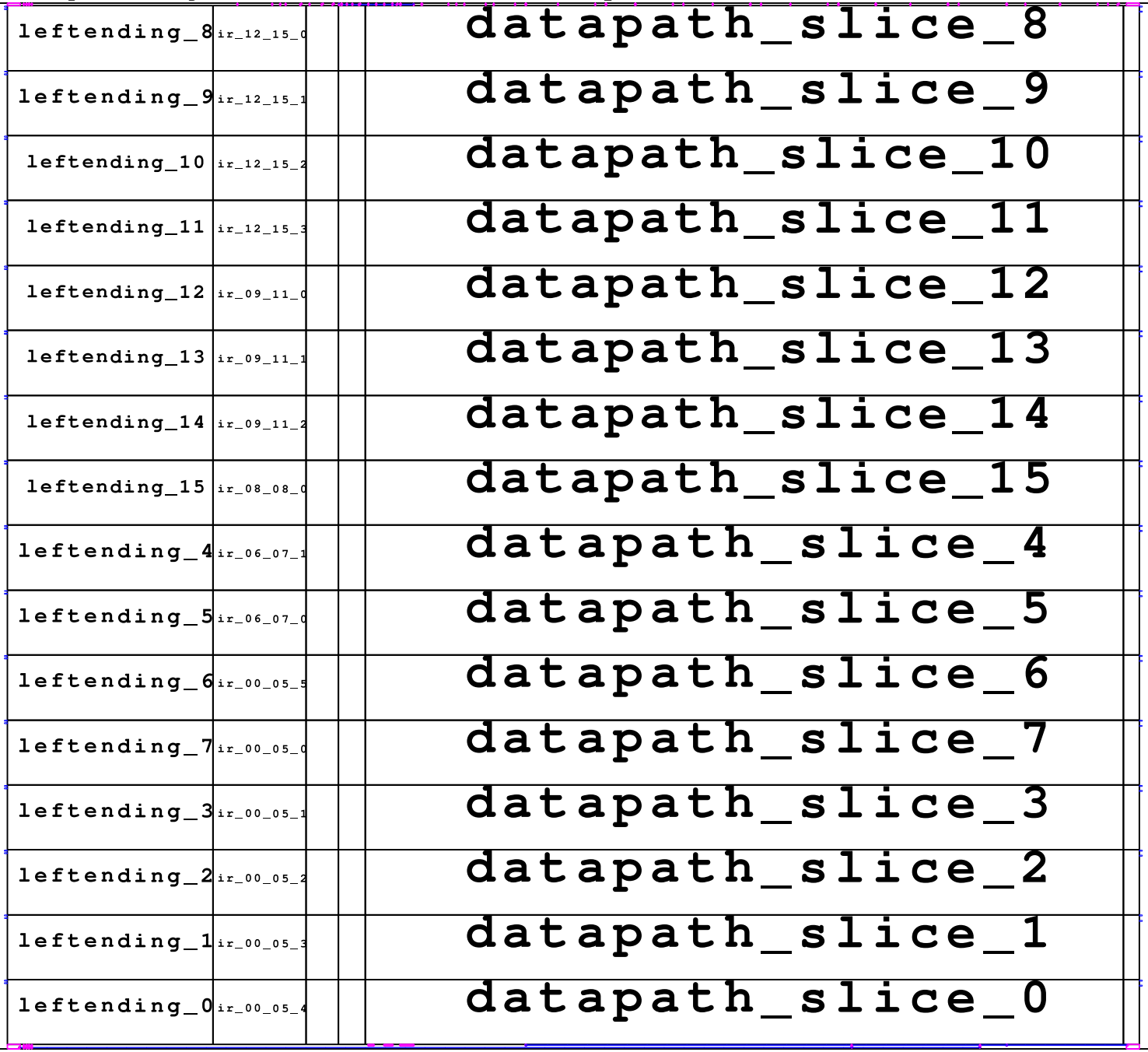
Plot of "cpu_cadence.mag" for "dkl105" at Fri May 26 00:35:52 2006 Scale: 2500 lambda/inch



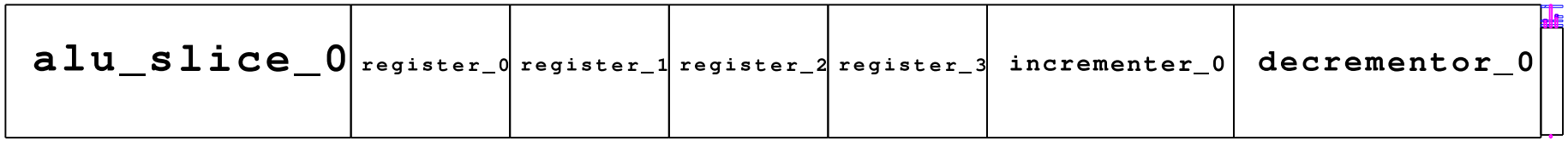

Plot of "cpu_core_print.mag" for "dkl105" at Fri May 26 00:26:47 2006
Scale: 1277 lambda/inch
datapath_0
control_0

Plot of "control.mag" for "dkl105" at Fri May 26 00:26:59 2006 Scale: 900 lambda/inch

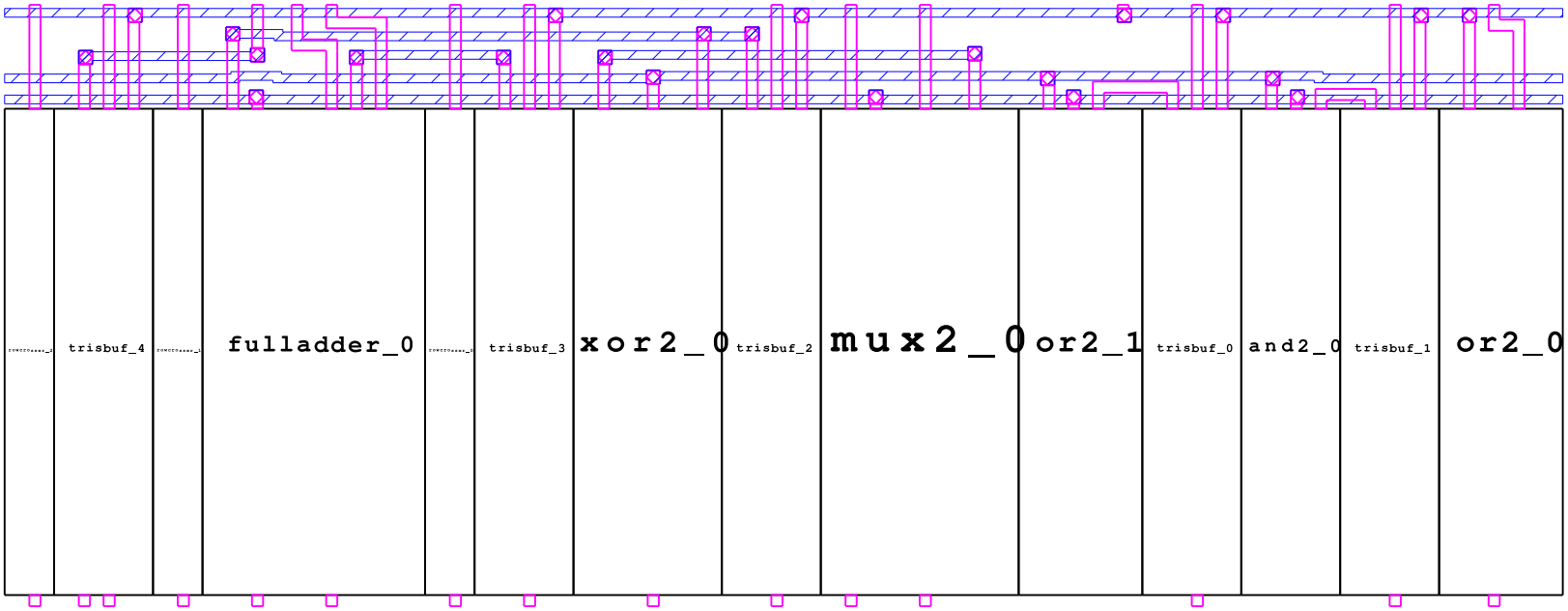



leftending_8
ir_12_15_0
datapath_slice_8
leftending_9
ir_12_15_1
datapath_slice_9
leftending_10
ir_12_15_2
datapath_slice_10
leftending_11
ir_12_15_3
datapath_slice_11
leftending_12
ir_09_11_0
datapath_slice_12
leftending_13
ir_09_11_1
datapath_slice_13
leftending_14
ir_09_11_2
datapath_slice_14
leftending_15
ir_08_08_0
datapath_slice_15
leftending_4
ir_06_07_1
datapath_slice_4
leftending_5
ir_06_07_0
datapath_slice_5
leftending_6
ir_00_05_5
datapath_slice_6
leftending_7
ir_00_05_0
datapath_slice_7
leftending_3
ir_00_05_1
datapath_slice_3
leftending_2
ir_00_05_2
datapath_slice_2
leftending_1
ir_00_05_3
datapath_slice_1
leftending_0
ir_00_05_4
datapath_slice_0

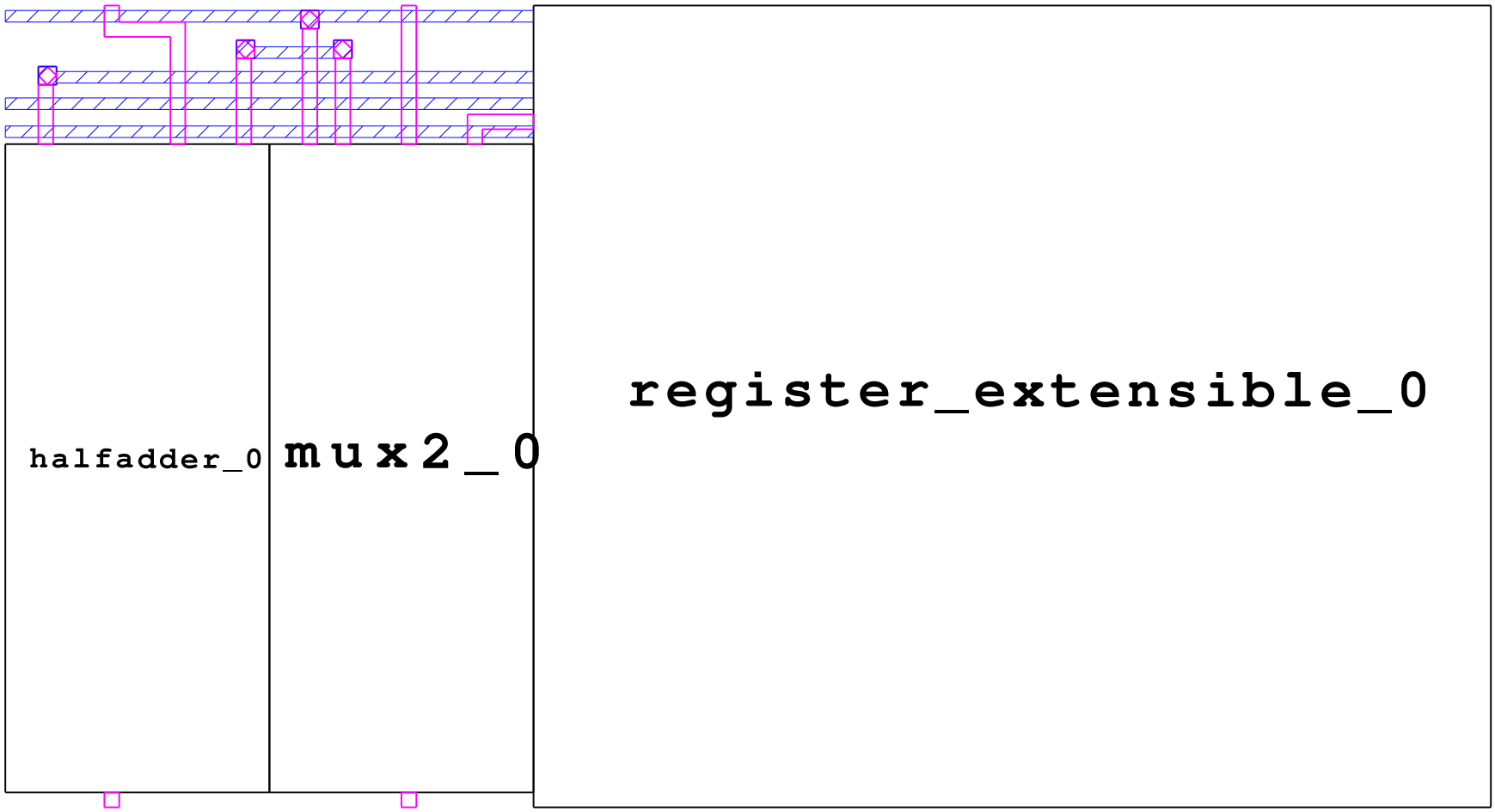
Plot of "datapath_slice.mag" for "dkl105" at Fri May 26 00:30:11 2006 Scale: 600 lambda/inch



Plot of "alu_slice.mag" for "dkl105" at Fri May 26 00:31:47 2006 Scale: 150 lambda/inch

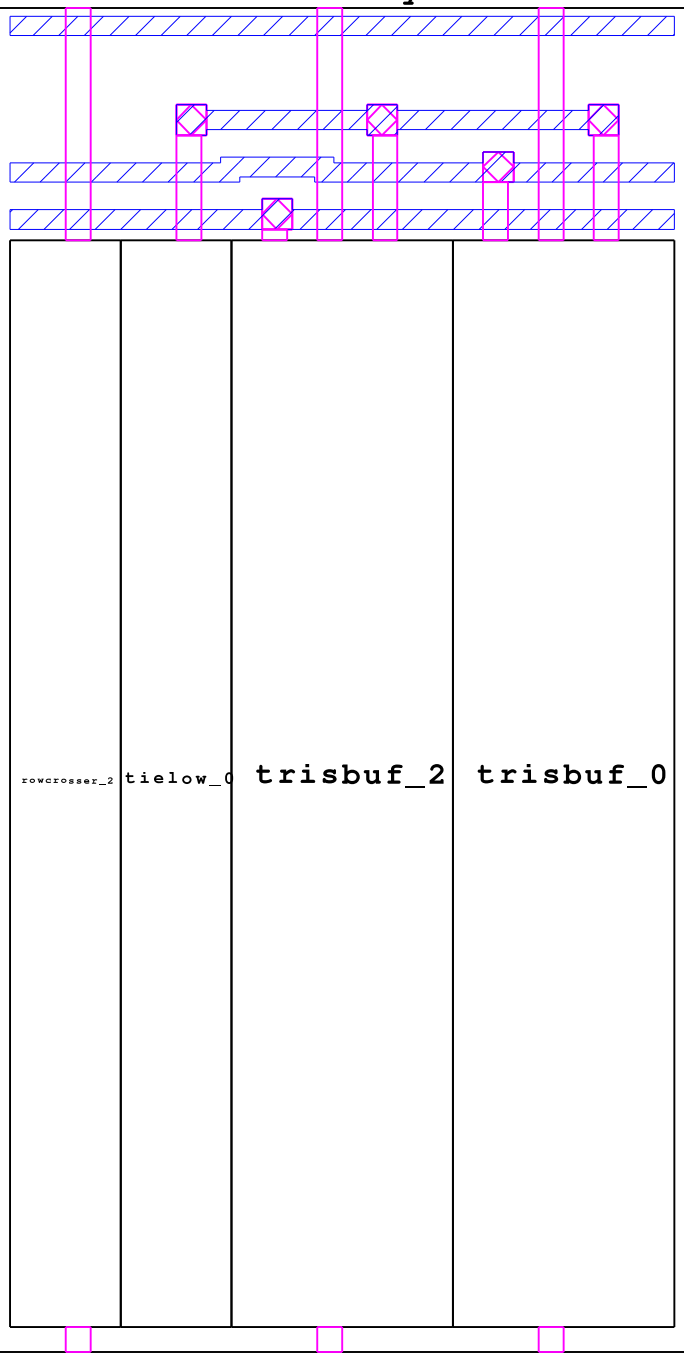


Plot of "incrementer.mag" for "dkl105" at Fri May 26 00:45:58 2006 Scale: 100 lambda/inch

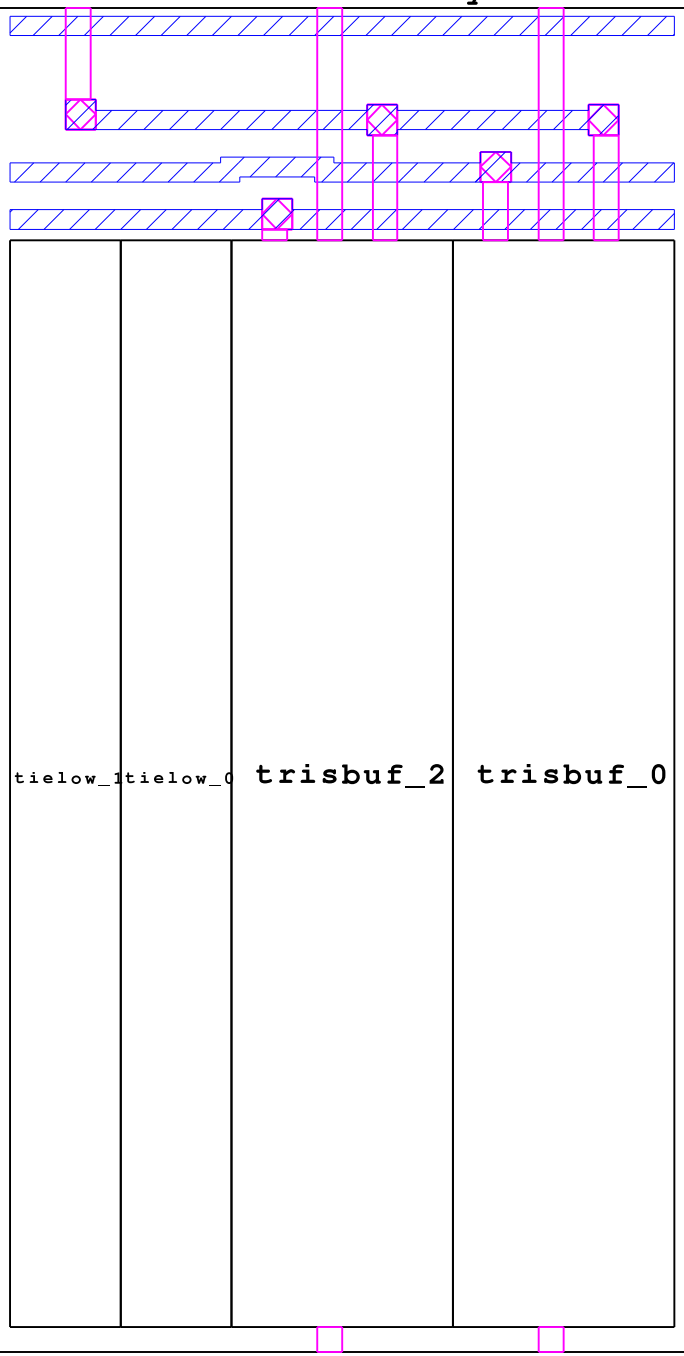


Plot of "decrementor.mag" for "dkl105" at Fri May 26 00:45:53 2006 Scale: 120 lambda/inch

