ECE451: VLSI Systems and Design
Lab 2: Datapath Slice Design
Due: February 6, 2015 (2-5pm, BA5000)

1 Introduction

In this and the next lab, you will design the datapath of a simplified version of the 4-bit AMD 2901 microprocessor. In this lab you will create the 1-bit version of the datapath, and in the next lab you will combine the 1-bit slices to complete the datapath and simulate it. In Lab 4 and 5, you will complete the processor design by building the control unit.

Nowadays you can rarely find a 2901 as a separate IC, but it might serve as a processor core in other bigger designs. For your interests some data sheets of the 2901 can be accessed from the course web page[1]. You need not understand the data sheets for this lab.

The top-level architecture of the processor you will design is shown in Figure 1. In this lab, we focus on building a 1-bit slice of the datapath. The design is much more complex than those in Lab 1, so a good floorplan is very important. **Read carefully the entire document, particularly the Design Recommendation section, before doing any real work.** The amount of work required from you is quite heavy, so start early!

There will be an in-lab marking session at the first week of Lab3. The questions will be released on February 6th, i.e. after you submit your Lab2 report. You will have a weekend to review the questions and you will be asked couple of them during the in-lab marking session. Ideally, once you complete Lab2 with a good understanding of the bit-slice operation, you should be able to answer those questions without any additional work.

2 Datapath

As shown in Figure 2, the datapath is a 4-bit processor slice that consists of an arithmetic/logic unit (ALU), a one-bit shifter, a 4-word register file, and a multiplexer for operand selection.
The 2901 cell uses a two-phase non-overlapping clock. The two clocks are denoted $\phi 1$ and $\phi 2$ in the text, and Phi1 and Phi2 in the schematics and layouts. Both $\phi 1$ and $\phi 2$ are used to latch values off a control bus (not shown in the figure). The datapath evaluates in two stages: first, operands are read out of the register file or latched off the D bus during $\phi 1$; then, an ALU operation is performed and the result is written into the register file during $\phi 2$.

2.1 Signal flow

Figure 1 and 2 describe the overall structure of the 4-bit CPU. Figure 3 describes the flow of data and control signals in the CPU. The whole 4-bit CPU is built by placing 4 1-bit slices one on top of another, with the

[1] The uncompressed files are pretty big!

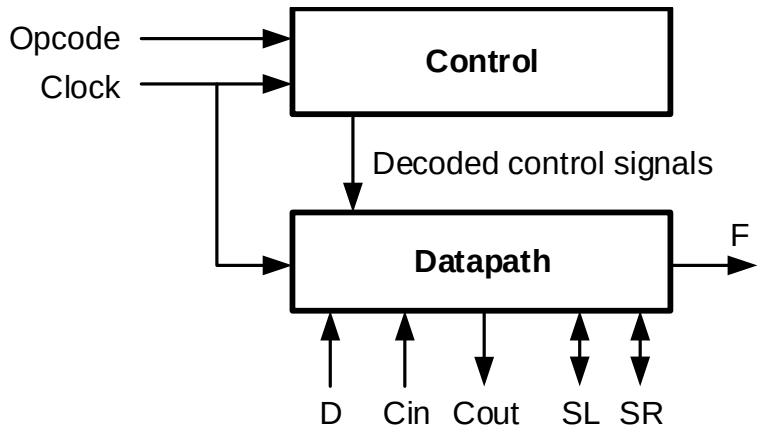


Figure 1: Top-level block diagram of the processor. D refers to the data bus. SL and SR refer to the MSB and LSB that shift in and out of the chip.

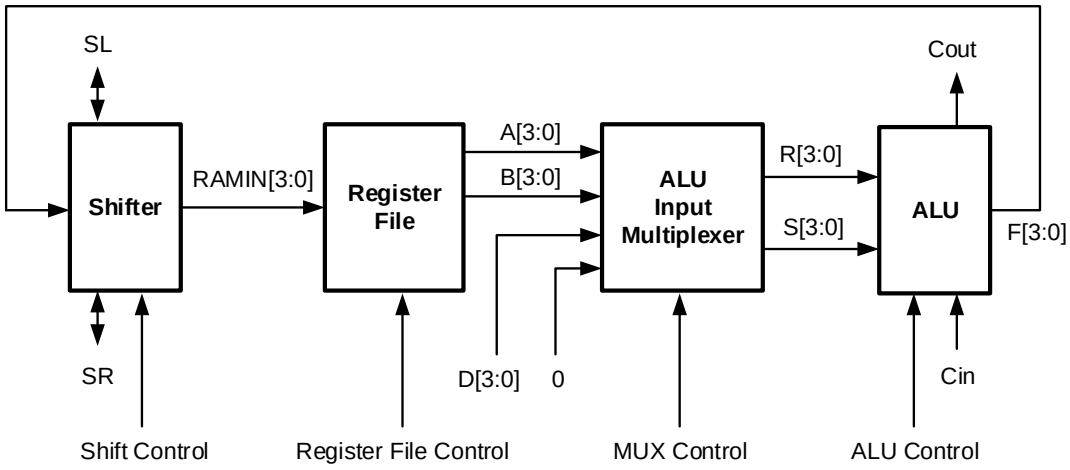


Figure 2: Block diagram of the datapath component (4-bits).

Most Significant Bit (MSB) slice (onebit[3]) on top, and the Least Significant Bit (LSB) slice (onebit[0]) at the bottom. These bit-slices are identical, and in this lab you will design and lay out one complete bit-slice.

A microprocessor typically reads some values from the register file, processes it through the ALU and writes back the result to the register file after the operation is performed. Data thus flows from left to right, from the read ports of Regfile through the whole 1-bit slice eventually looped back to the write ports of Regfile through the Shifter. The carry signal flows up from Cin to Cout. When shifting is performed, data that is looped back after the inverter at the end of the bit-slice is shifted vertically up or down to the adjacent bit and eventually written to the Regfile of the adjacent bit (depending on which direction the shifting is done).

Control signals come into the datapath from above and flow perpendicularly to the data flow. **It is critical to lay out control signals vertically because they are to be connected to the corresponding control signals of other bit-slices when you create the 4-bit datapath in Lab 3.** The control signals (including clocks) for each component are indicated in the diagrams in the next section.

