ECE425: Introduction to VLSI System Design
Machine Problem 2

CP Due: 11:59pm Friday, March 25th, 2022
MP2 final Due: 11:59pm Friday, April 8th, 2022

In this MP and the next, you will create a clone of the AMD Am2901, a datapath chip first introduced in 1976. We will use a complementary static CMOS design style as much as possible to simplify the design. You still need to build parts at transistor level as well as reuse the cell library built in MP1. Beginning with block diagrams publicly released by AMD, we will finish with layout, which could be manufactured on silicon by the MOSIS academic prototype service.

The Am2901 was the heart of many minicomputers in the late 1970s. Containing a register file and a simple arithmetic unit, it often replaced a collection of smaller chips and ran (in 1979) up to 14.5MHz. One Am2901 implemented per four bits of processor word-length (so a 16-bit processor required at least four chips), and the processor would usually fill a large circuit board. The Am2901 powered the 16-bit Xerox Star, the first graphical workstation.

PART I. Logic design

Architecture

The block diagram on the page 3 is reproduced from *The Am2900 Family Data Book*, the original reference on the Am2901 from AMD. This chip is composed of:

<Main components>

- Register file, or "16x4 bit 2-port RAM"
- Q register
- ALU

<Additional components>

- Multiplexers

A number of multiplexers allow data to be routed from one component to another. The triangles represent tri-state output drivers, electrical components, which we will ignore. To help you through the diagram's spaghetti, the tables on Page 2 capture the multiplexer connections, each input listed from top to bottom (the order is insignificant) and bit 3 down to bit 0 from left to right (high-order to low-order). Each column of these tables represents a single mux. You might also need to use additional muxes, for example, inside the ALU. Finally, Page 4 (also from AMD) lists the datapath functionality controlled by the muxes in terms of the inputs you are given. You will need to generate mux control inputs from i<8:0>. For more information, see *The Am2900 Family Data Book*, in particular pages 2-002 to 2-007. It can be found on the course website's Machine Problem page.

The plethora of available data routes allowed a team of Am2901's to perform a computation with the outside assistance of extra chips, then load the result into the Q register and send it serially to slower I/O logic while performing the next computation.

The chain of components, which operate on four bits of data is called the *datapath*. Each of these comprises four single-bit components. There are also several individual "decode" blocks, which do not directly handle data. They serve to reduce the number of pins connecting the Am2901 to the outside world. (Pin count is a major factor in the cost of any microchip.) We will only implement the datapath in this MP, and leave everything else for MP3. The part of the datapath operating on a single bit is called a *bitslice*. You will design and layout a single bitslice, then combine four bitslices to form a complete datapath.

Am2901 datapath muxes and their source operands

- RAM input pins D<3:0> -> Four 3-in Muxes

	Mux to D3	Mux to D2	Mux to D1	Mux to D0	Functionality
Input 1	RAM3 input pin	F3	F2	F1	Shift ALU result right
Input 2	F3	F2	F1	F0	Load register from ALU
Input 3	F2	F1	F0	RAM0	Shift ALU result left

- Q register input pins D<3:0> -> Four 3-in Muxes

	Mux to D3	Mux to D2	Mux to D1	Mux to D0	Functionality
Input 1	Q3 input pin	Q3 from register	Q2	Q1	Shift Q right
Input 2	F3	F2	F1	F0	Load Q from ALU
Input 3	Q2	Q1	Q0	Q0 input pin	Shift Q left

- ALU input pins R<3:0> -> Four 3-in Muxes

	Mux to R3	Mux to R2	Mux to R1	Mux to R0
Input 1	D3	D2	D1	D0
Input 2	RAM output A3	RAM output A2	RAM output A1	RAM output A0
Input 3	Constant zero	Constant zero	Constant zero	Constant zero

Note: 3rd input to the mux is not shown in the diagram, but some ALU operation's operand is 0.