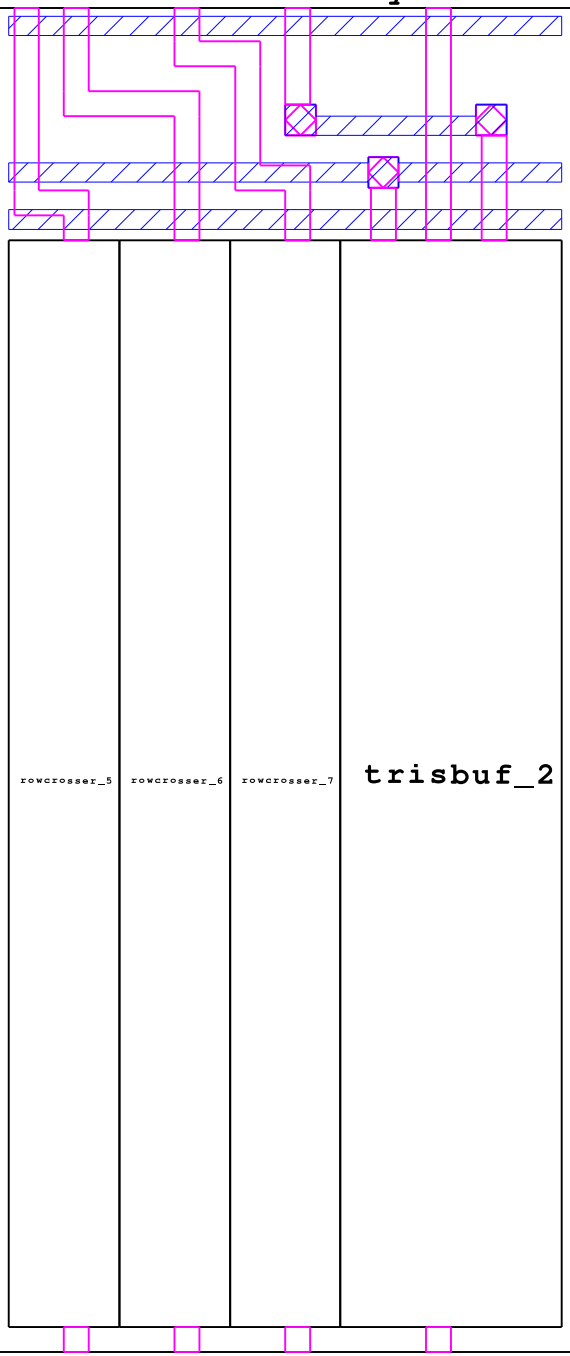
Plot of "const_all.mag" for "dkl105" at Fri May 26 00:37:13 2006 Scale: 69 lambda/inch



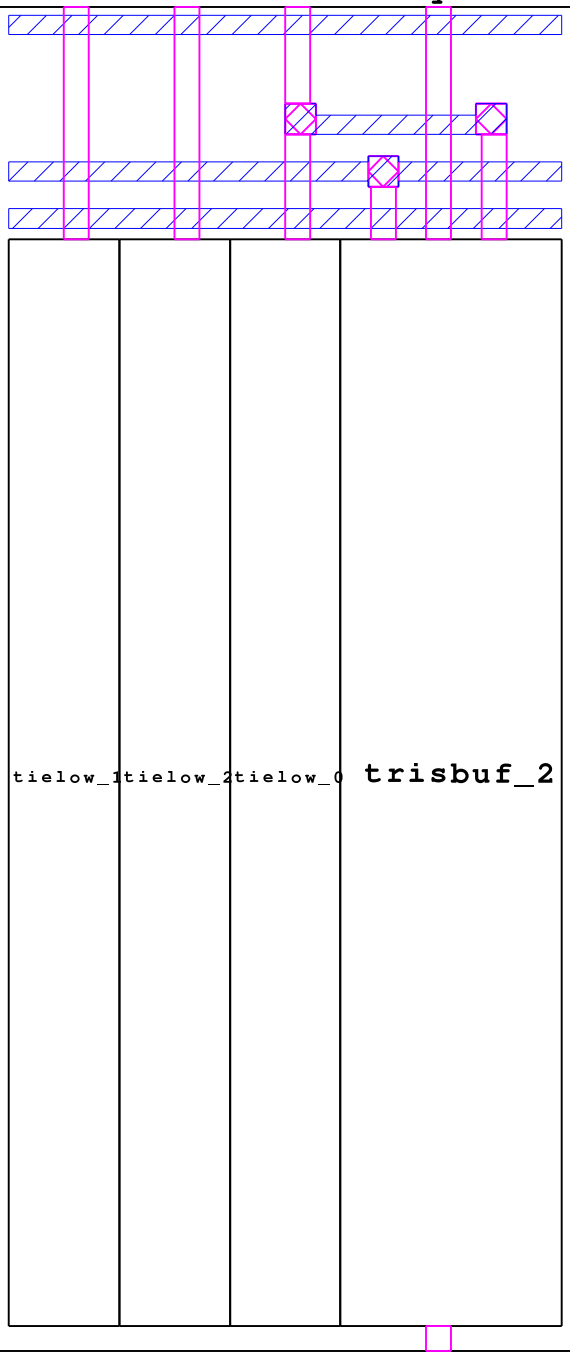
Plot of "const_bottom.mag" for "dkl105" at Fri May 26 00:37:17 2006 Scale: 69 lambda/inch



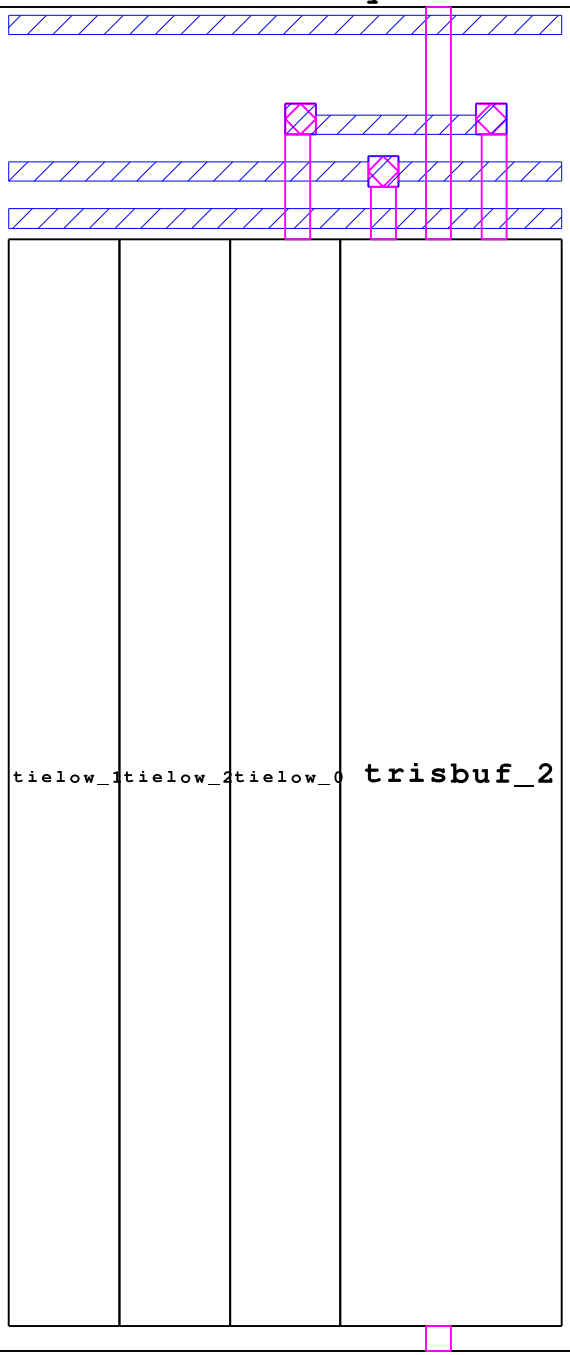
Plot of "flag_active.mag" for "dkl105" at Fri May 26 00:37:51 2006 Scale: 69 lambda/inch



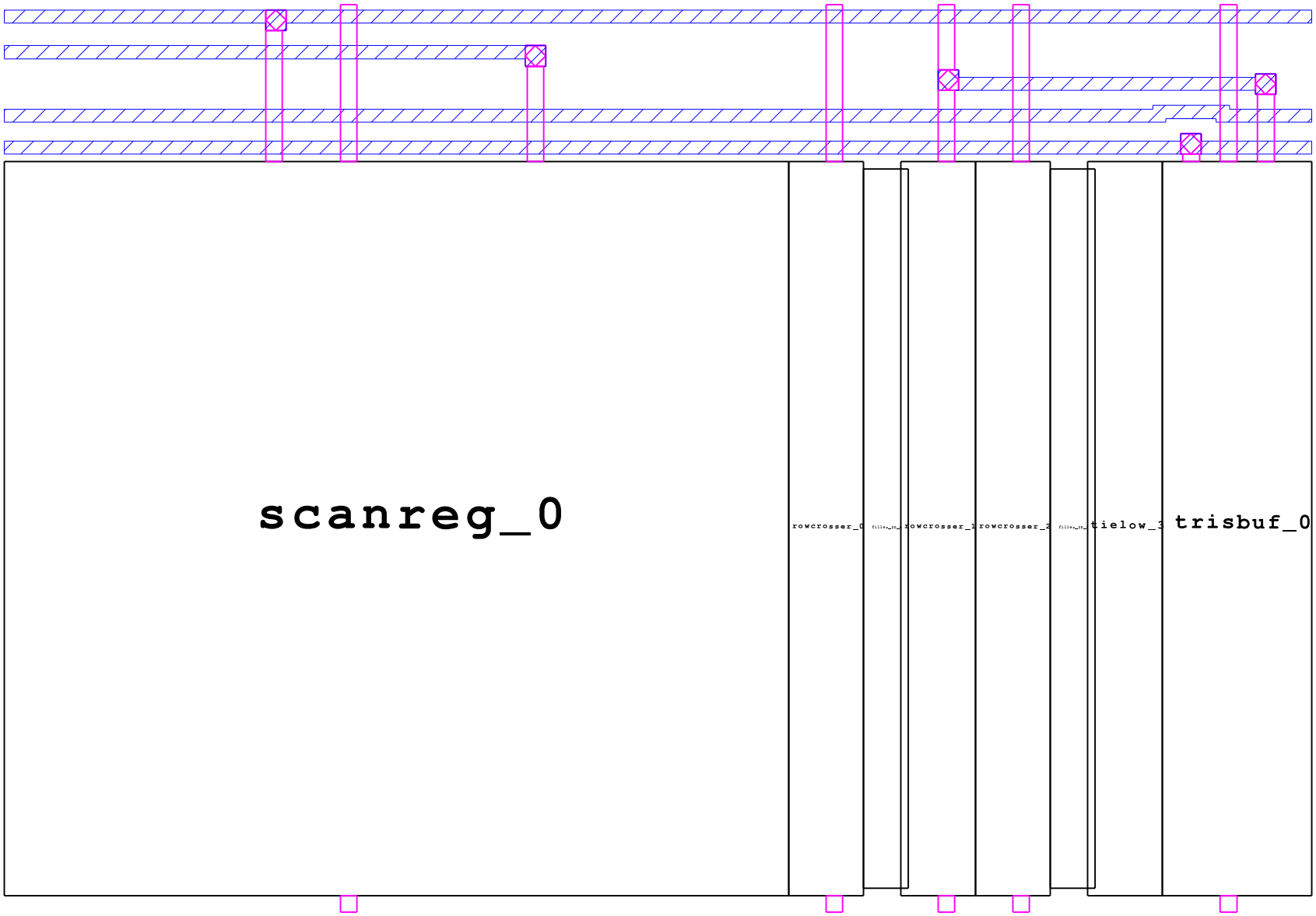
Plot of "flag_terminal.mag" for "dkl105" at Fri May 26 00:37:58 2006 Scale: 69 lambda/inch



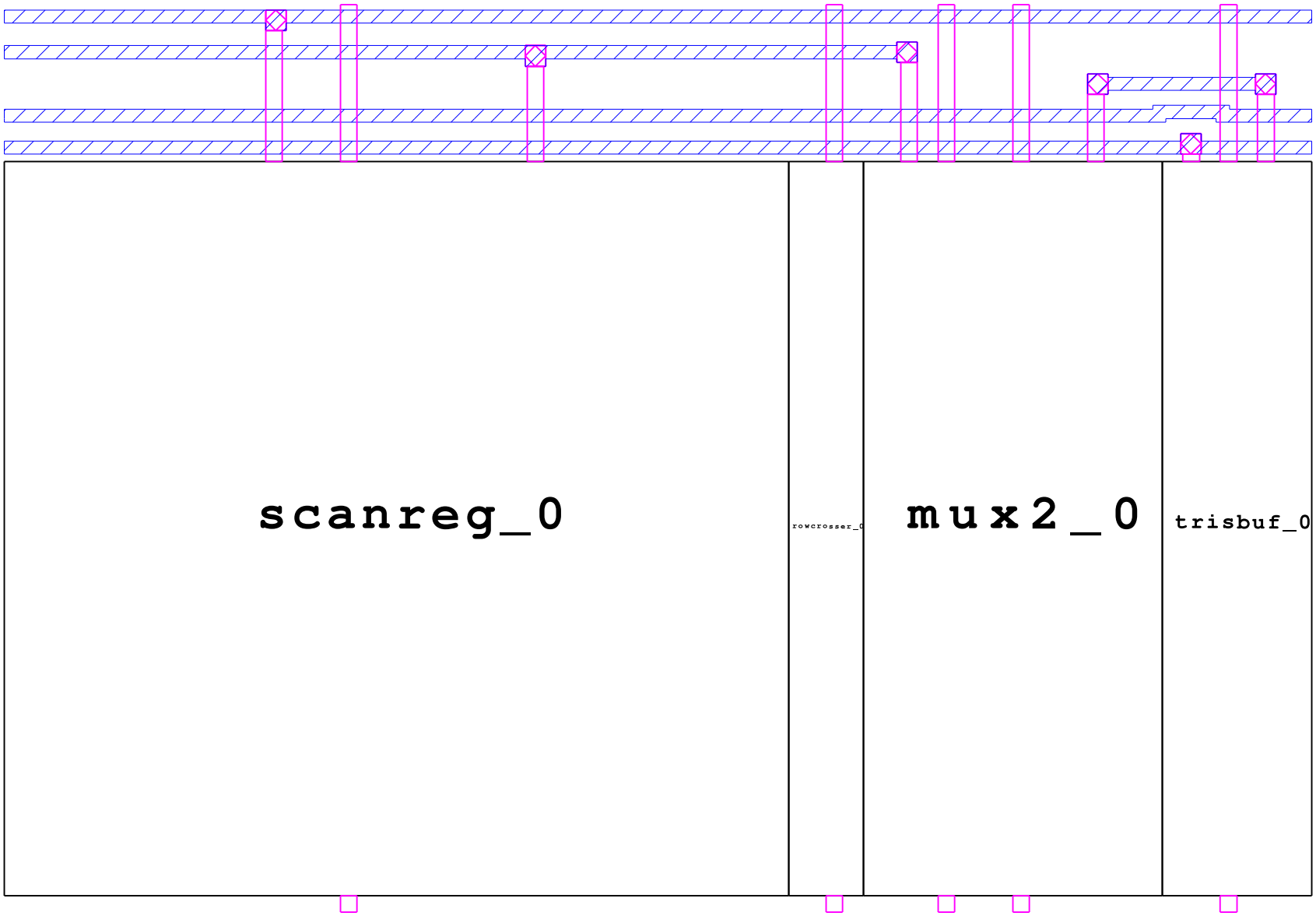
Plot of "flag_dummy.mag" for "dkl105" at Fri May 26 00:37:54 2006 Scale: 69 lambda/inch