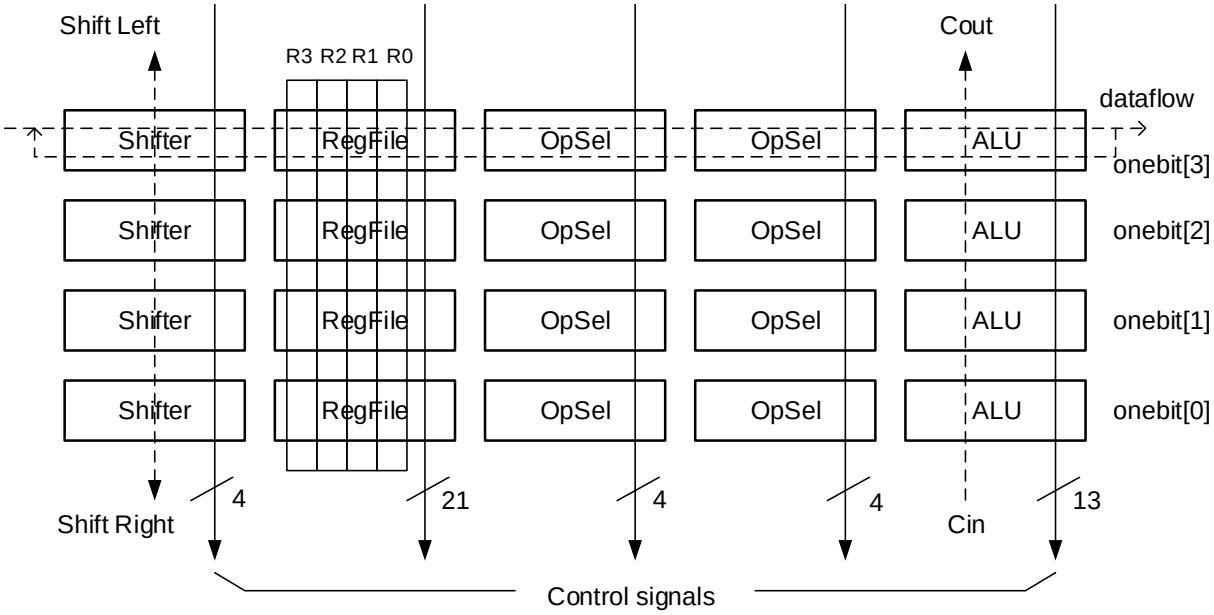


Figure 3: Flow of data and control signals.

3 Component Cells

The following cells only operate on 1-bit of the data. They are a part of the 1-bit datapath. You will enter these schematics in SUE and lay them out in MAX. Again, **finish reading the entire document before doing so.**

3.1 Register file

The register file consists of four registers R3, R2, R1 and R0, each of which has 4 bits that span across the 4 bit-slices. It supports two simultaneous reads through read port A and B (e.g., reading R0 and R1 for performing R0+R1) and one write.

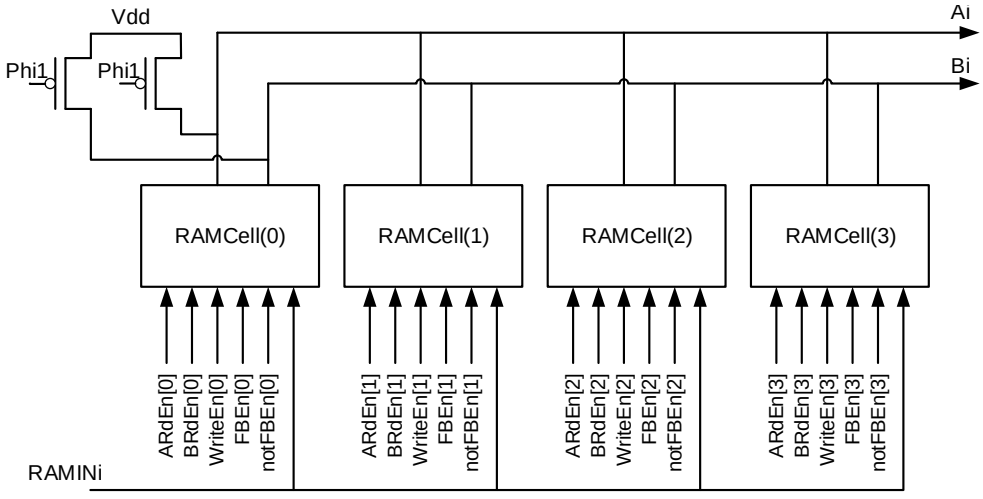


Figure 4: Block diagram of RegFile with precharge transistors for A and B.

As shown in Figure 4, a bit-slice of the register file (RegFile) is composed of four RAM cells, which

represent 1-bit of the four registers. Each RAM cell provides two read ports, connected to one bit of the operand buses A and B respectively, and one write port, connected to one bit of the input bus RAMIN. The control signals select the individual cell(s) to be read during $\phi 1$ and the cell to be written during $\phi 2$. Each RAM cell takes 5 control signals: ARdEn, BRdEn, WriteEn, FBEn, notFBEn, whose purposes are described in Table 1. In addition, for each control signal, Table 2 specifies which phase it is active in, its value during the active phase and the value it should be precharged to (when it is inactive).

Table 1: RegFile control signals.

Receiving Cell	Control Signal Name	Purpose
RegFile	Phi1	Controls precharging of the A and B lines
	ARdEn[3:0], BRdEn[3:0]	Read enables for each cell
	WriteEn[3:0]	Write enable for each cell
	FBEn[3:0], notFBEn[3:0]	Control refresh for each cell
	Total: 21 signals	

Table 2: RAM cell input/output signals.

Signal	Active/Valid During	Active/Valid Value	Precharge Value
RAMINi	$\phi 2$	Shifter result	X
Ai	$\phi 1$	RAM contents	1
Bi	$\phi 1$	RAM contents	1
WriteEn	$\phi 2$	1 if written, or else 0	0
ARdEn	$\phi 1$	1 if A is read, or else 0	0
BRdEn	$\phi 1$	1 if B is read, or else 0	0
FBEn	all the time	0 if the cell is to be written, or else 1	

Figure 5 shows the internal structure of a RAM cell. The cell drives each operand bus through a single NMOS transistor, which cannot propagate a high-value well. Therefore, the operand buses A and B must be precharged high when not being driven, i.e., during $\overline{\phi 1}$. Since all four RAM cells in a RegFile are tied to the operand buses in parallel, the precharge transistors are in the RegFile cell as opposed to individual RAM cells.