- ALU input pins S<3:0> -> Four 4-in Muxes

	Mux to D3	Mux to D2	Mux to D1	Mux to D0
Input 1	RAM output A3	RAM output A2	RAM output A1	RAM output A0
Input 2	RAM output B3	RAM output B2	RAM output B1	RAM output B0
Input 3	Q register out Q3	Q register out Q2	Q register out Q1	Q register out Q0
Input 4	Constant zero	Constant zero	Constant zero	Constant zero

Note: 4th input to the mux is not shown in the diagram, but some ALU operation's operand is 0.

- Output pins Y<3:0> -> Four 2-in Muxes

	Mux to R3	Mux to R2	Mux to R1	Mux to R0
Input 1	RAM output A3	RAM output A2	RAM output A1	RAM output A0
Input 2	ALU output F3	ALU output F2	ALU output F1	ALU output F0

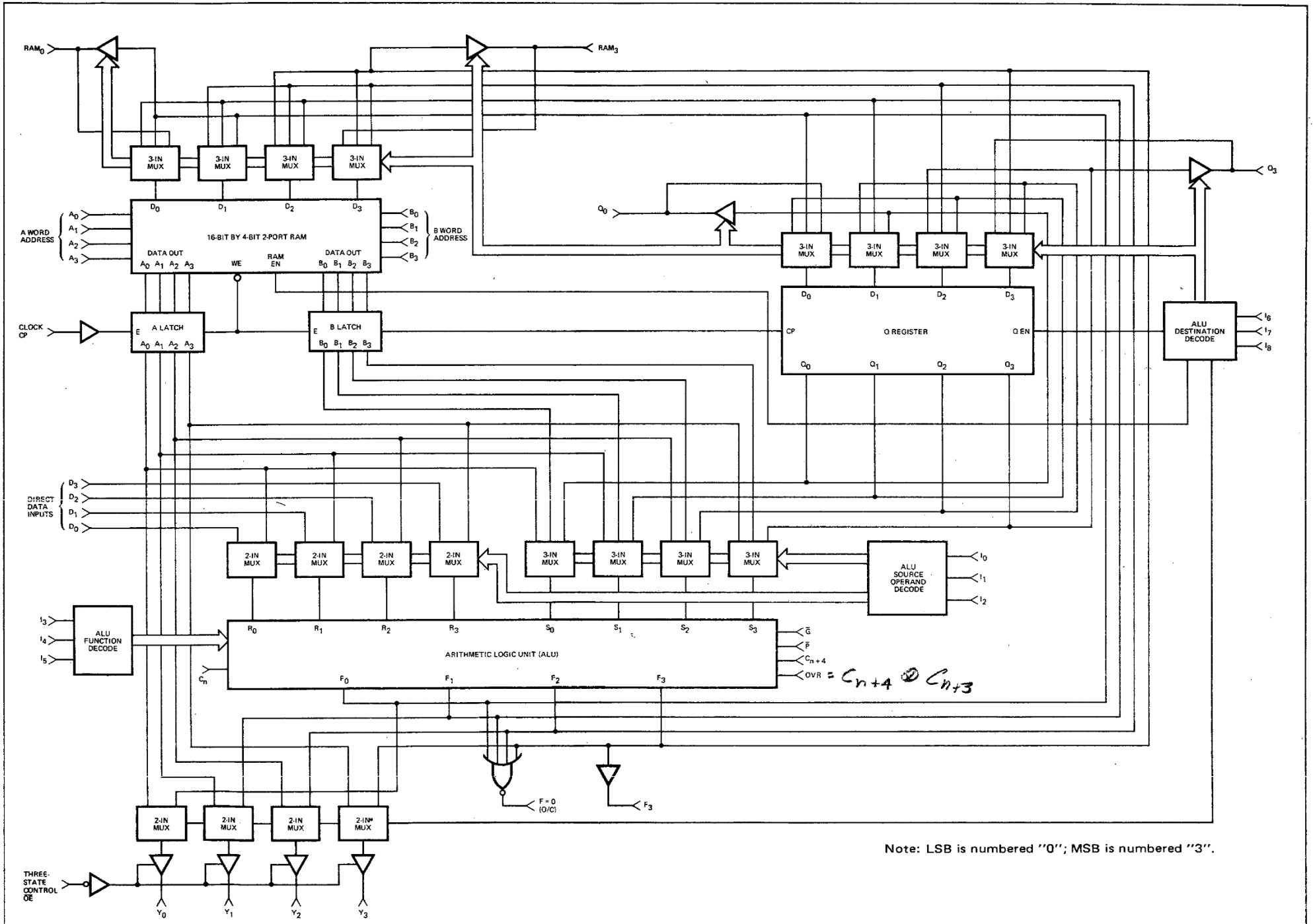


Figure 1. Detailed Am2901B Microprocessor Block Diagram.

MPR-005

Mnemonic	MICRO CODE I_2	I_1	I_0	Octal Code	ALU SOURCE OPERANDS R	S
AQ	L	L	L	0	A	Q
AB	L	L	H	1	A	B
ZQ	L	H	L	2	O	Q
ZB	L	H	H	3	O	B
ZA	H	L	L	4	O	A
DA	H	L	H	5	D	A
DQ	H	H	L	6	D	Q
DZ	H	H	H	7	D	O

Figure 2. ALU Source Operand Control.

Mnemonic	MICRO CODE I_5	I_4	I_3	Octal Code	ALU Function	SYMBOL
ADD	L	L	L	0	R Plus S	R + S