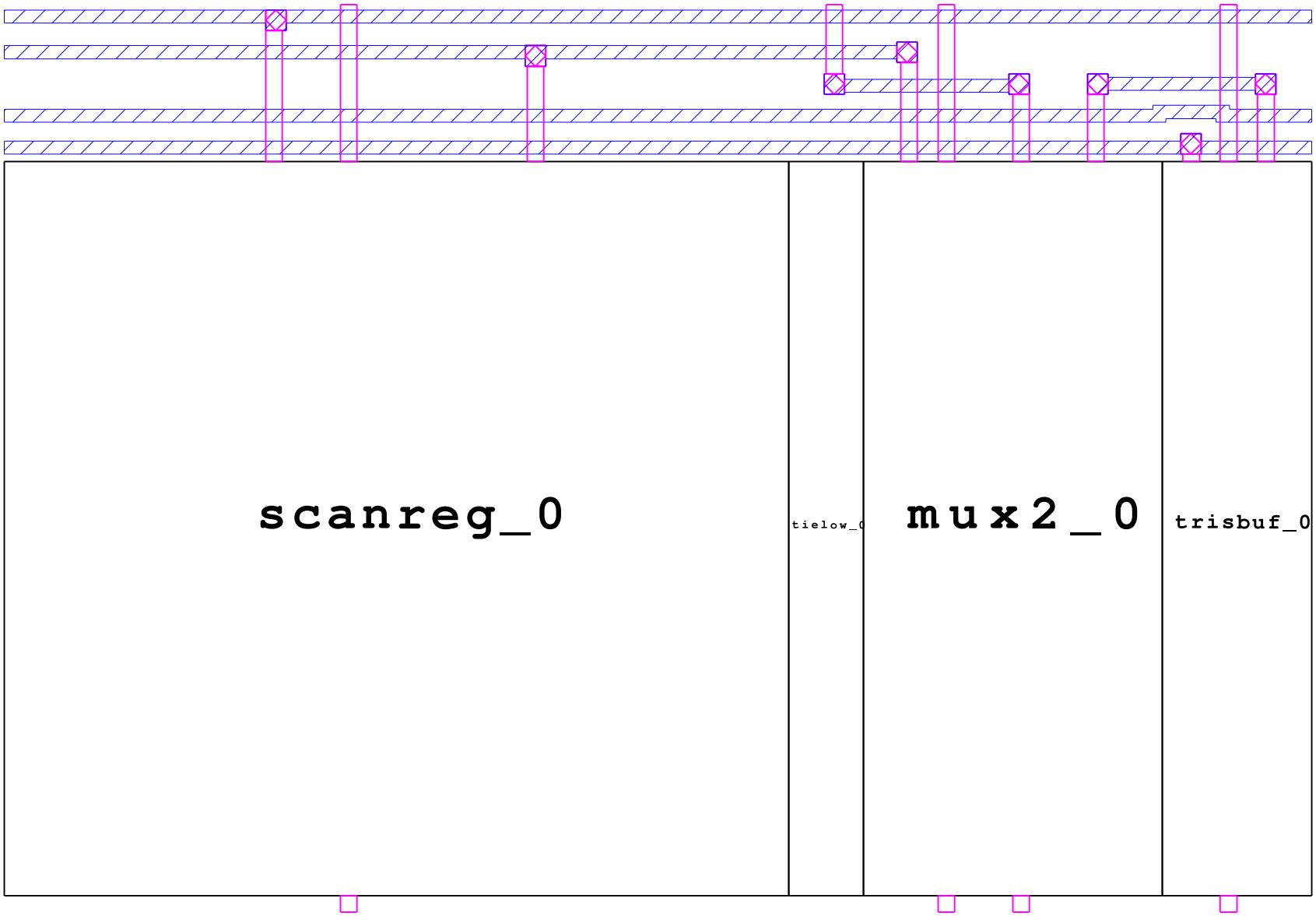
Plot of "ir_12_15.mag" for "dk1105" at Fri May 26 00:46:35 2006 Scale: 80 lambda/inch



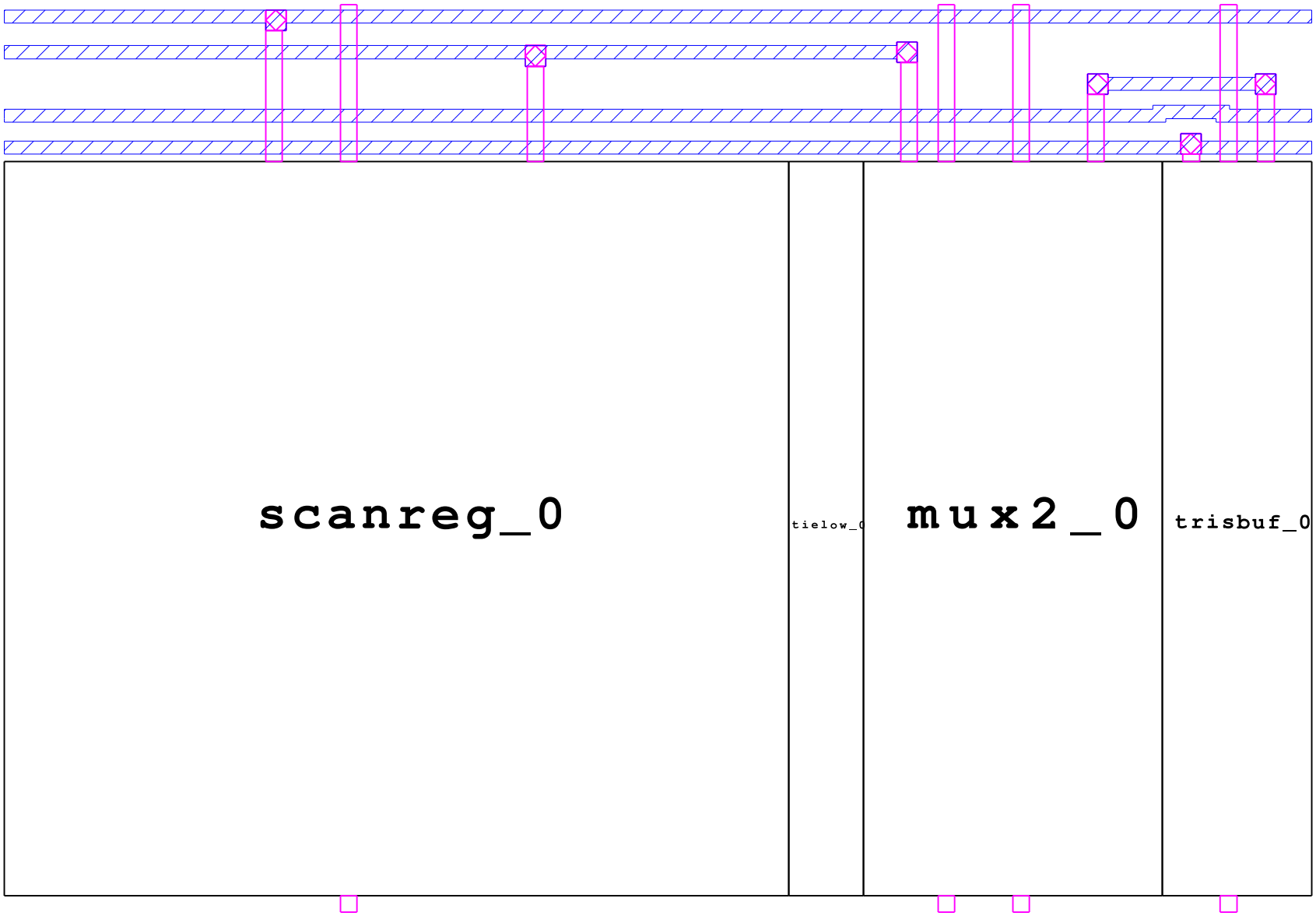
Plot of "ir_09_11.mag" for "dkl105" at Fri May 26 00:46:30 2006 Scale: 80 lambda/inch



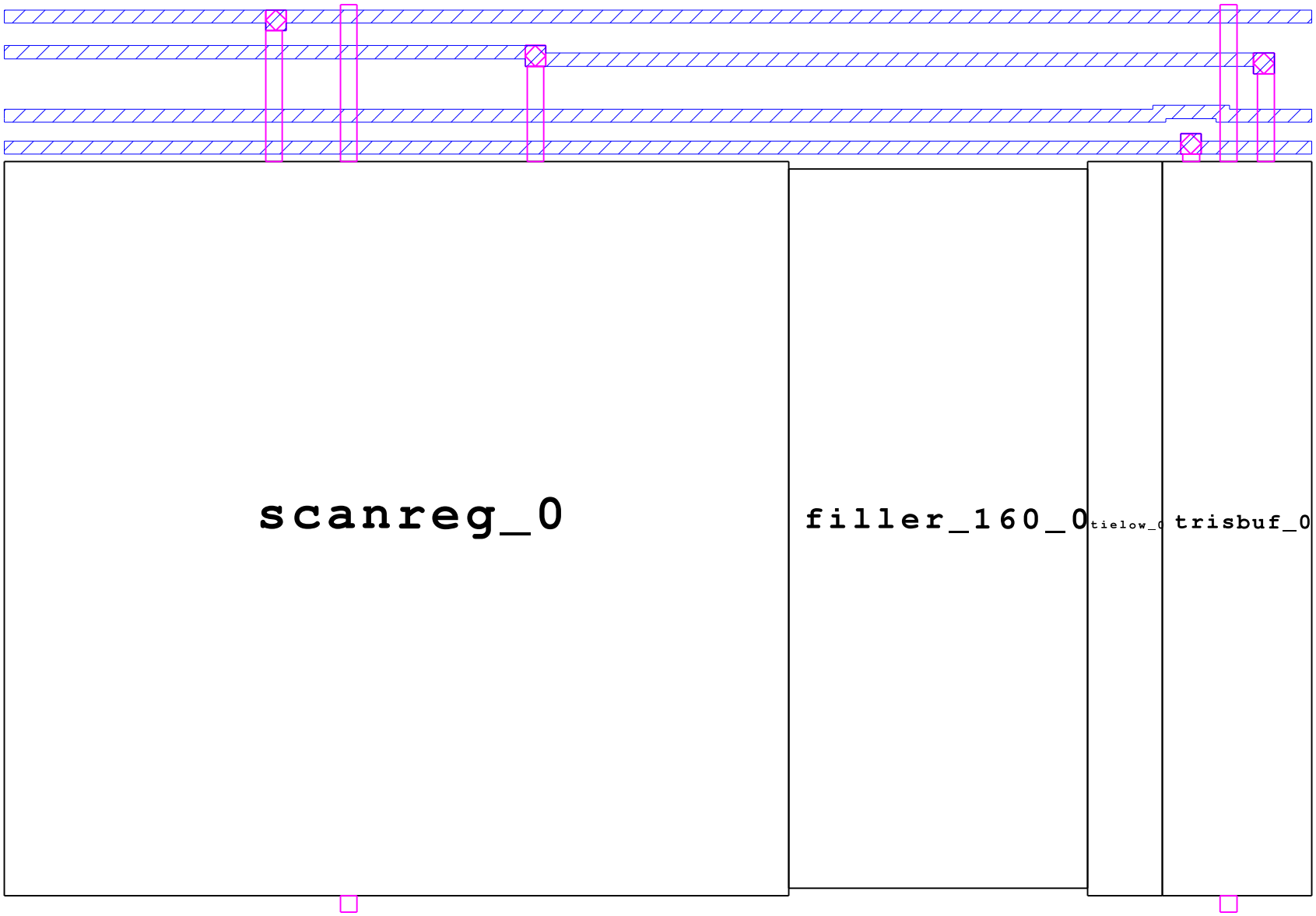
Plot of "ir_08_08.mag" for "dkl105" at Fri May 26 00:46:27 2006 Scale: 80 lambda/inch



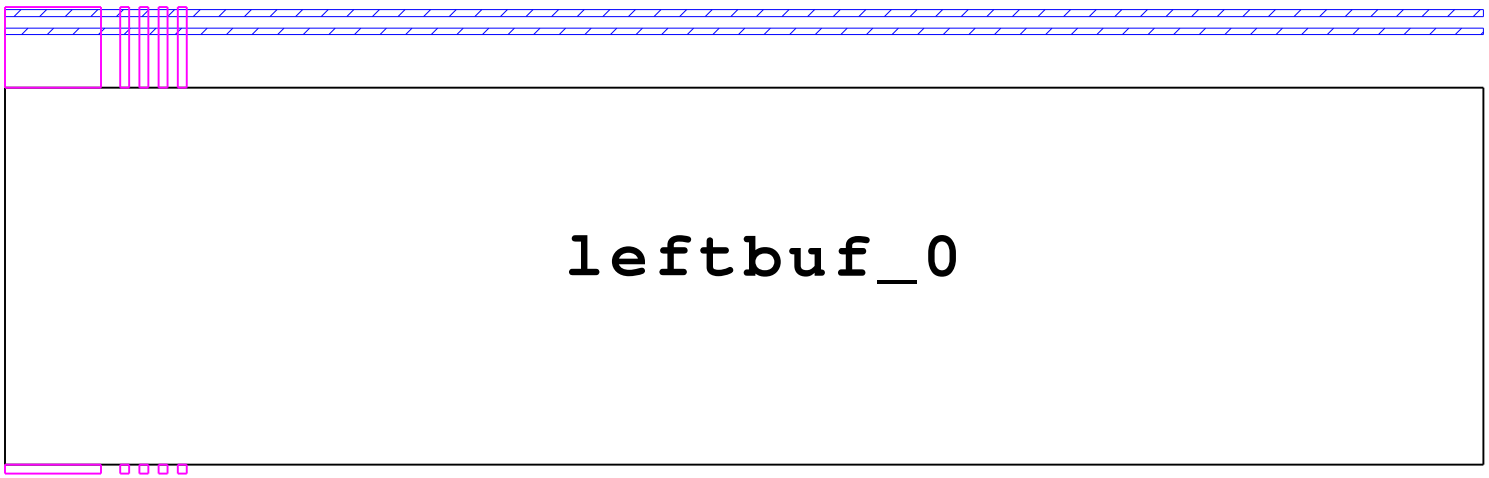
Plot of "ir_06_07.mag" for "dkl105" at Fri May 26 00:46:23 2006 Scale: 80 lambda/inch



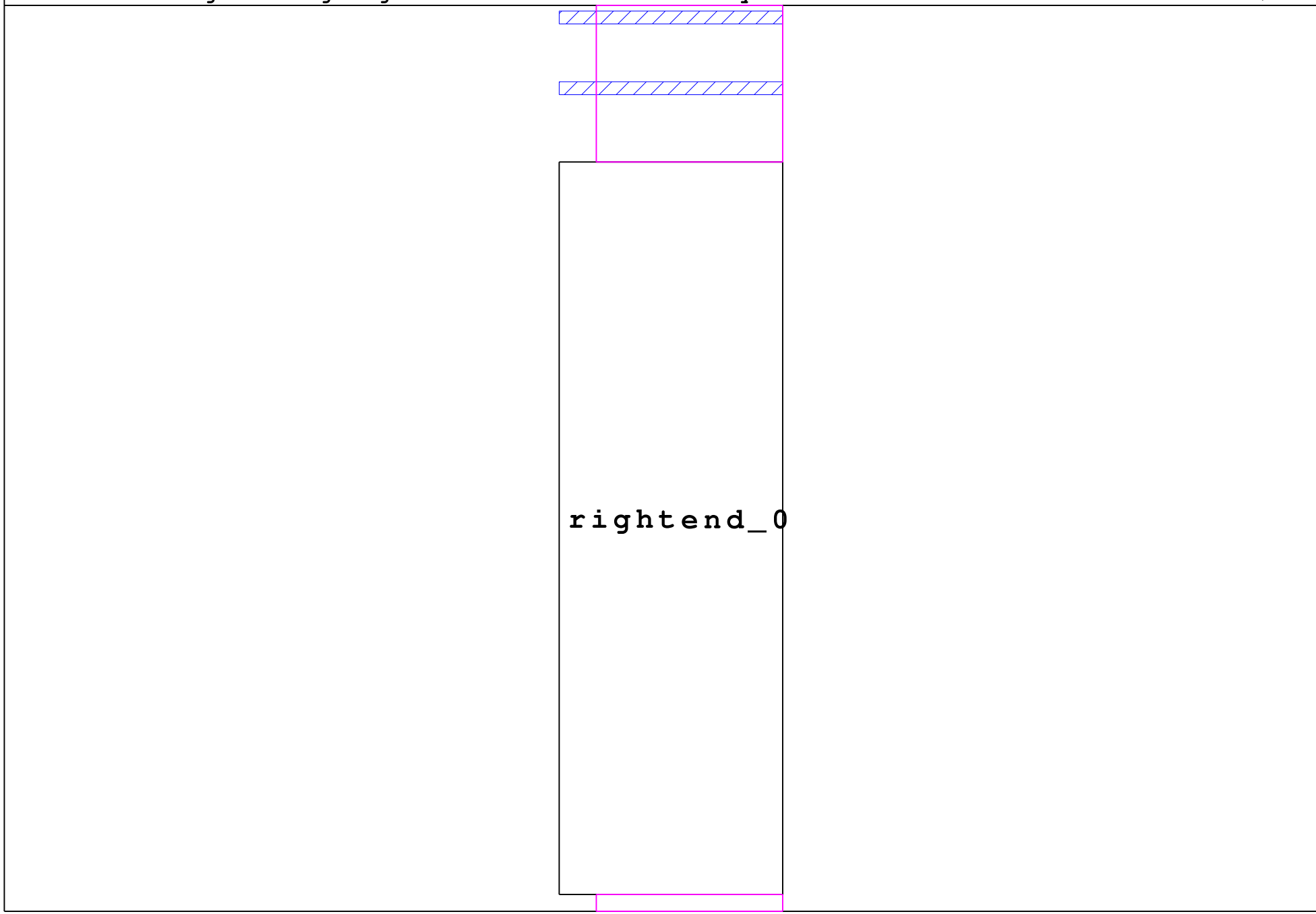
Plot of "ir_00_05.mag" for "dkl105" at Fri May 26 00:46:16 2006 Scale: 80 lambda/inch