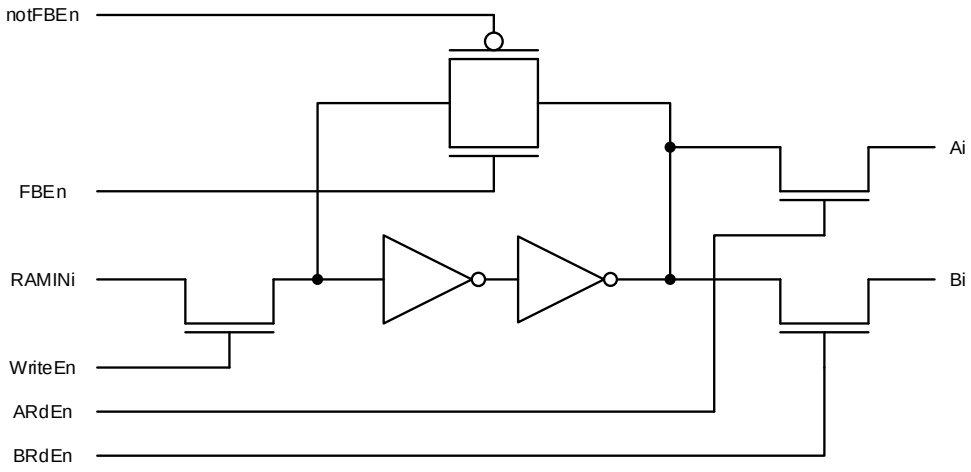


Figure 5: Schematic of a RAM cell.

The RAM cell is static in that it does not have to be refreshed externally like dynamic RAM. Instead, a transmission gate in the RAM cell refreshes the stored value unless new data is being written. The feedback path, controlled by FBEn (and notFBEn), is enabled during the read phase to refresh the content of the RAM cell, and is disabled during the write phase iff the cell is to be written.

3.2 Operand selector

The next cell in the data path is the ALU input multiplexer, which selects the operands fed to the ALU. Another function of the cell is to latch the values coming out of the register file and/or in from the external D bus so that they remain stable during the ALU evaluation phase ($\phi 2$).

Referring back to Figure 2, the multiplexer has four inputs (Ai, Bi, D, and 0) and two outputs (R and S). Two 2×1 multiplexers are used. **The S input of the ALU is selected from 0 and Ai from the register file. The R input of the ALU is selected from D and Bi from the register file.** The inputs and outputs must be connected in this manner in order for your datapath to work properly with the controller in Lab 4.

Figure 6 shows the schematic of the operand selector (OpSel), which includes a 2×1 multiplexer followed by a latch (implemented as a transmission gate). Two instances of this cell are required: one for each ALU input.

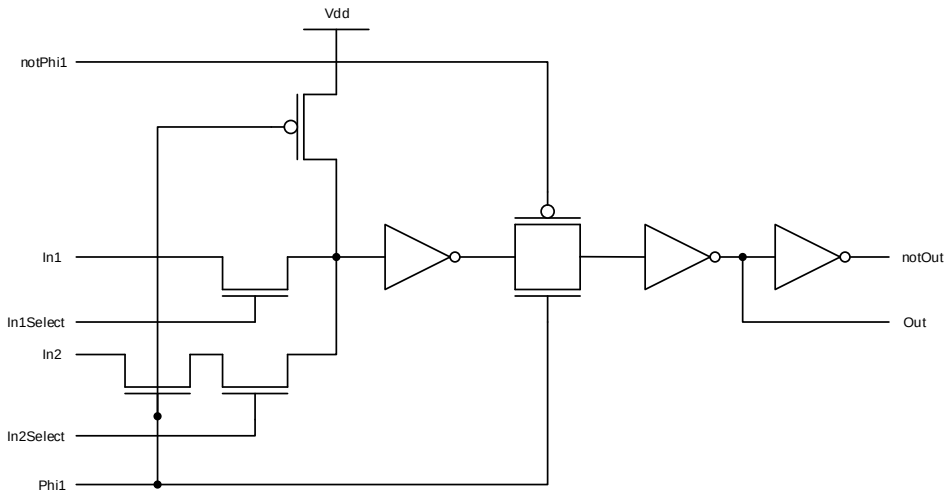


Figure 6: Schematic of an OpSel cell.

The input multiplexer in OpSel is based on dynamic logic, so a precharging transistor must be added at its output. **The enable paths for the two inputs In1 and In2 are not symmetric.** You must consider this difference when using OpSel in your bit-slice design.

The latch in OpSel passes the selected input to the output during $\phi 1$ and retains the previous value otherwise. This provides the ALU with a stable input during $\phi 2$. The input selectors (In1Select and In2Select) are only stable during the latching phase $\phi 1$; they may be either 0 or 1 otherwise.

3.3 ALU

The ALU can perform several operations, as listed in Table 4. The ALU consists of two Func cells, a TFunc cell, a Carry cell and an inverter, as shown in Figure 7. The Func and TFunc cells are generic gates which can be programmed to generate any of the sixteen possible functions of two variables. Such flexibility allows

Table 3: OpSel control signals.

Receiving Cell	Control Signal Name	Purpose
OpSel ($\times 2$)	Phi1 ($\times 2$), notPhi1 ($\times 2$)	Keep the four signals separate during layout; do not connect them internally in this component
	ASelect, zeroSelect	Control the data selected for OpSel #1
	BSelect, DSelect	Control the data selected for OpSel #2
	Total: 8 signals (4 per OpSel)	

us to implement the different ALU operations by simply changing the programming of these generic gates. Since the result of the ALU drives a bus which runs the length of the datapath, **a strong inverting buffer (e.g., of a double width) must be added at the TFunc output to boost the drive capability.** Table 5 describes the control signals of the ALU.

Table 4: ALU operations.