SUBR	L	L	H	1	S Minus R	S − R
SUBS	L	H	L	2	R Minus S	R − S
OR	L	H	H	3	R OR S	R ∨ S
AND	H	L	L	4	R AND S	R ∧ S
NOTRS	H	L	H	5	$\overline{R}$ AND S	$\overline{R} \wedge S$
EXOR	H	H	L	6	R EX OR S	R ∀ S
EXNOR	H	H	H	7	R EX-NOR S	$\overline{R \forall S}$

Figure 3. ALU Function Control.

Mnemonic	MICRO CODE I_8	I_7	I_6	Octal Code	RAM FUNCTION Shift	Load	Q-REG. FUNCTION Shift	Load	Y OUTPUT	RAM SHIFTER RAM_0	RAM_3	Q SHIFTER Q_0	Q_3
QREG	L	L	L	0	X	NONE	NONE	F → Q	F	X	X	X	X
NOP	L	L	H	1	X	NONE	X	NONE	F	X	X	X	X
RAMA	L	H	L	2	NONE	F → B	X	NONE	A	X	X	X	X
RAMF	L	H	H	3	NONE	F → B	X	NONE	F	X	X	X	X
RAMQD	H	L	L	4	DOWN	F/2 → B	DOWN	Q/2 → Q	F	F_0	IN_3	Q_0	IN_3
RAMD	H	L	H	5	DOWN	F/2 → B	X	NONE	F	F_0	IN_3	Q_0	X
RAMQU	H	H	L	6	UP	2F → B	UP	2Q → Q	F	IN_0	F_3	IN_0	Q_3
RAMU	H	H	H	7	UP	2F → B	X	NONE	F	IN_0	F_3	X	Q_3

✱ DON'T USE □ DISABLE $\overline{FEN}$ Δ ENABLE $\overline{FEN}$

X = Don't care. Electrically, the shift pin is a TTL input internally connected to a three-state output which is in the high-impedance state

B = Register Addressed by B inputs.

UP is toward MSB, DOWN is toward LSB.

Figure 4. ALU Destination Control.