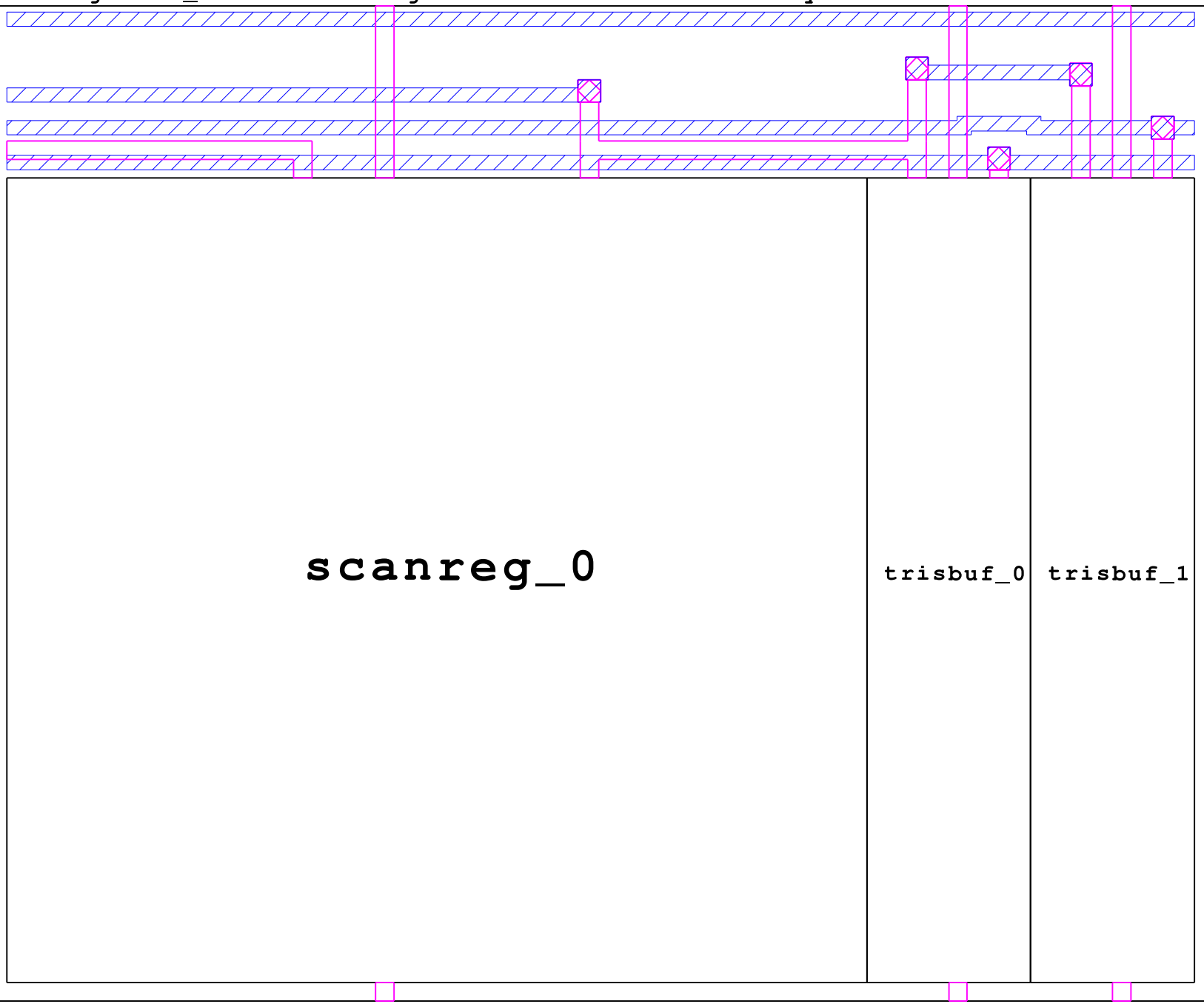
Plot of "leftending.mag" for "dkl105" at Fri May 26 00:46:45 2006 Scale: 200 lambda/inch



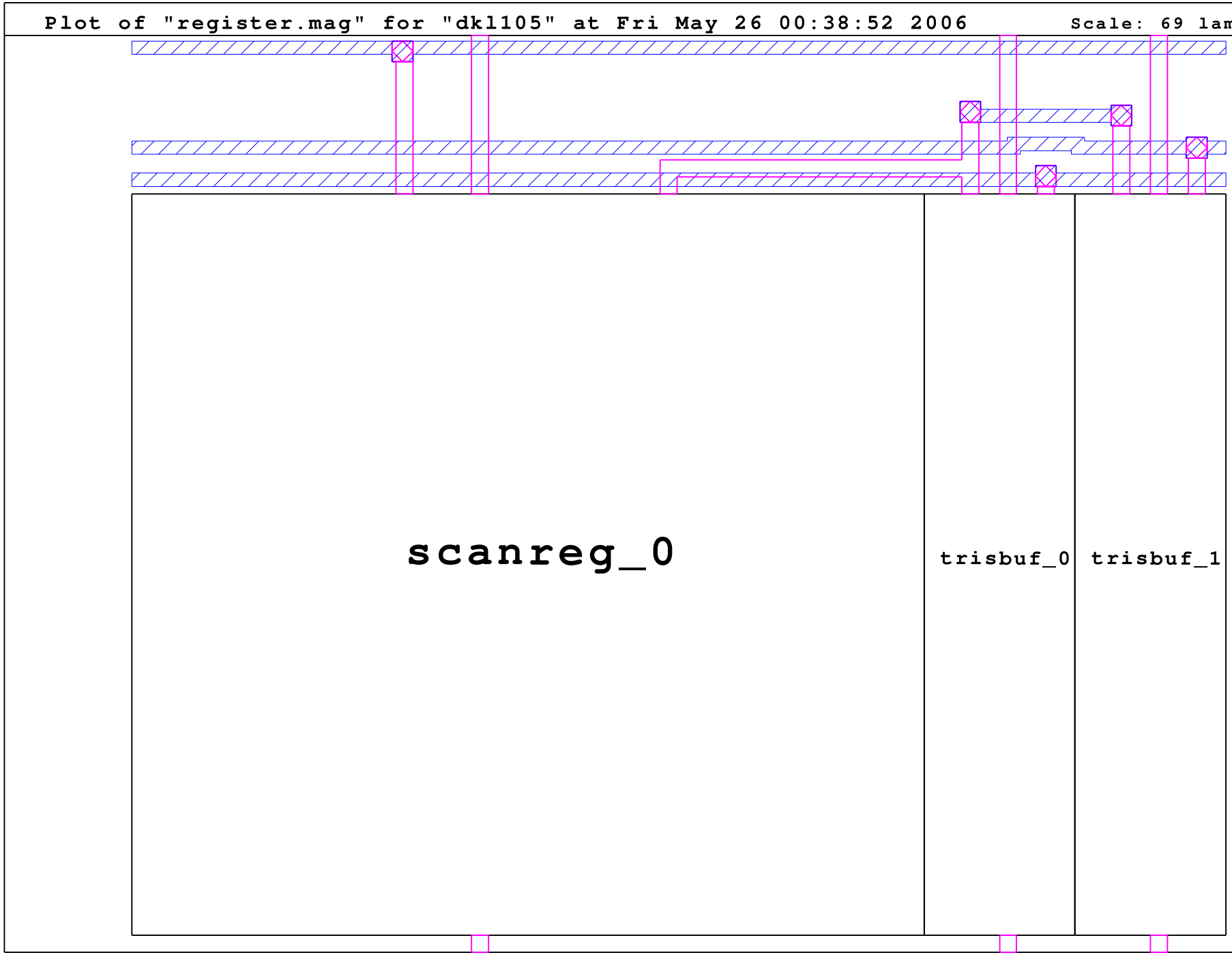
Plot of "rightending.mag" for "dkl105" at Fri May 26 00:39:08 2006 Scale: 69 lambda/inch



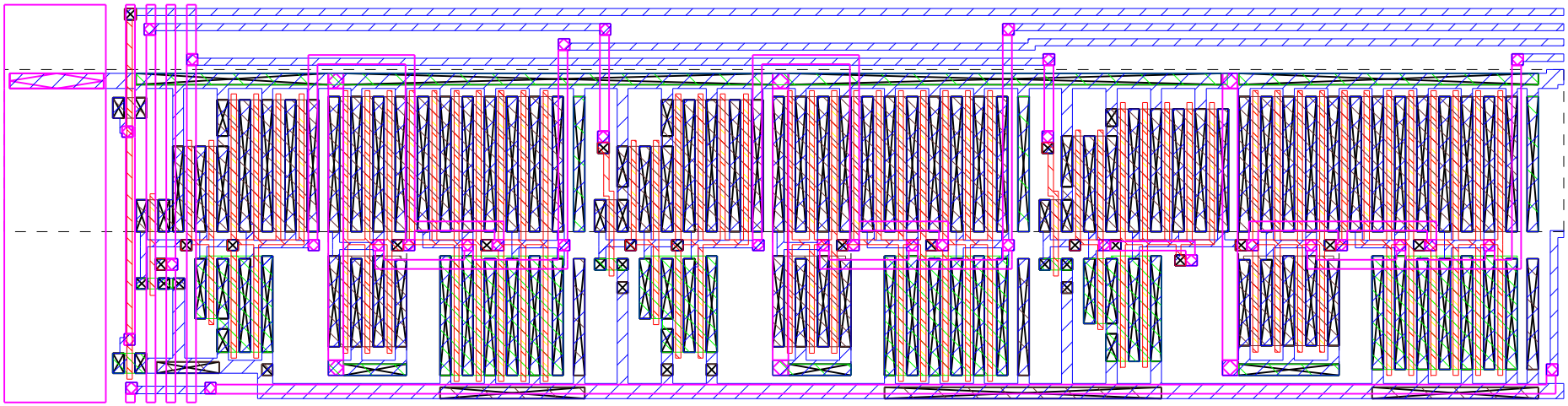
Plot of "register_extensible.mag" for "dkl105" at Fri May 26 00:38:55 2006 Scale: 69 lambda/inch



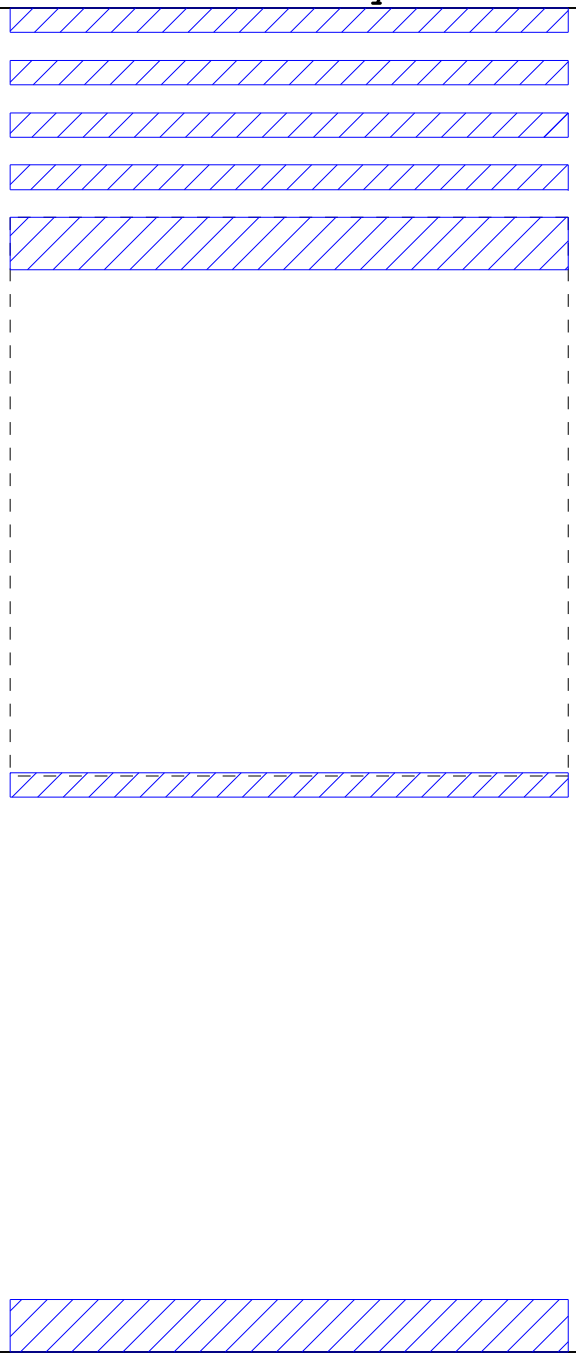
Plot of "register.mag" for "dk1105" at Fri May 26 00:38:52 2006 Scale: 69 lambda/inch



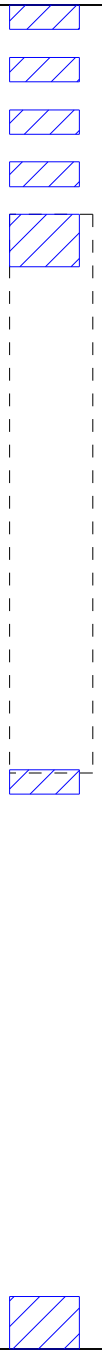
Plot of "leftbuf" for "dkl105" at Fri May 26 03:45:57 2006 Scale: 160 lambda/inch



Plot of "filler_160.mag" for "dkl105" at Fri May 26 00:37:44 2006 Scale: 55 lambda/inch



Plot of "filler_20.mag" for "dkl105" at Fri May 26 00:37:40 2006 Scale: 55 lambda/inch



## *Appendix E. Cadence DRC*

Here we describe the process of doing Cadence DRC to a magic design. We use the cpu.mag file as an example but any file can be tested this way.

Firstly we need to convert the magic design to cadence database format as cadence and magic use different file formats for design data storage. Hence it is necessary convert the magic file into GDS2 stream format. Following commands do the conversion of magic design to cadence database format:

% cd ~/design/fcde/magic/design

Translate the top level magic file to cadence format:

% do_cmos05_desin cpu.mag

Now we are going to enter the Cadence design environment. We move to the directory containing Cadence files:

% cd ~/design/fcde/cadence

Start cadence using,
% cds

Use the library manager for library, cell and cell view manipulation
Tools -> Library Manager...

Open the imported design for editing by selecting Library: fcde, Cell: cpu and View: layout then invoke the open command
File -> Open...
To perform DRC
Verify -> DRC...

DRC
Switch Names [ drc_shape? not_pads? float? ]
Rules File [ divaDRC.rul ]
Rules Library [ ] [ ]
OK

## *Appendix F. Running the simulations*

**Verilog-xl simulations**

To initiate the simulation on Verilog-xl simulator following commands are shown as examples;

- For behavioural model,

simulate -xl behavioural programs/multiply.hex 1500 +define+switch_value=2569

The ‘behavioural’ sub-directory contains the behavioural Verilog HDL model of our Microprocessor.

- For mixed model,

simulate –xl mixed programs/multiply.hex 200 +define+switch_value=2569

The ‘mixed’ sub-directory contains both of extracted netlists and behavioural models of our Microprocessor

- For structural model,

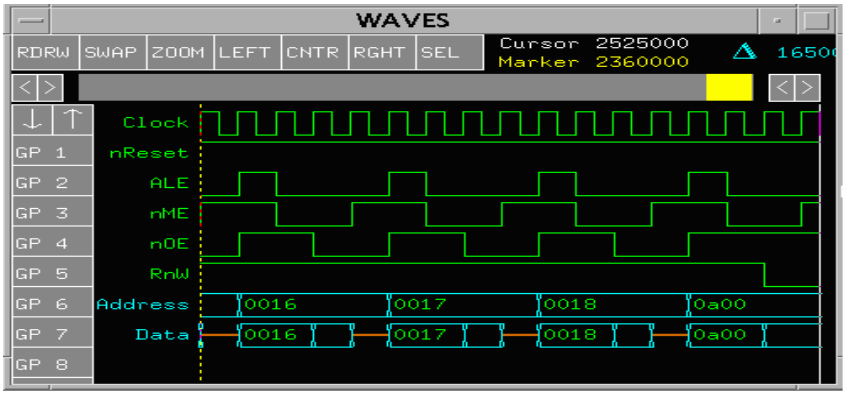
simulate –xl extracted programs/multiply.hex +define+switch_value=2569

The ‘extracted’ sub-directory contains the structural/gate level Verilog HDL model of our Microprocessor as extracted from the magic files.

On running the simulation using Verilog-xl simulator, both waveform window and Register window is available for the user. The register window shown in the following figure shows the contents of the all the registers used, control flags, switches (memory location 2048), LEDs (memory location 2560) along with other control signals. Value of the Switches is the data read by the program and the value stored in the LEDs is the result of the program being executed.


As seen in figure above, a register window also shows the content of memory location from 0 to 27 which contain the assembly program in hex. As the cursor on the waveform window is moved using the '<' and '>' buttons, the values within the register window changes to those values at that particular time.

The waveform window as in the figure bellow shows the content of the control signals, Address and data value. Hence with these features provided in the verilog-xl simulation environment, the user can have a reasonable convenient simulation GUI.




**NC-verilog simulation:**

To initiate the simulation on NC-Verilog simulator following commands are shown as examples;

- For behavioural model,

  simulate behavioural programs/multiply.hex 1500 +define+switch_value=2569

The 'behavioural' sub-directory contains the behavioural Verilog HDL model of our Microprocessor.

- For mixed model,

  simulate mixed programs/multiply.hex 200 +define+switch_value=2569

The 'mixed' sub-directory contains both of extracted netlists and behavioural models of our Microprocessor

- For structural model,

  simulate extracted programs/multiply.hex +define+switch_value=2569

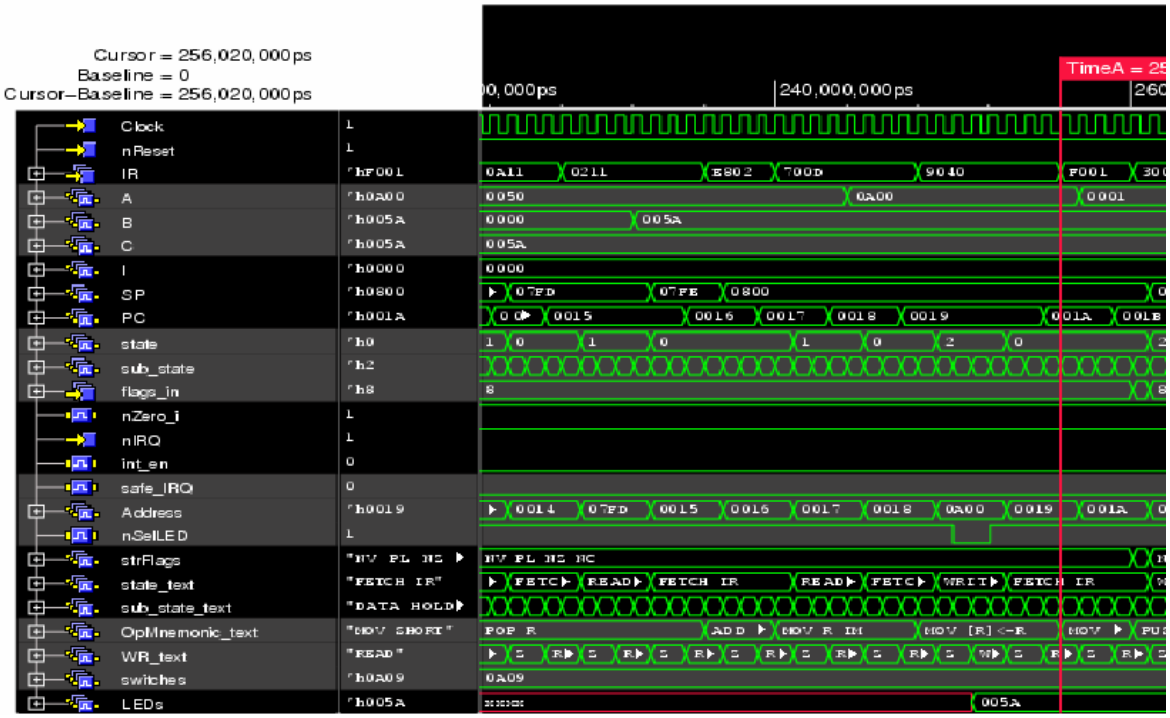
The 'extracted' sub-directory contains the structural/gate level Verilog HDL model of our Microprocessor as extracted from the magic files.