OpCode	Function
0	$R + S$
1	$S - R$
2	$R - S$
3	$R \vee S$
4	$R \wedge S$
5	$\overline{R} \wedge S$
6	$R \oplus S$
7	$\overline{R \oplus S}$

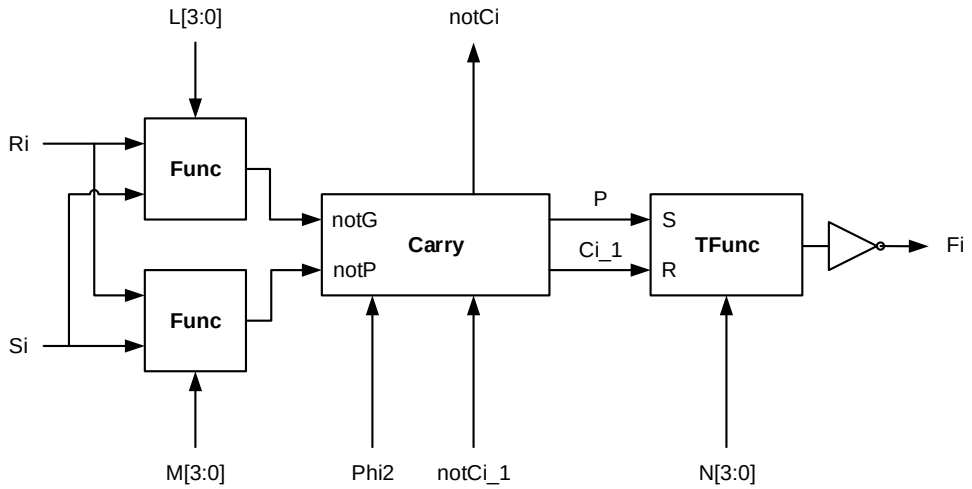


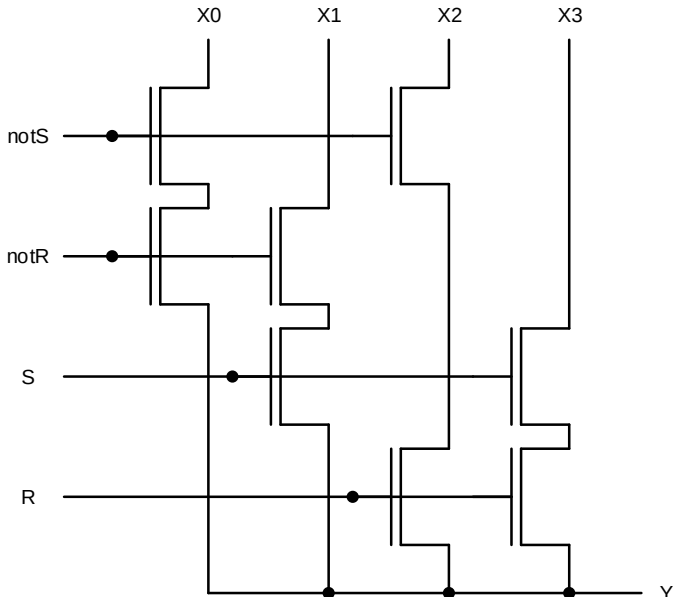
Figure 7: ALU block diagram. The notR and notS inputs of the Func and TFunc cells are not shown.

3.3.1 Func

The generic Func cell is shown in Figure 8a. It functions as a simple multiplexer of the X[3:0] bus under the control of R and S signals. We implement the cell using dynamic logic in order to minimize layout area and increase performance.

Table 5: ALU control signals.

Receiving Cell	Control Signal Name	Purpose
ALU	Phi2	For precharging the carry cell
	L[3:0], M[3:0], N[3:0]	Determine what ALU operation to perform
	Total: 13 signals	



(a) Schematic.

X[3:0]	Y
0	0
1	$\overline{R} \wedge \overline{S}$
2	$\overline{R} \wedge S$
3	$\overline{R}$
4	$R \wedge \overline{S}$
5	$\overline{S}$
6	$R \oplus S$
7	$\overline{R} \vee \overline{S}$
8	$R \wedge S$
9	$\overline{R \oplus S}$
A	S
B	$\overline{R} \vee S$
C	R
D	$R \vee \overline{S}$
E	$R \vee S$
F	1

(b) Y as a function of R and S.

Figure 8: The Func cell.

By selecting the proper value for X[3:0], the Func cell can generate any function of R and S as shown in Figure 8b. The Func cell requires both active-high and active-low versions of the R and S inputs (i.e., R and notR, S and notS). Since the cell consists only of NMOS transistors and thus cannot propagate a high-value well, it can only be used when the output F is precharged high. During precharge, the R, S and X[3:0] inputs must be controlled such that no path to ground exists (so that the charge on the output is lost erroneously). There are two ways to guarantee this: 1) the gates of all the transistors (i.e., R, S, notR and notS) must be held low, or 2) all X[3:0] inputs must be isolated from ground (i.e., driven high or not driven). We choose the second option because it is easier to control X[3:0] than R and S. Since the output Y is evaluated during $\phi 2$, it is precharged during $\overline{\phi 2}$. **We must ensure that X[3:0] are kept high during $\overline{\phi 2}$. Also, we must guarantee that R and S do not change during the evaluation phase $\phi 2$, in order to prevent erroneous discharge.**

3.3.2 TFunc

We have seen that, since the Func cell uses dynamic logic, we must guarantee that the inputs meet certain constraints. At the last stage of the ALU, however, this is not possible and thus we require another cell which can propagate both high and low logic values well. The TFunc cell, shown in Figure 9, provides this functionality. The addition of PMOS transistors allows the cell to propagate high logic values without degradation so it can be used in both dynamic and static situations. The logic function performed by the TFunc cell is identical to that of the Func cell.

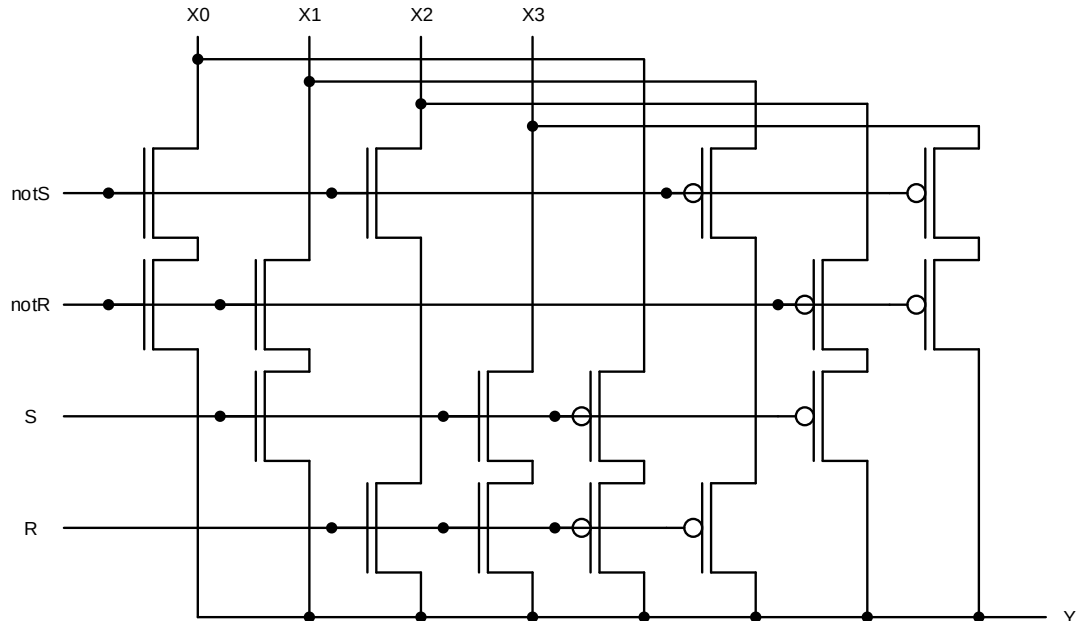


Figure 9: Schematic of a TFunc cell.