OCTAL I_{210} / I_{543}	ALU Function \ ALU Source	0 A, Q	1 A, B	2 O, Q	3 O, B	4 O, A	5 D, A	6 D, Q	7 D, O
0	C_n = L R Plus S	A+Q	A+B	Q	B	A	D+A	D+Q	D
	C_n = H	A+Q+1	A+B+1	Q+1	B+1	A+1	D+A+1	D+Q+1	D+1
1	C_n = L S Minus R	Q−A−1	B−A−1	Q−1	B−1	A−1	A−D−1	Q−D−1	−D−1
	C_n = H	Q−A	B−A	Q	B	A	A−D	Q−D	−D
2	C_n = L R Minus S	A−Q−1	A−B−1	−Q−1	−B−1	−A−1	D−A−1	D−Q−1	D−1
	C_n = H	A−Q	A−B	−Q	−B	−A	D−A	D−Q	D
3	R OR S	A ∨ Q	A ∨ B	Q	B	A	D ∨ A	D ∨ Q	D
4	R AND S	A ∧ Q	A ∧ B	0	0	0	D ∧ A	D ∧ Q	0
5	$\overline{R}$ AND S	$\overline{A} \wedge Q$	$\overline{A} \wedge B$	Q	B	A	$\overline{D} \wedge A$	$\overline{D} \wedge Q$	0
6	R EX-OR S	A ∀ Q	A ∀ B	Q	B	A	D ∀ A	D ∀ Q	D
7	R EX-NOR S	$\overline{A \forall Q}$	$\overline{A \forall B}$	$\overline{Q}$	$\overline{B}$	$\overline{A}$	$\overline{D \forall A}$	$\overline{D \forall Q}$	$\overline{D}$

+ = Plus; − = Minus; ∨ = OR; ∧ = AND; ∀ = EX-OR

Figure 5. Source Operand and ALU Function Matrix.

Preparation

First install a "starter kit" in your work directory(ece425.work). It contains schematics and Verilog files:

```
tar xf /class/ece425/ece425mp2_fall17.tar
```

Initialize cadence by running the following commands

- module load ece425-sp21
- virtuoso &

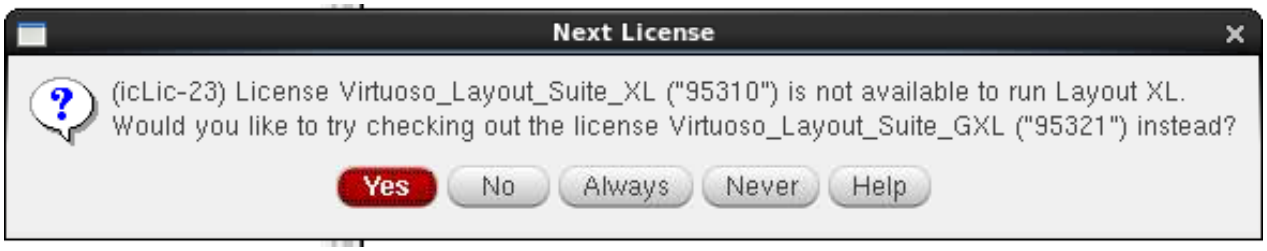
Then, open the library path editor and add the starter kit library *ece425mp2* along with a digital logic schematic library from North Carolina State University (NCSU):

Library name	*Path*
ece425mp2	ece425mp2

Do not use the XOR or XNOR gates from this library, because they lack schematic views. Use ece425mp1 instead. However, **never** use other basic logic symbols from ece425mp1 (nand2, nor2, etc.) because this will confuse Cadence later. You can use your ece425mp1 cells with NCSU_Digital_Parts symbols. Furthermore, if you are to use the *tx_gate*, only use the one from ece425mp2 library. Do not create tx_gate yourself, or use the one from NCSU library. The rules are summarized below:

- Use XOR, XNOR gates from ece425mp1 library
- Use basic logic symbols (e.g. NAND2, NOR2, …) from NCSU_Digital_Parts
- Use TX_GATE from ece425mp2 library

Briefly go through the provided schematics to have a rough idea about the MP. Only partial schematic is provided. If you encounter the following pop up window while opening any schematic, select "Always".