On running the simulation using NC-Verilog simulator, both waveform window and Register window is available for the user. The waveform window shown in the following figure shows the contents of the all the registers used, control flags, switches (memory location 2048), LEDs (memory location 2560) along with other control signals. The nice feature provided for the user convenience and better understanding is the inclusion of following signals in the wave form window:

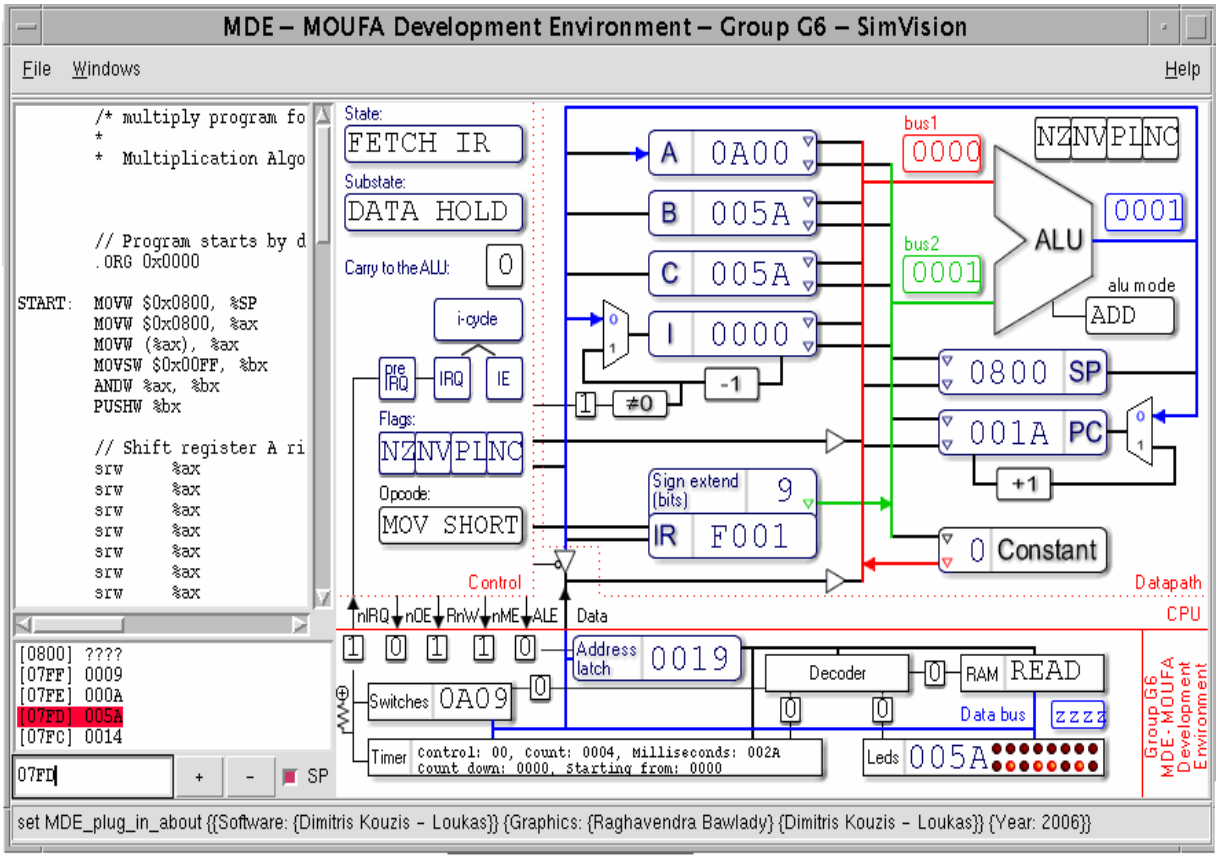
1. 'strFlags' which provides information on flag status after execution of each instruction.
2. 'state_text' which provides information on which state/cycle the processor is in, either Fetch cycle or Write cycle or Read cycle.
3. 'sub_state_text' which provides information on which sub-state the processor is in, either Address setup or Address hold or Data setup or Data hold.

4. ‘OpMnemonic_text’ which shows the opcode being executed that that particular time.
5. ‘prog_name’ showing the name of the program currently being executed.
6. ‘switches’ (memory location 2048) and ‘LEDs’ (memory location 2560), showing the data read and the result of the program respectively.

**Waveform 1 – SimVision**




Another salient feature provided for our processor simulation is the MDE plug-in window as shown in the following figure. It shows the contents of the all seven registers used, bus1 and bus2 values, state, substate and opcode information , signals (nIRQ, nOE, RnW, nME, ALE) to and out of the control block and memory content of location 2048 (Switches) and 2560 (LEDs). To the left of the register window the assembly program currently being executed is displayed. It has a red highlighter which shows the instruction being currently executed for the particular time where the cursor is positioned in the waveform window. As the cursor position is moved on the waveform window, the position of the red highlighter in the assembly program on the register window changes. The bottom left of the register window shows the contents of the memory location 0x0000 to 0xFFFF. It is provided with ‘+’ and ‘-‘buttons for the user to scroll up or down the order. If the user requires information on the stack, it is necessary to select the SP option provided near ‘+’ and ‘-‘buttons. The  buttons provided in the waveform window enables the user to perform step by step execution. With all these features provided, register window provides a friendly environment for the user to do the simulation and error checking.




**Applying test vectors to the cpu.**

Type the following instruction:

% ncverilog -y behavioural -y system -y core_test +libext+.v +incdir+behavioural core_test/core_test.v

You will see something like this.

Running X
Successful test vector 1.
Successful test vector 2.
….
Running 0
Successful test vector 1.
Successful test vector 2.
…
Running 1
Successful test vector 1.
Successful test vector 2.

All the test vectors are successful indicating that no error actually exists. If an error existed, state information about the system would be dumped to your terminal.

**Scanpath simulation.**

Type the following instruction:

% simulate -no_graphics extracted programs/multiply.hex +define+special_stimulus

It gives:

SCAN TEST: Found 125 dtypes/registers in scan path

Type exit to finish the simulation. Note that because components of the system (memories etc.) are connected during the scanpath test, some timing violations may occur.

**Extracted scanpath test vector application.**

Type the following instruction:

% ncverilog -y extracted -y system -y core_test +libext+.v +incdir+extracted core_test/extracted_test.v

You will see something similar to this:

Running 0
Successful test vector 1.
Successful test vector 2.
….
Running 1
Successful test vector 1.
Successful test vector 2.

All the test vectors are successful indicating that no error actually exists. If an error existed, state information about the system would be dumped to your terminal. This simulation is much slower than the behavioral one because the state has to be set-up via the scanpath and the model used is transistor level model.

## *Appendix G. Scanpath structure*

The scanpath has the following structure:

| | | | |
|---|---|---|---|
| SDI → | | | |
| (IR → A → B → C → SP → PC → I) [16] | | | |
| (IR → A → B → C → SP → PC → I) [15] … | | | |
| | | | |
| (IR → A → B → C → SP → PC → I) [0] | | | |
| Controller's row | Register's # | Variable name | Net name |
| f) | 12 | flags_reg_3 | cpu_core_0/datapath_0/flags_in<3> |
| f) | 13 | flags_reg_1 | cpu_core_0/datapath_0/flags_in<1> |
| d) | 11 | flags_reg_2 | cpu_core_0/datapath_0/flags_in<2> |
| b) | 10 | flags_reg_0 | cpu_core_0/control_0/mux2_4/I0 |

| | | | |
|---|---|---|---|
| a) | 1 | irq_tmp_reg | cpu_core_0/control_0/scandtype_8/Q |
| a) | 2 | safe_IRQ_reg | cpu_core_0/control_0/nand3_2/A |
| a) | 3 | int_en_reg | cpu_core_0/control_0/nand2_14/B |
| a) | 4 | extended_cycle_reg | cpu_core_0/control_0/nor3_1/C |
| a) | 5 | sub_state_reg_0 | cpu_core_0/control_0/nor2_1/B |
| a) | 6 | sub_state_reg_1 | cpu_core_0/control_0/inv_8/A |
| a) | 7 | state_reg_1 | cpu_core_0/control_0/inv_5/A |
| a) | 8 | i_cycle_reg | cpu_core_0/control_0/inv_2/A |
| a) | 9 | state_reg_0 | cpu_core_0/control_0/nor2_18/B |
| → SDO | | | |
| Notes: Controller's row is numbered from top to the bottom<br>Register's number is numbered from top left to bottom right | | | |

## *Appendix H. Final report contributions*

[removed]

## *Appendix I. TCL/TK development*

A nice tool to start with prototyping TCL/TK applications is "wish". Its main benefit is that it loads very quickly allowing rapid development of the GUI. When the GUI gets finalized, you can integrate it to the SimVision environment by using the plugin::Window class.

Great help on polishing the plug-in was provided by the excelent book "Effective Tcl/Tk Programming" from Mark Harrison and Michael McLennan (the Architect at Cadence who developed SimVision visualization and debugging environment for NC-Sim). It's written with a way that allows you to quickly find what you are looking for and provides a lot of complete examples for every topic.

http://users.belgacom.net/bruno.champagne/tcl.html (Quick start tutorial)
http://www.tcl.tk/man/tcl/ (The reference manual for TCL/TK)
http://wiki.tcl.tk/ (Wiki with many solutions to common problems)
http://hegel.ittc.ku.edu/topics/tcltk/tutorial-noplugin/ (Quick tutorial)
http://www.eso.org/projects/vlt/sw-dev/tcl8.3.3/itcl3.2.1/contents.html (Documentation for itcl, the object oriented extension of TCL).

The most important resource for writing TCL plug-ins for SimVision is of course the "SimVision Command Language Reference" which is quite well written. Chapter 3 "Writing Plug-In Applications" should be the starting point for each such development effort.

Finally a valuable resource will be the source code of MDE that can be found in any simulation directory (e.g. verilog/behavioural). It's named coreview.tcl and it's about 500 lines long.

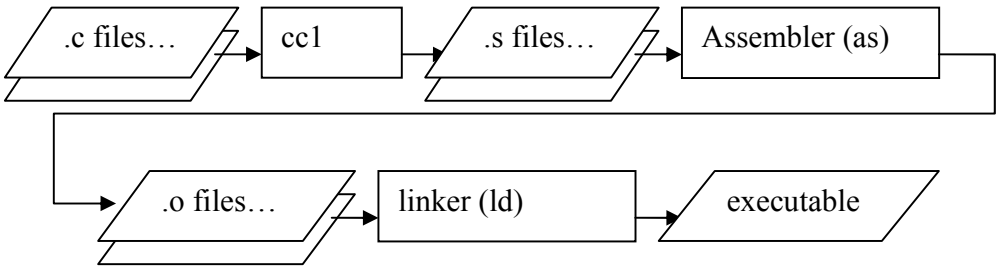
## *Appendix J. Porting the gcc compiler*

No modern processor is allowed to be released to the market nowadays without a C compiler. From a very early stage there was the willingness to port the gcc compiler for the MOUFA processor.

A lot of work (about two weeks) has been devoted to this task but it hasn't yet been able to deliver something useful. Porting the gcc to a new processor is not supposed to be an easy task and it really isn't. The main problem is the lack of a quick-start type documentation[5,6,7]. This holds true because poring compilers is not and is not expected to become a mainstream activity.

At the moment of writing we may be one day away from a working C compiler or we may be two weeks away from it. The pressing deadlines of other tasks of this project halted the gcc porting at the moment that it was starting to get interesting.

gcc is actually a set of tools as shown in the following picture. All but cc1 are in a separate package called binutils. The actual C compiler is just cc1.




Porting binutils is not so interesting. Porting the compiler is the most interesting part because there you can see processor abstraction models that are accurate enough to allow the optimization of code. It also gives you an insight on a tool that you use very often to create software; the C compiler. The modified data flow for the MOUFA processor would be this:


Unfortunately it operates in only one file but this file can be… arbitrary long!

In order to port the gcc, you only need to modify a few files. The most important are two, the processor.md and the processor.h. The first one has a large list of lisp-like "insn" patterns. The compiler parses the .c file and creates a list of "insn" nodes that represent the whole program. Then it goes through the list defined in the processor.md and tries to match each node. If this attempt is not successful it applies transformations to the original nodes and the tries again.

For example the statement i+=2 would translate to an "insn" node of the form reg = reg + immediate. Imagine that the target architecture doesn't have an instruction of this form. As a result searching in the processor.md's patterns fails. Then gcc will transform that to reg=reg+1 x2 and will try again. Assuming that the processor has the increase instruction, the pattern will match and two increase instructions will be included in the assembly file. Otherwise (and if the compiler can't thing of any more tricks) the compilation will fail either with a friendly message like "This processor doesn't support additions" if the one who did the porting took care of these cases or with a core dump (in which case the debugging is really difficult).

[5] http://gcc.gnu.org/onlinedocs/gcc-2.95.3/gcc.html
[6] Even NASA engineers admit it: http://compilers.iecc.com/comparch/article/96-04-132
[7] A useful introductory resource was the MSc thesis "Porting GCC for Dunces" of Hans-Peter Nilsson, May 21, 2000

processor.h has some macros inside and helps binding with the processor.c file that is usually needed. These macros are important because they define the number and classes of registers of the processor, the way the stack works and all the other parameters that are needed for compilation.

The current status of development is this. After having successfully modified the .md file of AVR microprocessors to deliver something that looked very similar to a MOUFA assembly listing, I modified the .h file to change the number and classes of registers. Of course for many reasons that I expected** lots of errors appeared (these macros are being referenced by many gcc's files). Many of these errors were fairly easily to repair. At this phase development stopped.

** Unfortunately in such big projects as the gcc you can't easily change just one parameter of the system and continue debugging. Often you have to change a lot of things to get back to a stable state. Very rarely so many changes can be done without mistakes or without forgetting to change something more. For example, in our case, removing a class of a registers requires modifications in all the instructions and macros that use them.

What I expect is that some more relatively easy to fix compile-time errors exist. After that a thorough re-viewing of the macros is required to ensure that everything is correct and there is nothing necessary that is missing. A review of the .md file would be beneficial as well. All these could be completed within a day.

Then I would attempt a first compilation of some easy programs including some for loops and arithmetic operations. It is almost sure that the resulting assembly files will have enough errors to keep me busy for another half a day. After that a compiler could be said to exist.

Although the compiler might be working I would allow a lot of testing until I start trusting it. Verifying compilers is a very difficult task because of their complexity. "One thing that helps is having a test suite; I'm using a C compiler validation suite from Metaware ($2k)" [8]. The fact that our compiler was based on another configuration file that has been already been tested for years in the market, decreases the probabilities of large errors. An easy and cheap "test suite" would be trying to compile GNU functions starting from putc up to printf. If printf's code compiled successfully, I would be quite confident that the compiler could be used for many useful applications.
I will probably try to complete this task at some moment because I believe that by being able to support a processor with a C compiler, you make it able to be practically useful in the market. Now that you can prototype a powerful multi-processor (in your garage) with an FPGA one of the few limiting factors on creating a useful product is its software support.

[8] http://compilers.iecc.com/comparch/article/96-04-132

## *Appendix K. ALU model's details*

An Arithmetic Logic Unit (ALU) is a combinational circuit in the microprocessor that performs all arithmetic and logical computations. The ALU is designed to perform the following functions:

1. Addition
2. Subtraction (using 2's Complement)
3. And
4. Or
5. Xor
6. Shift/Rotate Right with carry
7. Not

The operations performed by the ALU are controlled by the input 'ALUmode', which governs the type of operation to be performed by the ALU.

| | **ALUmode** | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Function** | [6] NegEn | [5] SubEn | [4] AritEn | [3] XorEn | [2] AndEn | [1] OrEn | [0] SrEn | **Operations** |
| FN_SRC | 0 | X | 0 | X | 0 | 0 | 1 | in1 >> 1 (with carry) |
| FN_OR | 0 | X | 0 | X | 0 | 1 | 0 | in1 \| in2 |
| FN_AND | 0 | X | 0 | X | 1 | 0 | 0 | in1 & in2 |
| FN_ADD | 0 | 0 | 1 | 0 | 0 | 0 | 0 | in1 + in2 |
| FN_SUB | 0 | 1 | 1 | 0 | 0 | 0 | 0 | in1- in2 (2's complement) |
| FN_PASS | 1 | 0 | 0 | 0 | 0 | 0 | 0 | in1 (Pass) |
| FN_XOR | 1 | X | 0 | 1 | 0 | 0 | 0 | in1^ in2 |
| FN_NOT | 1 | 1 | 0 | 0 | 0 | 0 | 0 | ~ in1 |

**Implementation of the Functions**:

**Addition:** The full adder cell from the cell library is used to implement addition function. To perform the addition operation, the 'XorEn' is set to low and the 'AritEn' is set to high, resulting in input1 (in1) and input2 (in2) to appear at the inputs of the fulladder and their resulting sum to appear at the Result bus.

**Subtraction:** The subtraction function is implemented using 2's complement method. It is done by using a xor gate at the input 'A', of the fulladder. The select input of the multiplexer 'XorEn' and signal 'SubEn' are set to low and high respectively, resulting in 1's complement of input2 (in2) to appear at the input 'A' of the fulladder. The 'Cin' input of the fulladder is set to high through flags_in<0>; hence enabling to perform subtraction using 2's complement method. The 'XorEn' control signal selects input2 (in2) during subtract operation. All the necessary control logic is implemented in the Control Module.

**And:** Two input AND gate from the cell library has been used to implement the logical AND operation. The control signal 'AndEn' is set to high to enable the result of the AND operation to appear at the Result bus.

**Or:** Two input OR gate from the cell library has been used to implement the logical OR operation. The control signal 'OrEn' is set to high to enable the result of the OR operation to appear at the Result bus.

**Xor**: Two input XOR gate from the cell library has been used to implement the logical XOR operation. The control signal 'XorEn' (select input of the multiplexer) is set to high to select the input2 (in2). Also the control signal 'NegEn' is set high to enable the result of the XOR operation to appear at the Result bus.