3.3.3 Manchester Carry Adder

Among the operations the ALU supports, the arithmetic operations are significantly more difficult to implement than the others. We will concentrate on implementing an adder with the Func and TFunc cells. We can expect that the flexibility of the generic cell will enable us to implement the other operations simply by changing the programming of the generic gates.

Many different adder architectures exist, the most well-known of which are the ripple carry and carry lookahead adder. Ripple carry is the simplest adder implementation, but it suffers from poor performance and large layout area. Instead, most high performance processors use carry lookahead, but this has several drawbacks for our design. Carry lookahead requires very large fan-in gates and has an irregular structure. Moreover, the high-fan-in gates can result in performance problems for an ALU as small as the 4-bit slice we are considering. The Manchester carry adder has an especially attractive realization in CMOS. By using dynamic logic, a middle ground between ripple carry and carry lookahead is achieved. For a 4-bit design, the Manchester carry adder may even perform better than a carry lookahead adder.

Figure 10 shows the function each of the Func and TFunc cells takes on when the ALU is programmed to be a Manchester carry adder, which is derived from the definition of P and G in Figure 8.18 of Weste. The two Func cells are used to generate the inverted versions of P and G respectively (i.e., notP and notG). The carry cell, shown in Figure 11, generates the inverted version of carry-out (notCi). The TFunc cell combines P and cary-in (Ci_1) to generate the *inverted* result, which is flipped back to the correct version by the inverter at TFunc's output. Keep in mind that Figure 10 omits the inverted versions of R and S as inputs of the Func and TFunc cells. You need to have them in your design.

By comparing the block diagram in Figure 7 with the functional diagram in Figure 10, we can see that the Func cell driving notP must generate $\overline{\mathrm{R} \oplus \mathrm{S}}$ while the one driving notG must generate $\overline{\mathrm{R} \wedge \mathrm{S}}$, or by Demorgan's theorem, $\mathrm{notR} \vee \mathrm{notS}$. If the TFunc cell is programmed to generate $\overline{\mathrm{Ci\_1} \oplus \mathrm{P}}$, the result of the ALU is R + S. Referring back to Figure 8b, we require $L = 7$, $M = 9$, and $N = 9$ (in hexadecimal).

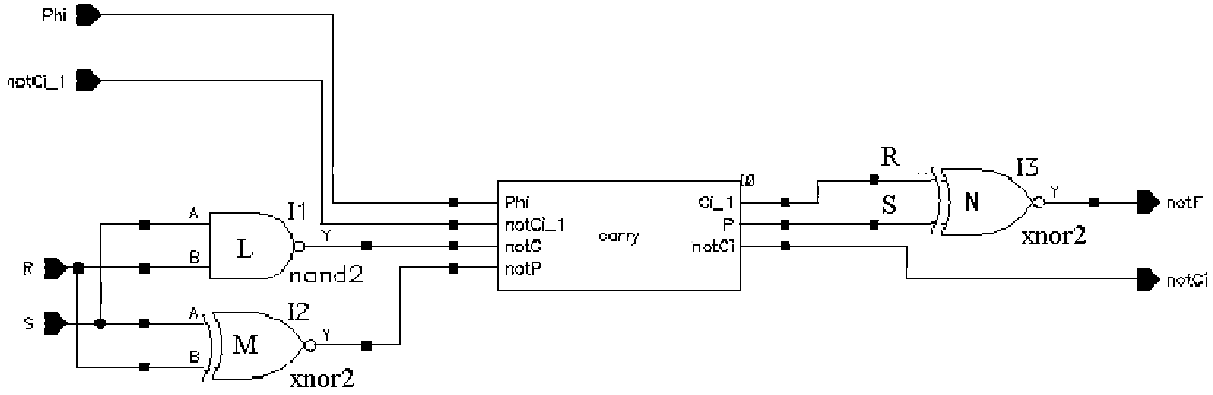


Figure 10: ALU operation when configured as $R + S$.

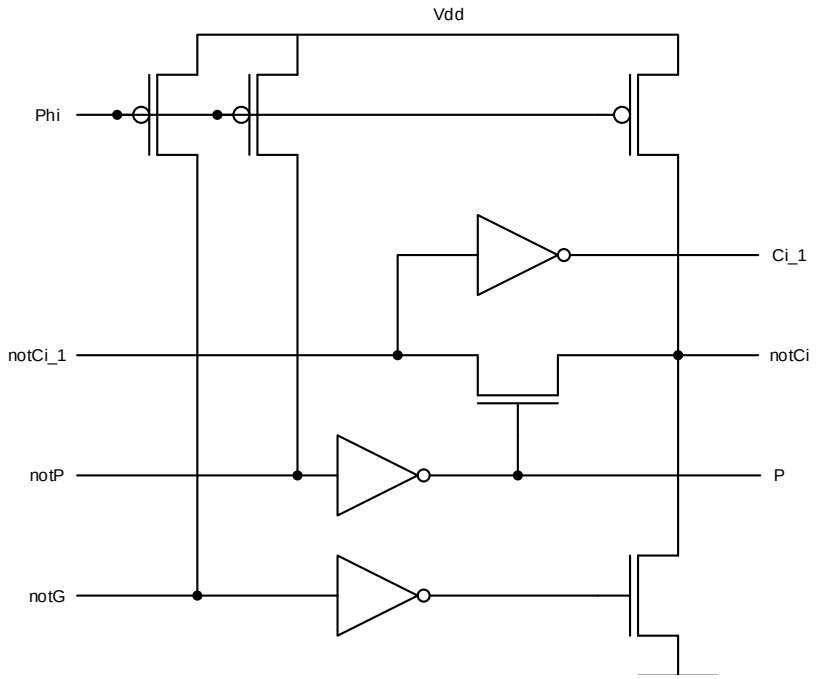


Figure 11: Manchester carry cell.