**Not:** Two input XOR gate from the cell library has been used to implement the NOT operation by passing '1' and input1 (in1) to the two inputs of the XOR gate. The control signal 'SubEn' and 'XorEn' are set to high and low respectively to enable the logic '1' to appear at one of the inputs of the XOR gate. The control signal 'NegEn' is set to high to enable the result of the XOR operation (1's complement) to appear at the Result bus.

**Shift Right with carry:** Two input XOR gate from the cell library has been used to implement the NOT operation by passing '1' and input1 (in1) to the two inputs of the XOR gate. The control signal 'SubEn' and 'XorEn' are set to high and low respectively to enable the logic '1' to appear at one of the inputs of the XOR gate. The control signal 'NegEn' is set to high to enable the result of the XOR operation (1's complement) to appear at the Result bus.

In addition to the above functions, the ALU also generates the Arithmetic Carry, Shift right Carry, Arithmetic last carry flag, Shift last carry flag, Zero, and Negative flags represented as Cout, SrC, Cpre, SrLC, nZout and Nout respectively. These flags are updated only when that particular instruction/operation is executed. These flags are transferred to the control logic to update the nZout, Overflow, Nout and Cout flags (flags_in (3:0)) generated by control logic. Figure 2.3 illustrates how the ALU flags have been implemented in order to be bit sliced along with the implementation of above functions.

**Flags**:

**Arithmetic Carry Flag (Cout):** The Cout of fulladder in the (N-1)th bitslice is connected to the Cin of fulladder in the Nth Bitslice. The Cout of the 16th bitslice is the value which is to be updated in the Arithmetic Carry flag. The Cin of the fulladder in the $0^{th}$ bitslice comes from the control unit.

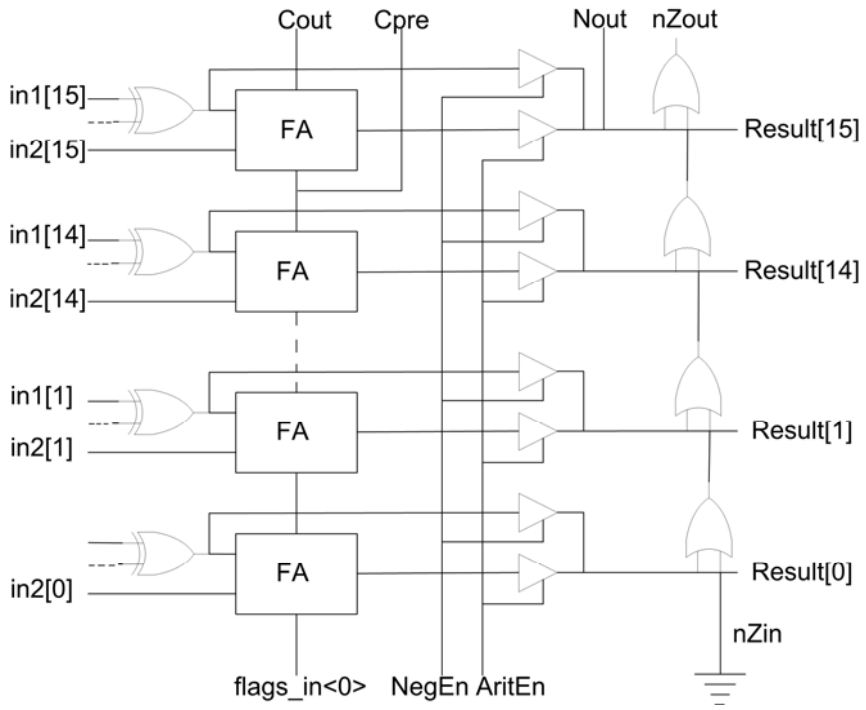
**Arithmetic Last Carry Flag (Cpre):** The Cpre flag is updated by the Cout of the fulladder in the $14^{th}$ bitslice. This flags is used by control block to update the overflow flag by performing OR operations with Cout flag for all the ALU operations other than shift right.

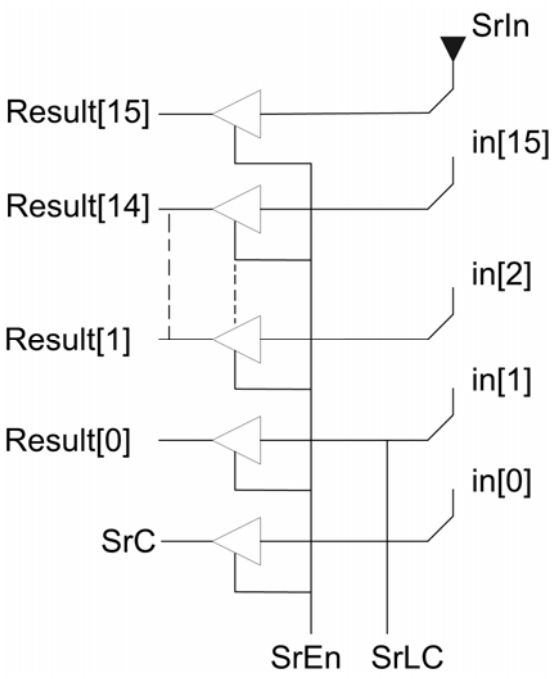
**Negative Flag (Nout):** The Negative flag is set to high when the result is negative i.e. when the most significant bit of the result is one and is set to low when the result is non negative i.e. when the most significant bit of the result is zero. Hence this flag is data from the MSB of the 16 bit result of the ALU.

**Zero Flag (nZout):** The Zero Flag is set to high only when the result is non-zero; else the flag has a value of zero.

**Shift right Carry Flag (SrC):** The SrC flag is set to high when the data out of the shift register generated during shift right operation is bit '1', else the flag has a value of zero. Hence this flag is the LSB of the 16-bit operand to be shifted right.

**Shift right Last Carry Flag (SrLC):** The SrC flag is set to high when the $0^{th}$ bit of the 16-bit result generated during shift right operation is bit '1'; else the flag has a value of zero. Hence this flag is the $1^{st}$ bit of the 16-bit operand to be shifted right. This flags is used by control block to update the overflow flag by performing OR operation with SrC flag for the shift right operation. The hardware implementation of the flags can be seen in the figure bellow.






## *Appendix L. Synthesis utilities*

**count_instances**

```
#!/bin/bash
cat $1 | egrep
"(and2|fulladder|halfadder|inv|mux2|nand2|nand3|nand4|nor2|nor3|or2|scandtype|scanreg|trisbuf|xor2)[^(]*\(" | wc -l
```

**synth**

```
#!/bin/bash
synth_custom -batch control.v control_synth.v control_synth.edif
count_instances control_synth.v
```

**backup**

```
#!/bin/bash
dir=`date +%y_%m_%d_%H_%M_`$1
mkdir $dir
cp control.v opcodes.v control_synth.edif control_synth.v $dir
```

## *Appendix M. Behavioral refinement data*

The following two tables are the low level specifications of the operation of MOUFA processor with two different perspectives. The “MOUFA control logic Karnaugh maps” presents the functionality in a control-signal-centric manner suitable for control logic creation while the “Overview of operations’ functionality” is more instruction-centric suitable for testing. The yellow boxes indicate values that differentiate from the defaults. The light gray values are don’t cares that have been set in a manner that optimizes the area.

An overview of the operation's functionality

| | Fetch | | | | Write | | | | Read | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | asetup | ahold | dsetup | dhold | asetup | ahold | dsetup | dhold | asetup | ahold | dsetup | dhold | e-cycle |
| PUSH reg | DB←PC | DB←PC+ | IR←DB | INIT_W,SP-- | DB←SP | DB←SP | DB←OP1 | DB←OP1,cINT | | | | | |
| PUSH status | DB←PC | DB←PC+ | IR←DB | INIT_W,SP-- | DB←SP | DB←SP | DB←STA | DB←STA,cINT | | | | | |
| CALL short_offset | DB←PC | DB←PC+ | IR←DB | INIT_W,SP-- | DB←SP | DB←SP | DB←PC | DB←PC,INIT_E | | | | | PC←PC+SE12 |
| CALL reg | DB←PC | DB←PC+ | IR←DB | INIT_W,SP-- | DB←SP | DB←SP | DB←PC | DB←PC,INIT_E | | | | | PC←OP1 |
| CALL offset | DB←PC | DB←PC+ | IR←DB | INIT_W,SP-- | DB←SP | DB←SP | DB←PC+ | DB←PC+1,INIT_ | DB←PC | DB←PC+ | PC←DB | cINT | |
| MOV {reg} – reg, short_offset | DB←PC | DB←PC+ | IR←DB | INIT_W | DB←OP2 | DB←OP2 | DB←OP1 | DB←OP1,cINT | | | | | |
| JIFAR [Z,C,OF,?], ne, offset | DB←PC | DB←PC+ | IR←DB | INIT_R OR PC++, cINT | | | | | DB←PC | DB←PC+ | PC←DB | cINT | |
| MOV reg-{reg}, short_offset | DB←PC | DB←PC+ | IR←DB | INIT_R | | | | | DB←OP1 | DB←OP1 | OP2←DB | cINT | |
| ARITH_REG_IMM | DB←PC | DB←PC+ | IR←DB | INIT_R | | | | | DB←PC | DB←PC+ | DO_OP | cINT | |
| POP reg | DB←PC | DB←PC+ | IR←DB | INIT_R | | | | | DB←SP | DB←SP | OP2←DB | SP++,IE?,cINT | |
| POP status | DB←PC | DB←PC+ | IR←DB | INIT_R | | | | | DB←SP | DB←SP | STA←DB | SP++,cINT | |
| ADD reg – short_immediate | DB←PC | DB←PC+ | IR←DB | DO_OP,cINT | | | | | | | | | |
| MOV reg – short_immediate | DB←PC | DB←PC+ | IR←DB | DO_OP,cINT | | | | | | | | | |
| JIF [Z,C,OF,?], ne, short_offset | DB←PC | DB←PC+ | IR←DB | DO_OP,cINT | | | | | | | | | |
| DECIJNQ short_offset | DB←PC | DB←PC+ | IR←DB | DO_OP,cINT | | | | | | | | | |
| ARITH_REG_REG | DB←PC | DB←PC+ | IR←DB | DO_OP,cINT | | | | | | | | | |
| STC | DB←PC | DB←PC+ | IR←DB | DO_OP,cINT | | | | | | | | | |
| STIE | DB←PC | DB←PC+ | IR←DB | DO_OP,cINT | | | | | | | | | |
| LDIE | DB←PC | DB←PC+ | IR←DB | DO_OP,cINT | | | | | | | | | |
| NOP | DB←PC | DB←PC+ | IR←DB | DO_OP,cINT | | | | | | | | | |
| i-cycle | | | | | DB←SP-1 | SP(DB)← | DB←PC | DB←PC, INIT_E | | | | | PC=2,DI,endI,endE |

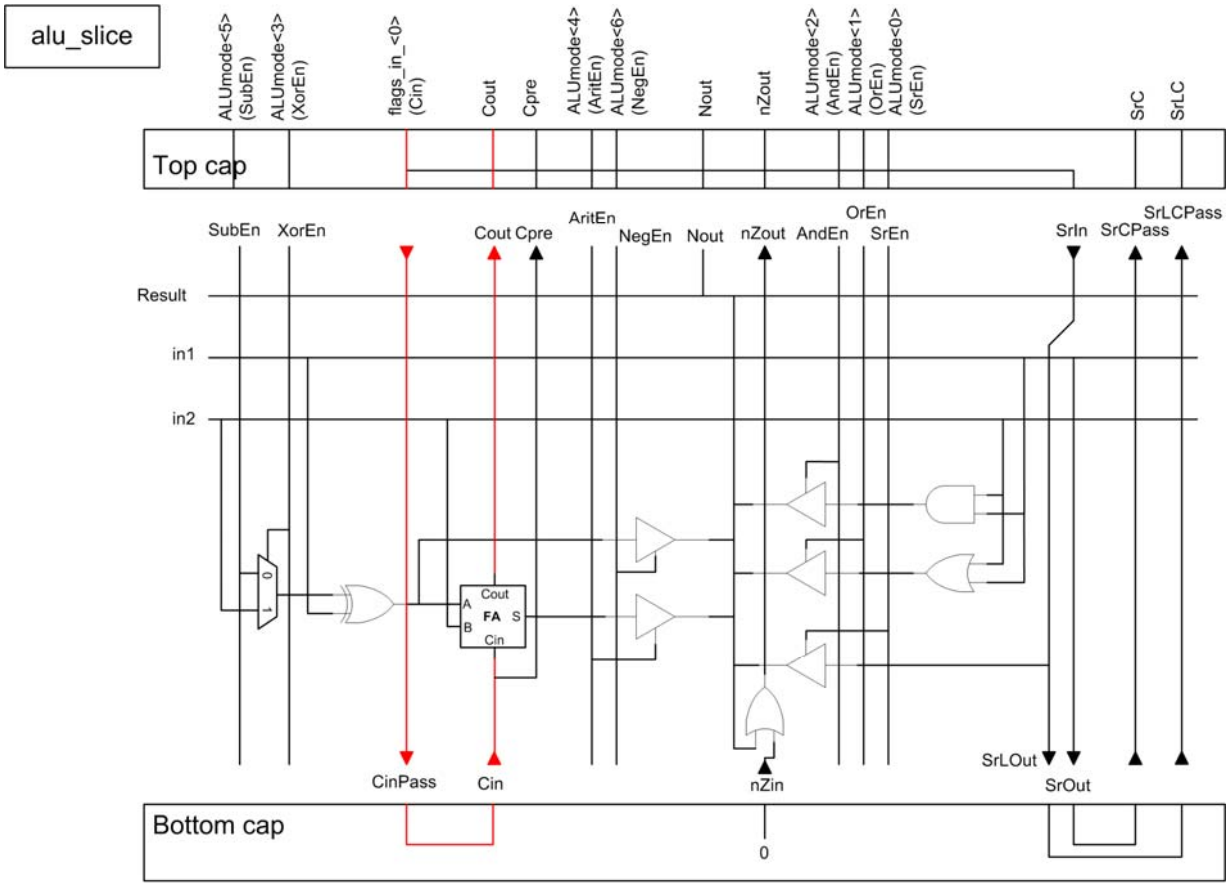
**MOUFA control logic Karnaugh maps**

Result target | BUS1 | BUS2 | ALU function | Next State | Sign extend | Misc | PC increase

| State / Instruction | Fetch | | | | Write | | | | Read | | | | e | Fetch | | | | Write | | | | Read | | | | e | Fetch | | | | Write | | | | Read | | | | e | Fetch | | | | Write | | | | Read | | | | e |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | as | ah | ds | dh | as | ah | ds | dh | as | ah | ds | dh | as | as | ah | ds | dh | as | ah | ds | dh | as | ah | ds | dh | as | as | ah | ds | dh | as | ah | ds | dh | as | ah | ds | dh | as | as | ah | ds | dh | as | ah | ds | dh | as | ah | ds | dh | as |
| CALL short_offset | DB | DB | IR | SP | DB | DB | DB | DB | DB | DB | DB | DB | PC | PC | PC | DB | C1 | C0 | C0 | PC | PC | PC | PC | DB | SP | PC | C0 | C0 | C0 | SP | SP | SP | C0 | C0 | C0 | C0 | C1 | SE | + | + | + | - | + | + | + | + | + | + | + | + | + |
| CALL R | DB | DB | IR | SP | DB | DB | DB | DB | DB | DB | DB | DB | PC | PC | PC | DB | C1 | C0 | C0 | PC | PC | PC | PC | DB | SP | O1 | C0 | C0 | C0 | SP | SP | SP | C0 | C0 | C0 | C0 | C1 | C0 | + | + | + | - | + | + | + | + | + | + | + | + | + |
| CALL offset | DB | DB | IR | SP | DB | DB | DB | DB | DB | DB | PC | DB | PC | PC | PC | DB | C1 | C0 | C0 | PC | PC | PC | PC | DB | SP | PC | C0 | C0 | C0 | SP | SP | SP | C1 | C1 | C0 | C0 | C0 | C1 | C0 | + | + | + | - | + | + | + | + | + | + | + | + | + |
| PUSH status | DB | DB | IR | SP | DB | DB | DB | DB | DB | DB | DB | DB | PC | PC | PC | DB | C1 | C0 | C0 | ST | ST | PC | PC | DB | SP | PC | C0 | C0 | C0 | SP | SP | SP | C0 | C0 | C0 | C0 | C1 | C0 | + | + | + | - | + | + | + | + | + | + | + | + | + |
| PUSH R | DB | DB | IR | SP | DB | DB | DB | DB | DB | DB | DB | DB | PC | PC | PC | DB | C1 | C0 | C0 | O1 | O1 | PC | PC | DB | SP | PC | C0 | C0 | C0 | SP | SP | SP | C0 | C0 | C0 | C0 | C1 | C0 | + | + | + | - | + | + | + | + | + | + | + | + | + |
| MOV {R} – R | DB | DB | IR | DB | DB | DB | DB | DB | DB | DB | DB | DB | PC | PC | PC | DB | PC | O2 | O2 | O1 | O1 | PC | PC | DB | SP | PC | C0 | C0 | C0 | C0 | SE | SE | C0 | C0 | C0 | C0 | C0 | C1 | C0 | + | + | + | + | + | + | + | + | + | + | + | + | + |
| JIFAR | DB | DB | IR | DB | DB | DB | DB | DB | DB | DB | PC | DB | PC | PC | PC | DB | PC | PC | PC | PC | PC | PC | PC | DB | SP | PC | C0 | C0 | C0 | C0 | SP | SP | C0 | C0 | C0 | C0 | C0 | C1 | C0 | + | + | + | + | + | + | + | + | + | + | + | + | + |
| MOV R-{R} | DB | DB | IR | DB | DB | DB | DB | DB | DB | DB | O2 | DB | PC | PC | PC | DB | PC | PC | PC | PC | PC | O1 | O1 | DB | SP | PC | C0 | C0 | C0 | C0 | C0 | C0 | C0 | C0 | SE | SE | C0 | C1 | C0 | + | + | + | + | + | + | + | + | + | + | + | + | + |
| ARITH R-IMM | DB | DB | IR | DB | DB | DB | DB | DB | DB | DB | ? | DB | PC | PC | PC | DB | PC | PC | PC | PC | PC | PC | PC | DB | SP | PC | C0 | C0 | C0 | ? | SP | SP | C0 | C0 | C0 | C0 | ? | C1 | C0 | + | + | + | ? | + | + | + | + | + | + | ? | + | + |
| POP reg | DB | DB | IR | DB | DB | DB | DB | DB | DB | DB | O2 | SP | PC | PC | PC | DB | PC | PC | PC | PC | PC | SP | SP | DB | SP | PC | C0 | C0 | C0 | C0 | SP | SP | C0 | C0 | C0 | C0 | C0 | C1 | C0 | + | + | + | + | + | + | + | + | + | + | + | + | + |
| POP status | DB | DB | IR | DB | DB | DB | DB | DB | DB | DB | ST | SP | PC | PC | PC | DB | PC | PC | PC | PC | PC | SP | SP | DB | SP | PC | C0 | C0 | C0 | C0 | SP | SP | C0 | C0 | C0 | C0 | C0 | C1 | C0 | + | + | + | + | + | + | + | + | + | + | + | + | + |
| ADD/MOV R–shrt | DB | DB | IR | O2 | DB | DB | DB | DB | DB | DB | DB | DB | PC | PC | PC | DB | ? | PC | PC | PC | PC | PC | PC | DB | SP | PC | C0 | C0 | C0 | SE | SP | SP | C0 | C0 | C0 | C0 | C0 | C1 | C0 | + | + | + | + | + | + | + | + | + | + | + | + | + |
| JIF_short | DB | DB | IR | ? | DB | DB | DB | DB | DB | DB | DB | DB | PC | PC | PC | DB | PC | PC | PC | PC | PC | PC | PC | DB | SP | PC | C0 | C0 | C0 | SE | SP | SP | C0 | C0 | C0 | C0 | C0 | C1 | C0 | + | + | + | + | + | + | + | + | + | + | + | + | + |
| DECIJNQ short | DB | DB | IR | ? | DB | DB | DB | DB | DB | DB | DB | DB | PC | PC | PC | DB | PC | PC | PC | PC | PC | PC | PC | DB | SP | PC | C0 | C0 | C0 | SE | SP | SP | C0 | C0 | C0 | C0 | C0 | C1 | C0 | + | + | + | + | + | + | + | + | + | + | + | + | + |
| ARITH R-R | DB | DB | IR | ? | DB | DB | DB | DB | DB | DB | DB | DB | PC | PC | PC | DB | O1 | PC | PC | PC | PC | PC | PC | DB | SP | PC | C0 | C0 | C0 | ? | SP | SP | C0 | C0 | C0 | C0 | ? | C1 | C0 | + | + | + | ? | + | + | + | + | + | + | ? | + | + |
| STC, STIE, LDIE | DB | DB | IR | DB | DB | DB | DB | DB | DB | DB | DB | DB | PC | PC | PC | DB | PC | PC | PC | PC | PC | PC | PC | DB | SP | PC | C0 | C0 | C0 | C0 | SP | SP | C0 | C0 | C0 | C0 | C0 | C1 | C0 | + | + | + | + | + | + | + | + | + | + | + | + | + |
| NOP | DB | DB | IR | DB | DB | DB | DB | DB | DB | DB | DB | DB | PC | PC | PC | DB | PC | PC | PC | PC | PC | PC | PC | DB | SP | PC | C0 | C0 | C0 | C0 | SP | SP | C0 | C0 | C0 | C0 | C0 | C1 | C0 | + | + | + | + | + | + | + | + | + | + | + | + | + |
| i-cycle | DB | SP | IR | DB | DB | SP | DB | DB | DB | SP | DB | DB | PC | PC | PC | PC | PC | C1 | C1 | PC | PC | PC | PC | PC | PC | C1 | C0 | C0 | C0 | C0 | SP | SP | C0 | C0 | C0 | C0 | C0 | C0 | C1 | + | + | + | + | - | - | + | + | + | + | + | + | + |
| CALL short_offset | F | F | F | W | W | W | W | FE | R | R | R | F | !E | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | | | | | | | | | | | | | | i | | | | | | | | | | | |
| CALL R | F | F | F | W | W | W | W | FE | R | R | R | F | !E | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | Next sub state: | | | | | | | | | | | | | | i | | | | | | | | | | | |
| CALL offset | F | F | F | W | W | W | W | R | R | R | R | F | !E | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | ah | ds | dh | as | ah | ds | dh | as | ah | ds | dh | as | as | | | | | i | | | | | | | |
| PUSH status | F | F | F | W | W | W | W | F | R | R | R | F | !E | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | | | | | | | | | | | | | | i | | | | | | | | | | | |
| PUSH R | F | F | F | W | W | W | W | F | R | R | R | F | !E | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | | | | | | | | | | | | | | i | | | | | | | | | | | |
| MOV {R} – R | F | F | F | W | W | W | W | F | R | R | R | F | !E | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | | | | | | | | | | | | | | i | | | | | | | | | | | |
| JIFAR | F | F | F | ? | W | W | W | F | R | R | R | F | !E | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | | | | | | | | | | | | | | i | | | ? | | | | | | | |
| MOV R-{R} | F | F | F | R | W | W | W | F | R | R | R | F | !E | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | | | | | | | | | | | | | | i | | | | | | | | | | | |
| ARITH R-IMM | F | F | F | R | W | W | W | F | R | R | R | F | !E | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | | | | | | | | fla,cen? | | | | | | fla,cen? | | | | i | | | | | | | |
| POP reg | F | F | F | R | W | W | W | F | R | R | R | F | !E | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | | | | | | | | | | | | | | ie? | | | | i | | | | | | | |
| POP status | F | F | F | R | W | W | W | F | R | R | R | F | !E | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | | | | | | | | | | | | | | fla | | | | i | | | | | | | |
| ADD/MOV R–shrt | F | F | F | F | W | W | W | F | R | R | R | F | !E | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | | | | | | | | fla | | | | | | | | | | i | | | | | | | |
| JIF_short | F | F | F | F | W | W | W | F | R | R | R | F | !E | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | | | | | | | | | | | | | | i | | | | | | | | | | | |
| DECIJNQ short | F | F | F | F | W | W | W | F | R | R | R | F | !E | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | | | | | | | | *LoadI,deci | | | | | | | | | | i | | | | | | | |
| ARITH R-R | F | F | F | F | W | W | W | F | R | R | R | F | !E | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | | | | | | | | fla,cen? | | | | | | fla,cen? | | | | i | | | | | | | |
| STC, STIE, LDIE | F | F | F | F | W | W | W | F | R | R | R | F | !E | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | ie | ie | ie | ie, fla | | | | | | | | | | | | | | i | | | | | | | |
| NOP | F | F | F | F | W | W | W | F | R | R | R | F | !E | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | | | | | | | | | | | | | | i | | | | | | | | | | | |
| i-cycle | F | F | F | F | W | W | W | FE | R | R | R | F | !E | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | | | | | di | di | di | di | | | | | di, enc | i | | | | | | | | | | |

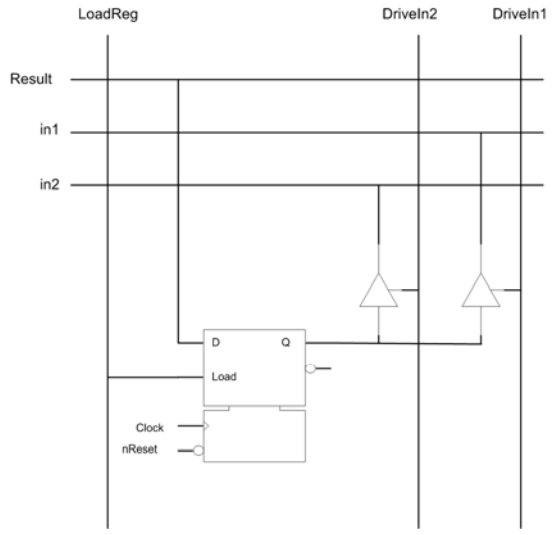
NOTES: ((next=F and dh and not E) or e)->Interrupt_check | No bus must be driven from DB if not DataSetup and (Fetch or Read) | i-cycle always wins

## *Appendix N. Datapath schematics*

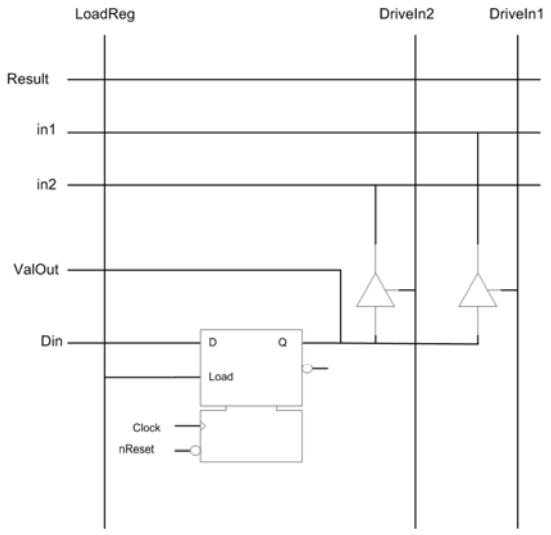



| ALUmode | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| Function | NegEn | SubEn | AritEn | XorEn | AndEn | OrEn | SrEn |
| FN_ADD | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| FN_SUB | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| FN_XOR | 1 | - | 0 | 1 | 0 | 0 | 0 |
| FN_NOT | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| FN_AND | 0 | - | 0 | - | 1 | 0 | 0 |
| FN_OR | 0 | - | 0 | - | 0 | 1 | 0 |
| FN_SRC | 0 | - | 0 | - | 0 | 0 | 1 |

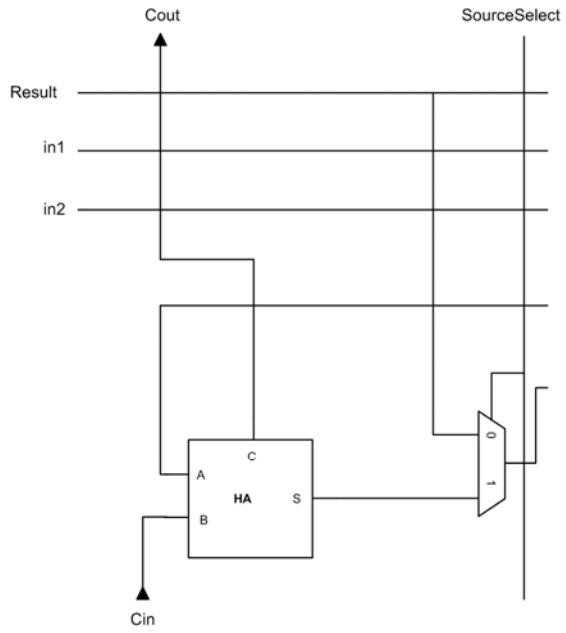
register

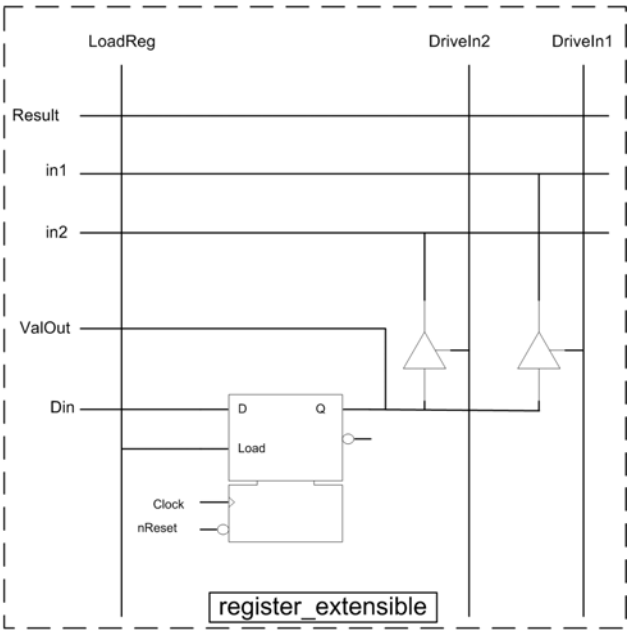


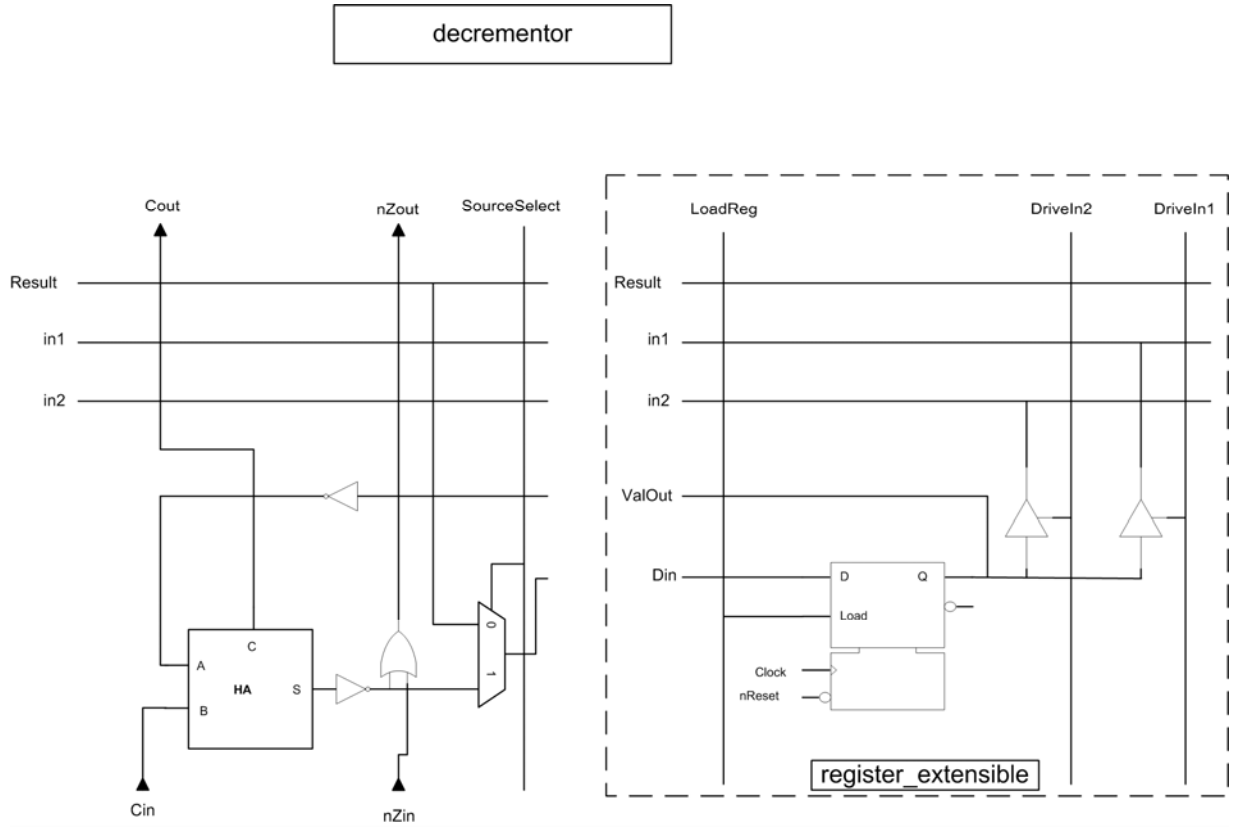

register_extensible

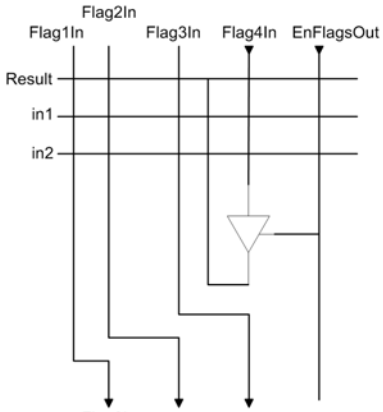



incrementer



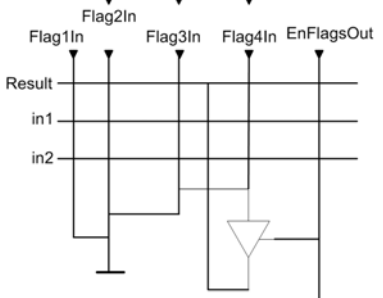




decrementor
Cout
nZout
SourceSelect
Result
in1
in2
C
A
HA
S
B
0
1
Cin
nZin
LoadReg
DriveIn2
DriveIn1
Result
in1
in2
ValOut
Din
D
Q
Load
Clock
nReset
register_extensible