Let us next consider the R $\vee$ S operation (OpCode 3). We can actually generate this with a single Func or TFunc cell, but we would like to realize it using the circuit in Figure 7 by just changing the values of control signals L, M and N. The easiest way to do this is to compute R $\vee$ S using the notP Func cell and then setting the TFunc cell to simply pass the S input to the output. The inversion in the carry cell cancels the strong inverting buffer and the final result will be correct. The control settings for this combination are $M = E$ and $N = A$. Although notG is not used in the computation, the control bus L cannot take any value. Take a closer look at the carry cell (Figure 11). When the inputs notP and notG are both 0, the value of output notCi may become indeterminate. The easiest way to prevent this case from happening is to force notG to be 1, which implies that $L = F$. Another benefit of having notG = 1 is that it avoids discharging notCi every clock cycle thus saving a small amount of power. The control settings for the two operations we have examined are shown in Table 6. In the next lab, you will fill out the rest of the table for the remaining operations, and use it as input to your datapath simulations.

Table 6: ALU programming.

Function	notG	notP	notF	L	M	N
$R + S$	$\overline{R \wedge S}$	$\overline{R \oplus S}$	$\overline{R \oplus S}$	7	9	9
$R \vee S$	1	$R \vee S$	S	F	E	A

3.3.4 Carry simulation

You will simulate the Carry cell using both SPICE and IRSIM.

To create the input vectors for SPICE simulation, you need to setup a test bench in SUE, and test the carry cell with pwl_source's. Here are two examples of configuring a pwl_source to set the test vector:

Case 1: input vector starts with V = 0
e.g., 0 0 1 1 0 1 (step = 100ns)
then the edges should be specified as: 200ns 400ns 500ns

Case 2: input vector starts with V = vddp
e.g., 1 1 0 0 1 (step = 100ns)
then the edges should be specified as 100**ps** 200ns 400ns

In the pwl_source configuration, the initial_voltage must be set to 0 regardless of the initial value of the test vector. To start a vector with 1, you can raise the input in 100ps (as in Case 2).

To perform the SPICE simulation, you may need to customize the default SPICE configurations in SUE, e.g., the time-step and duration of the simulation. On startup, SUE automatically loads the configuration file `/cad2/mmi_local/sue/.suerc`. It defines a variable `DEFAULT_SPICE_HEADER`, which points to `default_spice_header.h` in the `/cad2/mmi_local/sue` directory. In this header file, there is the transient response command for SPICE (`.TRAN`). Its first and second arguments specify the time-step and duration of the SPICE simulation respectively. To customize these values, copy this header file to your local directory that has the test bench. Rename the header file so that it has the same base name as the test bench `.sue` file. For example, if your test bench is in `carry_spice.sue`, then the header file must be named `carry_spice.h`. In this way, the SPICE simulation for the test bench uses your local header file instead of the global one. Then you can change the local header file to suit your need.

When plotting simulations using NST, use one panel for each signal you want to show. It separates the signals nicely with no overlap. To avoid creating these panels every time you run NST, save them in a script by selecting *File →Save Panels ...* in the NST window. Then you can restore the panels by selecting *File →Restore Panels ....*

Your overall SPICE simulations output should look very similar to the IRSIM output. Your simulations must not have any unknown values. More specifically, you should omit the test case notP = notG = 0, because notCi may be at an indeterminate state. As described in the previous section, this case can never happen for arithmetic operations, and must be avoided for logic operations by setting a proper value of control bus L.

3.4 Shifter

Finally the shifter cell can be implemented efficiently using transmission gates as shown in Figure 12. Note that, in the figure, **FiPlus1 and Fi_1 are bidirectional ports.** Figure 13 shows the resulting flow of data signals for the two shift operations. The control signals of the shifter cell are shown in Table 7.

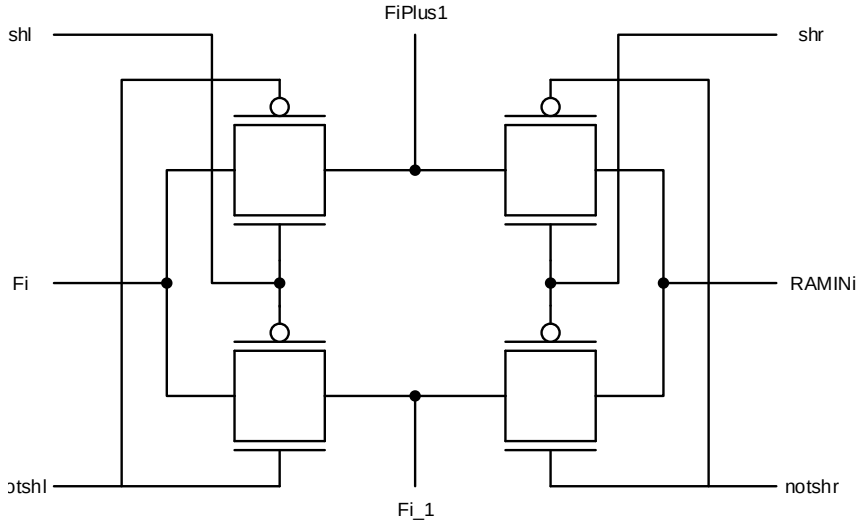
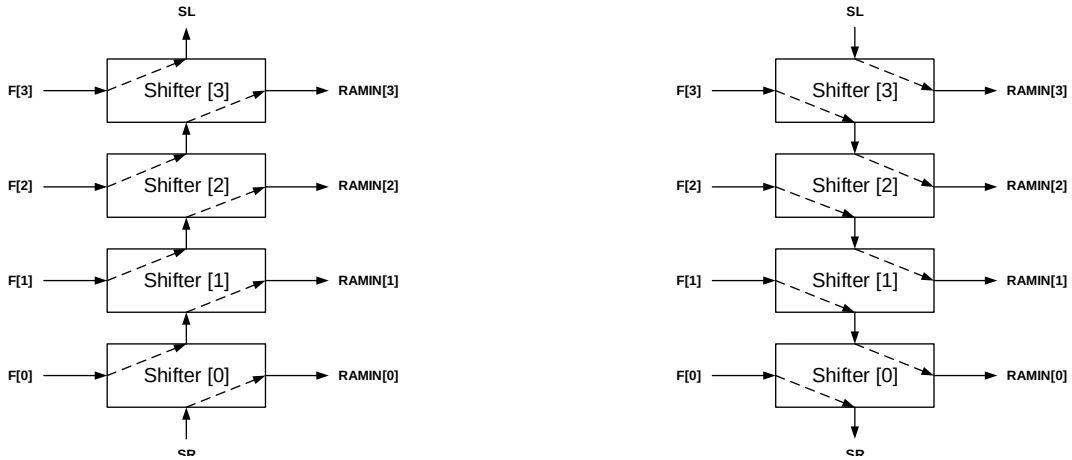


Figure 12: Schematic of a (single-bit) shifter.



(a) Shift left. (b) Shift right.

Figure 13: Flow of data signals in shift operations.

Table 7: Shifter control signals.

Receiving Cell	Control Signal Name	Purpose
Shifter	shr, notshr, shl, notshl	To shift left, set shl = 1 and shr = 0 To shift right, set shl = 0 and shr = 1 If no shifting, set shl = shr = 0
	Total: 4 signals	

4 Design Recommendations

4.1 Hierarchy

Please follow the hierarchy shown in Figure 14.

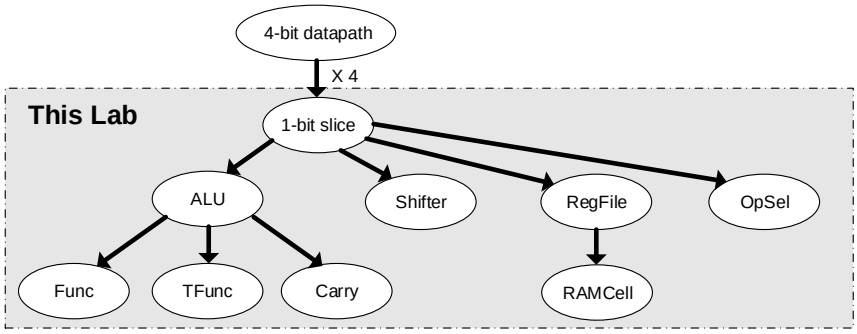


Figure 14: Complete 2901 hierarchy.

4.2 Schematic

- Do NOT use transistor width larger than 4um.
- Keep Precharge Transistors small (e.g., 1um).
- All buses must have names in square brackets "[" and "]" so that LVS will perform correctly.

4.3 Simulation

The amount of simulation you perform on the different components (ALU, Shifter, OpSel and RegFile) is left to your discretion. You are only asked to submit the simulation of the carry cell in this lab. You will, however, simulate the whole 4-bit datapath in the next lab, and thus waiting until your design is complete is probably not wise for two reasons. First, it is more difficult to debug a design composed of untested cells than with cells you are reasonably sure work correctly. Second, if you are unable to get your overall simulation to work by the due-date, you will be awarded partial credit based on the fraction of the circuit you can show worked correctly.

4.4 Layout

- Layout the cells in a row, following the order given in Figure 2 and Figure 7. **Cells must be pitch-matched, with a pitch no more than 18um.**
- In the next lab, you will stack four bit-slice layouts on top of one another to create the entire 4-bit datapath. Therefore, **control signals must pass across each bit-slice vertically** so that they are available to the neighbouring bit-slices. These signals include ARdEn[3:0], BRdEn[3:0], WriteEn[3:0], FBEn[3:0], notFBEn[3:0], ASelect, BSelect, DSelect, ZSelect, L[3:0], M[3:0], N[3:0], shl, shr, notshl, notshr, Phi1 and Phi2 (and their inverted versions). In addition to control inputs, the carry signals should also run vertically across the bit-slice and should be aligned such that the carry-out of one bit-slice feeds the carry-in of its neighbour. The same principle applies to the shifted bits (FiPlus1 and Fi_1 of the Shifter cell). **These vertical wires should run in M1.**

- The external D bus can be routed in two ways. On one hand, it can be treated as a control bus and thus should run vertically across the bit-slice. Since each bit of D (Di) is connected to one of the OpSel cells in each bit-slice, the D bus should be placed near that cell. On the other hand, the D bus can be treated as a data line and thus should run horizontally along the bit-slice. This approach requires you to leave some horizontal routing space (or a track, defined later) unused for all cells to the left of the OpSel cell that is connected to Di. You are free to to choose either approach.
- The output of the ALU cell (rightmost) is fed back to the input of the Shifter cell (leftmost). This connection may be routed either inside or outside the cells. Routing inside saves area but requires you to leave one track unused for *all* cells in the bit-slice. You are free to choose either approach.
- While you should use the PMOS-above-NMOS style for most cells, there are a couple of exceptions, namely the Func and TFunc cells. Since the Func cell has no PMOS transistors, you can use its full pitch for NMOS transistors. For the TFunc cell, you will find it easier to make two full-pitch sections side-by-side, one containing all NMOS's and the other containing all PMOS's. Using the PMOS-above-NMOS style for the TFunc cell may lead to a poorer layout.
- As in Lab 1, VDD and GND should run horizontally across the top and bottom of each bit-slice respectively. You only need one bus of each type for the bit-slice, in contrast to Lab 1 which used 2 VDD buses. **You may find it easiest to run VDD and GND in M2.**
- **Do not use Metal 3 and above.** Only use Metal 1 and Metal 2.

Here are additional hints that will make your layout more electrically or physically robust, and that are followed by industry for layout of compact and high performance circuits.

- **Transistor gates should be in the same orientation (i.e., all vertical or all horizontal).** In modern semiconductor processes, the uncertainty in mask alignment and patterning can be a measurable fraction of the minimum channel length. The patterning and alignment during processing largely determine the effective channel length and hence the transistor's electrical characteristics. Since alignment and patterning variations in different directions are independent, two identically drawn but perpendicular transistors may end up with very different characteristics. For this reason, it is best to keep all transistors in the same direction.
- **Share wells across cell boundaries.** One of the largest design rules in many processes is the well separation distance. To minimize the area while meeting this requirement, wells in subcells are often extended to the cell boundary if an adjacent cell will contain a similarly placed well.
- **For this lab, every point in a well/substrate should be within 150λ (18um) of a body contact.** All real-world processes have a design rule for the area of well or substrate that can be covered by a body contact (or a well tap). As the distance to the contact increases, the substrate/well resistance causes the bulk voltage to get further from the ideal value. This results in a higher threshold voltage (and correspondingly lower drain current) for transistors far from a body contact. In an extreme case, the result can be the forward biasing of the diode that exists between the source and the substrate/well. The default tech mmi25 is 0.24um CMOS, because the smallest transistor length (i.e. poly width) is 0.24um. λ is half of the smallest transistor length, which is 0.12um for the given process. Thus, 150λ is equal to 18um.