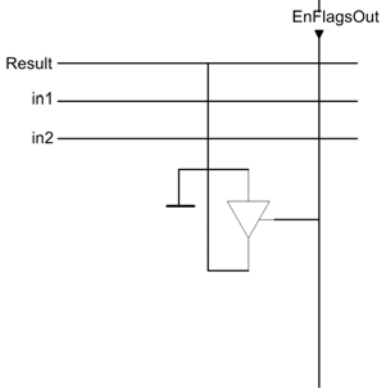



flag_active

… x4

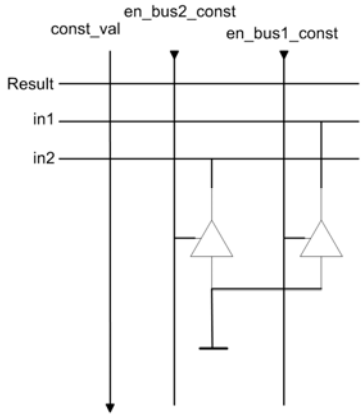



flag_terminal



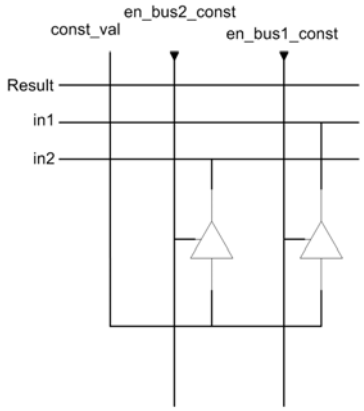

flag_dummy

… x11

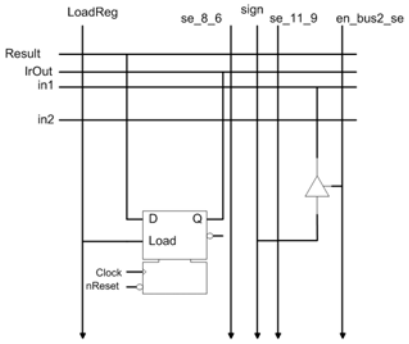



const_all

… x15

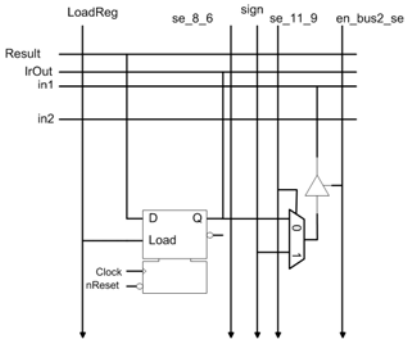



const_bottom



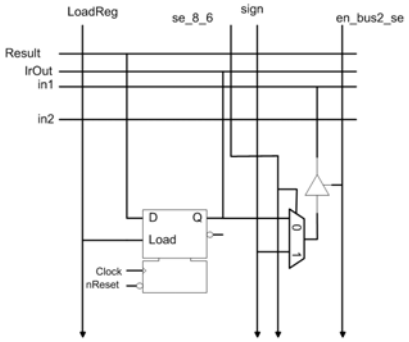

ir_12_15

… x4

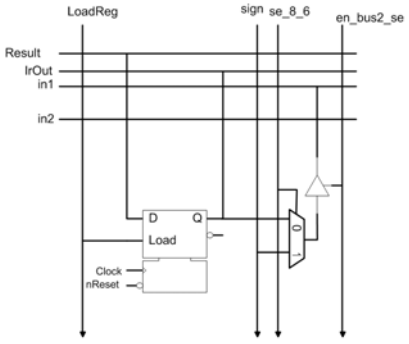



ir_09_11

… x3

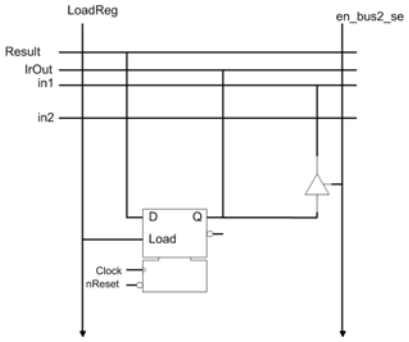



ir_08_8




ir_06_07

… x2

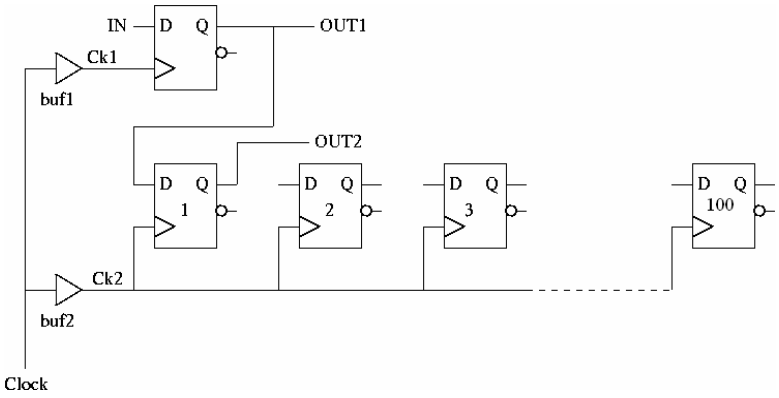



ir_00_05

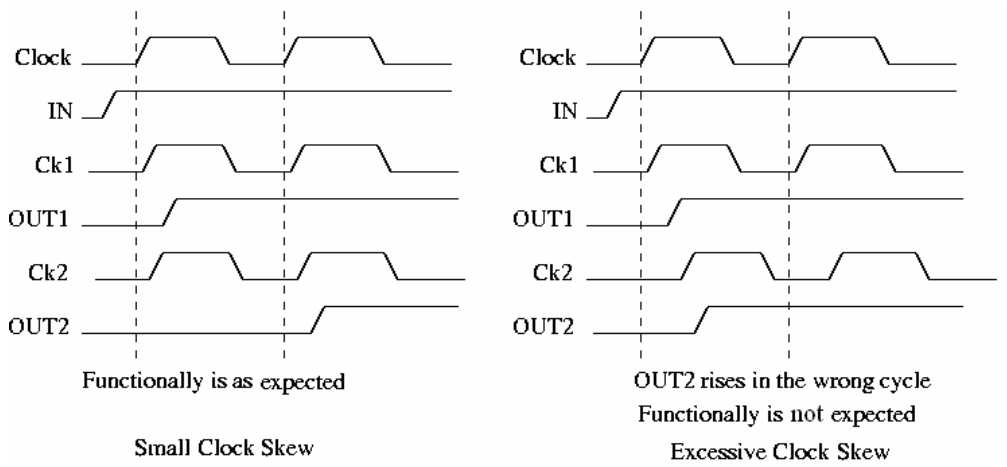
… x6

## *Appendix O. Testing the leftbuf*

The figure bellow below illustrates the circuit used to test leftbuf for excessive clock skew.

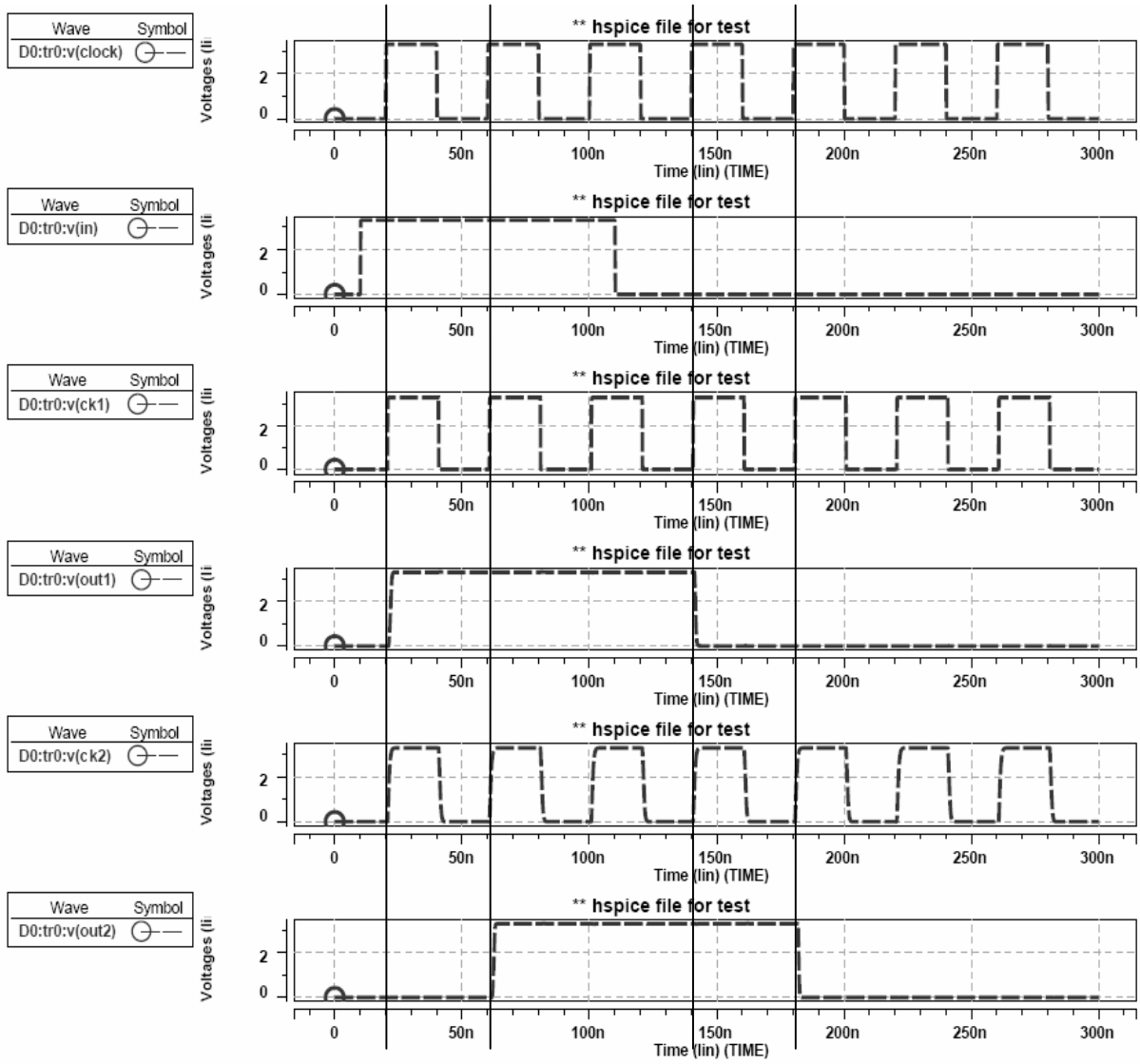



Buf1 and Buf2 are identical clock buffers which are used to drive two rows of scandtypes. But as the row one contains only one D-type and row2 contains 100 D-types, hence buf1 drives lower load compared to buf2 which drives 100 scandtypes. Due to this unequal loads being driven, the signal Ck2 is delayed/skewed with respect to Ck1. There is possibility of race hazard if the clock skew is too large. The following figure shows the waveform for small clock skew and excessive clock skew.




Under normal conditions, as seen in Figure 3.1, change in the value of signal IN from ‘0’ to ‘1’ is passed on to OUT1 after the first rising edge of the Ck1. The change is then passed on to OUT2 after second rising edge of Ck2. If there is excessive clock skew then OUT2 will change in the same clock cycle as OUT1 since the change in OUT1 will appear too early the input of the first scandtype on the second row, hence resulting in race hazard.

Hspice simulation results of the leftbuf using the circuitry described above can be seen in the figure bellow. We can see that the pulse switches in the correct clock edge as expected.




## *Appendix P. leftbuf's analytical calculations*

scandtype's width 32μm

length of a row 3000μm

Row's actual length up to the moment: 2918.2

→ 3000 / 32 = 94 dtypes

Specification of the problem:

We want to have a buffer that will have the following specifications:

- Equal rise/fall time
- Give small skew compared to the setup time (?) maybe 1/10 of the setup time

- Examine dinstictly nreset, clock and Test

87 dtypes → 87 x 4 transistors ~ 87 x 2 = 174 inverters

ln(174) = 5.16 stages

Non-Inverting → 4 stages

Stage ratio (initial) $\sqrt[4]{184} = 3.68$

3 stages to the output buffer sized 129.8 / 2 = 64.9 times minimum inverter:
$\sqrt[3]{64.9} = 4.02$

| Isometric | Last stage boosted |
|---|---|
| 47.9 | 64.9 |
| 13.2 | 16.1 |
| 3.63 | 4.0 |
| 1 | 1 |

All inverters symmetrical (P:N ~ (3-3.5):1)

Divide and conquer.

Problem 1: Design symmetrical output buffer such as output skew is as low as possible.
SubProblem a): First the falling edge
SubProblem b): Then adjust the rising edge to have equal rise/fall.

Problem 2: Design its driving buffers to provide equal rise/fall time and equal prop/delay.

Problem 3: Optimize the layout

**Problem 1:**

a) 0.83ns is our fastest propagation delay (nout, scandtype, 0oC). Our fastest setup time is: 0.56ns. This means that we have to have clock skew equal to 1.39ns to have a race condition. To avoid race conditions we require a maximum skew of 1.39/2 = 0.7ns

Results:

```
PMOS UP_HEIGHT    = 13.3
PMOS UP_ROWS      =  25

PMOS DOWN_HEIGHT  = 9.5
PMOS DOWN_ROWS    =  12

NMOS DOWN_HEIGHT  = 11.8
```

NMOS DOWN_ROWS = 11

TOTAL P_LENGTH = 13.3 * 25 + 9.5 * 12 = 332.5 + 114 = 446.5
TOTAL N_LENGTH = 11.8 * 11 = 129.8

RATIO P/N = 446.5 / 129.8 = 3.44

Facts:

| fall_delay | rise_delay | prop_rise | prop_fall | dif_delay |
|---|---|---|---|---|
| 0.45 | 0.42 | 0.26 | 0.33 | -0.03 |
| 0.11 | 0.12 | 0.03 | 0.06 | 0.02 |

skew_rise: 0.23
skew_fall: 0.27

**Problem 2:**

Module inverter 3:

Estimated:

TOTAL P_LENGTH = 446.5 / 4.02 = 111.1
TOTAL N_LENGTH = 129.8 / 4.02 = 32.3

NMOS DOWN_ROWS = 3 (3 * **10.8** = 32.4)
PMOS UP_ROWS = (3+2) + 2
PMOS DOWN_ROWS = 2

Because: (X + 2 + 3) * 13.3 + X * 9.5 = 111.1 → X = (111.1 – (5 * 13.3)) / (9.5+13.3) → X = 1.96

We round X to 2. Now we have:

TOTAL P_LENGTH = (13.3 * 7) + (**9.0** * 2) = 111.1

Facts:

| fall_delay | rise_delay | prop_rise | prop_fall | dif_delay |
|---|---|---|---|---|
| 0.44 | 0.40 | 0.23 | 0.30 | -0.04 |
| 0.07 | 0.09 | 0.03 | 0.05 | 0.02 |

skew_rise: 0.24
skew_fall: 0.19

Module inverter 2:

Estimated:

TOTAL P_LENGTH = 111.1 / 4.02 = 27.6
TOTAL N_LENGTH = 32.3 / 4.02 = 8.0

27.6 / 3 = 9.2

Module inverter 1:

Estimated:

TOTAL P_LENGTH = 27.6 / 4.02 = 6.9
TOTAL N_LENGTH = 8.0 / 4.02 = 2.0

Final facts:

| fall_delay | rise_delay | prop_rise | prop_fall | dif_delay |
|---|---|---|---|---|
| 0.42 | 0.39 | 0.28 | 0.26 | -0.03 |
| 0.38 | 0.34 | 0.24 | 0.22 | -0.04 |
| 0.37 | 0.34 | 0.24 | 0.22 | -0.03 |
| **0.44** | **0.40** | **0.31** | **0.24** | **-0.04** |
| | | | | |
| 0.42 | 0.39 | 0.28 | 0.26 | -0.03 |
| 0.38 | 0.34 | 0.24 | 0.22 | -0.04 |
| 0.42 | 0.37 | 0.25 | 0.22 | -0.05 |
| **0.07** | **0.09** | **0.05** | **0.03** | **0.03** |
| | | | | |
| | total: | **0.98** | **1.03** | |
| | total: | **0.78** | **0.78** | |
| | | | | |
| | **skew_rise** | 0.20 | | |
| | **skew_fall** | 0.25 | | |

* Loaded version

After layout optimization:

| fall_delay | rise_delay | prop_rise | prop_fall | dif_delay |
|---|---|---|---|---|
| **0.44** | **0.40** | | | **-0.04** |
| 0.07 | 0.10 | | | 0.03 |
| | | | | |
| | total: | **0.94** | **1.00** | |
| | total: | 0.75 | 0.75 | |
| | | | | |
| | skew_rise | 0.20 | | |
| | skew_fall | 0.24 | | |

nReset research:

87 dtypes → 87 x 3 transistors = 261 inverters

ln(261) = 5.6 stages

Non-Inverting → 4 stages

Stage ratio (initial) $\sqrt[4]{261} = 4.02$

3 stages to the output buffer sized 129.8 / 2 = 64.9 times minimum inverter:
$\sqrt[3]{89.6} = 4.47$

| Isometric | Last stage boosted |
|---|---|
| 64.9 | 89.6 |
| 16.2 | 20.0 |
| 4.0 | 4.5 |
| 1.0 | 1.0 |

PMOS UP_HEIGHT = 13.3
PMOS UP_ROWS = 35

PMOS DOWN_HEIGHT = 8.9
PMOS DOWN_ROWS = 17

NMOS DOWN_HEIGHT = 11.2
NMOS DOWN_ROWS = 16

TOTAL P_LENGTH = 13.3 * 35 + 8.9 * 17 = 465.5 + 151.3 = 616.8
TOTAL N_LENGTH = 11.2 * 16 = 179.2

RATIO P/N = 616.8 / 179.2 = 3.44

**Problem 2:**

Module inverter 3:

Estimated:

TOTAL P_LENGTH = 616.8 / 4.47 = 138.0
TOTAL N_LENGTH = 179.2 / 4.47 = 40.1

Real:

TOTAL P_LENGTH = (13.3 * 9) + (**6.1** * **3**) = 138.0

TOTAL N_LENGTH = **10** * **4** = 40.0

Module inverter 2:

Estimated:

TOTAL P_LENGTH = 138.0 / 4.47 = 30.9
TOTAL N_LENGTH = 40 / 4.47 = 8.9

30.9 / 3 = 10.3

Module inverter 1:

Estimated:

TOTAL P_LENGTH = 30.9 / 4.47 = 6.9
TOTAL N_LENGTH = 8.9 / 4.47 = 2.0

Final facts:

| | fall_delay | rise_delay | prop_rise | prop_fall | dif_delay |
|---|---|---|---|---|---|
| Test | 0.43 | 0.41 | 0.94 | 0.99 | -0.02 |
| nReset | 0.37 | 0.35 | 0.98 | 1.02 | -0.02 |
| Clock | **0.45** | **0.41** | **0.96** | **1.01** | **-0.04** |
| | | Lightly loaded | | | |
| Test | 0.08 | 0.11 | 0.75 | 0.76 | 0.03 |
| nReset | 0.07 | 0.10 | 0.80 | 0.81 | 0.03 |
| Clock | **0.08** | **0.11** | **0.77** | **0.77** | **0.03** |
| | | Skews | | | |
| Test | | | 0.19 | 0.23 | |
| nReset | | | 0.17 | 0.21 | |
| Clock | | | 0.19 | 0.24 | |

**Final: 83 Dtypes in a 3000um row**

| | fall_delay | rise_delay | prop_rise | prop_fall | dif_delay |
|---|---|---|---|---|---|
| Test | 0.42 | 0.40 | 0.94 | 0.98 | -0.02 |
| nReset | 0.36 | 0.35 | 0.97 | 1.01 | -0.02 |
| Clock | **0.44** | **0.40** | **0.95** | **1.00** | **-0.04** |
| | | Lightly loaded | | | |
| Test | 0.08 | 0.11 | 0.75 | 0.76 | 0.03 |
| nReset | 0.07 | 0.10 | 0.80 | 0.81 | 0.03 |
| Clock | **0.08** | **0.11** | **0.77** | **0.77** | **0.03** |
| | | Skew | | | |
| Test | | | 0.19 | 0.23 | |
| nReset | | | 0.17 | 0.20 | |
| Clock | | | 0.19 | 0.23 | |