4.5 Floorplanning

In executing a layout of this complexity, it is highly worthwhile to develop a plan for how the various subcells are to be placed and connected, before embarking on the actual work. This process is generally

referred to as layout planning or floorplanning. You will rapidly discover during this lab that it is the routing and not the transistors themselves that determines the area of modern layout. A well-done floorplan greatly minimizes the routing complexity and area requirements and can even simplify the layout of subcells. The remainder of this section describes some strategies for developing a floorplan and creating subcells to fit within that plan.

One of the earliest and most important decisions made during floorplanning is layer usage. There are a finite number of layers available for routing, and they have unique characteristics. Poly is high resistance, but can be useful for short interconnections, especially if it turns into a gate at some point. M1 is moderately narrow, low resistance, and easily accessible, making it ideal for local interconnect. M2 is even lower resistance, but has larger space requirements due to a higher minimum width. It is also harder to access. An M2 connection to a transistor gate, for example, requires a sizable space for a contact, a via, and the required poly, M1, and M2 landing pads. Higher metal layers (commonplace in modern processes) have correspondingly larger spacing requirements and are used for even longer distance routing.

Global routing is often planned using the concept of *tracks*. Generally, global routing layers (the higher metal layers) are assigned a preferred direction that alternates every layer. Usually this direction restriction does not apply to lower metal layers (e.g., M1) because of their use in convoluted local routes. In this lab, however, it may be beneficial to route vertical control signals and local wires in M1, and horizontal wires in M2. Picture the entire layout overlaid with M2 lines running the length of the bit-slice. Each of these hypothetical metal lines is referred to as a track. You are free to pre-define the location of these horizontal tracks. Usually they are separated with minimal distance that respects the design rules. Aligning horizontal wires (in each subcell) with tracks allows efficient utilization of the M2 wiring area (within the pitch). Routing these wires in an undisciplined manner often leads to situations where one cannot find straight routing space for some horizontal wire.

Interconnection between adjacent cells is as simple as ensuring that the input and output signals of each cell go all the way to the edge of the cell at the same location on the common edge. In this way, the mere placement of the cells adjacent to each other in the next level of hierarchy connects them properly, eliminating the need for extra wiring. However, the cell layouts must meet design rules when placed adjacent to each other. The most effective way to ensure this is to keep all structures within a cell that are not connected to the adjacent cell at least half of a design rule away from the cell boundary. If the design rule is an odd number of λ, you can arbitrarily choose which sides always get a rounded-up margin. For example, assuming that the M1-M1 design rule is 3λ, you might decide that the top and left boundaries of every cell have M1 at least be 2λ, and the bottom and right sides, at least 1λ. This ensures that M1 structures in any two cells that share any edge meet the M1-M1 design rule, given that the cells are not flipped nor rotated. If the cells may be placed in arbitrary orientations (not applicable to this lab), the rounded up clearance would be required on all edges.

In more complex circuits, a subcell may be used many times. Planning the connection interface in such cases requires careful thought, especially if the cell is to be instantiated with different neighboring cells. This type of problem can be solved by putting an *option* in the cell layout, which allows the signal to be routed out of the cell boundary in multiple ways. The selection of a given option is made on the next level of the hierarchy. This may involve the extension of a metal line or the addition of a contact or via inside of the cell boundary.

The Func cell is an example of a cell that will require an option. Laying out the cell with a single fixed output port (for Y) makes it difficult to place two Func cells side by side and connect the outputs of both to the carry cell. By not bringing the output to the edge of the Func cell and making the space available to make two different connections for Y, you are creating an option for that signal. One of the possible connections

is to an M2 track so that the output of the left Func cell can be routed across the right Func cell to the carry cell. To avoid M2 conflict, the chosen track cannot be used within the Func cell.

While floorplanning greatly simplifies and accelerates hand layout, it remains an iterative process. Your floorplan will almost certainly change as you create the subcells. Since cell placement is more or less fixed for this lab, such floorplan changes are most likely to the track assignments or the signal interfaces between adjacent cells. For example, your floorplan specifies that a certain subcell's input comes from track six, but this creates a tremendous routing problem within the cell. Then the floorplan will have to change to alleviate this problem. Hopefully the changes can be made with minimal impact to other cells that have already been drawn, but this will not always be the case. **Therefore, be aware of the coarse structure of each subcell when making your original floorplan.** Do not be apprehensive about spending several hours entirely on your floorplan; the payoff in time savings will be significant.

4.6 Cell flattening

You are not required to flatten a cell as long as LVS passes and the layout is reasonably compact.

However, when you do want to flatten a cell, the sub-cells' labels will be exposed to the current level, prefixed with the subcell instance names. LVS tends to short labels with the same name in the lowest hierarchy. For example, two instantiated components will have ports connected together, even though they are not meant to connect, like OpSel_0.A and OpSel_1.A. To make LVS work properly, rename these subcell labels and remove unwanted (internal) ones in the flattened design.

Manually changing label names in MAX may be tedious for a large design. Here is a better way:

1. Save the MAX design and exit MAX.
2. Bring up an editor and change the label names in the `.max` file directly.
3. Load up MAX again with the design; the changes will be reflected.

5 Submission

The submission includes both electronic and paper parts.

5.1 Electronic submission

Submit the schematic and layout of each cell in the hierarchy (Figure 14). The file base names should be:

```
Func TFunc Carry ALU
Shifter
RAMCell RegFile
OpSel
OneBit_Slice
```

You will submit the above files with the following command:

```
submitece451s 2 OneBit_Slice.sue OneBit_Slice.max ...
```

You can check your submission with the command `submitece451s -l 2`

5.2 Paper submission

Submit the SPICE and IRSIM simulation of the Carry cell. **Please follow the same guidelines for simulation annotations described in Appendix A of the Lab 1 handout.**

Use the cover page on Blackboard and clearly indicate (a single) UG account where you do electronic submission, and **the PRA session both of you will attend in the first week of Lab 3 for in-lab marking of this report**.