Complementary logic schematics

You can begin your project by adding the missing logic structures. Start with the muxes. Consider using multiple stages of simple NANDs or NORs, or alternatively larger complex logic gates like AOIs. (See section 9.2.1 of Weste & Harris.) The goal is to use the least number of transistors.

We will introduce several basic rules that will keep your circuit design "reasonable" in physical considerations, beyond ensuring the basic Boolean logic functionality. Two will affect your logic gates and one is common sense:

Rule 1. *Complementary static CMOS*: Your logic **must** be composed of logic gates, each consisting of an NMOS network connected to ground and a PMOS network connected to the supply voltage. At any time, either the NMOS or else the PMOS network must be conducting, no matter what inputs are given to the gate. (**No transmission gates**!)
Rule 2. *Fan-in*: No path from a gate output to power or ground may be more than 3 transistors long. This improves reliability by limiting the body effect near the output node, and also ensures you do not add long chains of bulky transistors.
Rule 3. *Combinational logic only*: do not introduce feedback loops besides the ones given.

Please place each logic gate (according to the definition of Rule 1) in its own cell. Therefore, gate cells will contain only transistors, and all other cells will contain only sub-cells but no individual transistors. You may need to add input pins to your cells, for example, mux select inputs. You will write Verilog code to generate these inputs once the schematics are finished. **Do not implement logic to generate a function when you could simply add an input pin!** For example, if you need the complementary signal of pin *p*, instead of adding an inverter in your datapath, you can also choose to add an input pin *p'*. In this way, you can greatly reduce the number of gates in your datapath, which means smaller area for your layout.

Memory schematics

Although complementary CMOS flip-flops and memory arrays are possible, they are never used. Simpler alternatives use *ratioed logic*: rather than requiring exactly one path to power or ground from the data-storage node, allow either one or two. Consider this circuit:

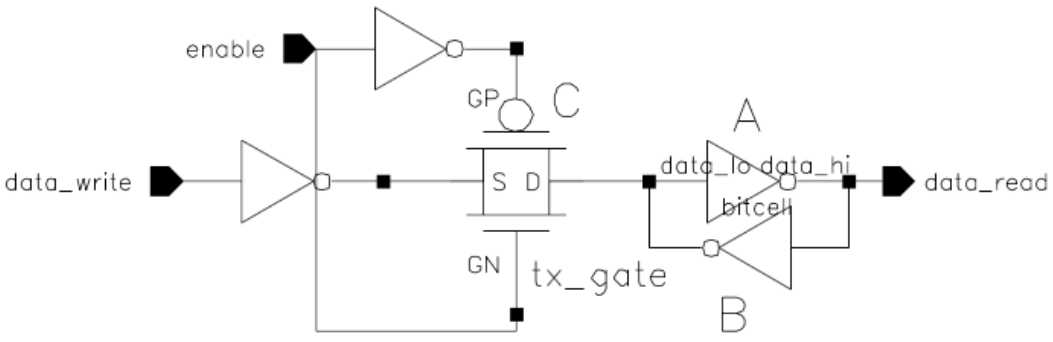


Figure 6. Example schematic of using bitcell

Inverter *A* is always driven by inverter *B*. But if the pass-gate *C* is turned on, it will be driven by both *B* and *C*. This may result in a direct path from power to ground as *B* and *C* form a voltage divider. If *C* has much less resistance than *B*, however, the voltage divider will always drive *A*'s input close to Vdd or ground. Inverter *A*, in turn, sends a perfect Vdd-or-ground (*rail-to-rail*) output to *B*, which also flips, shutting off the power-to-ground short circuit. The cross-coupled inverter structure, which we will call a *bitcell*, is at the heart of "static" computer memories.

Use only the starter kit *bitcell* cell – do not attempt to create your own. Furthermore there is an extra rule to follow within the *regfile* and *latch*:

Rule 4. *Drive ratio*: the bitcell must be driven with at least 0.72 microns effective width to perform a write. To simplify our MP, we have done the sizing for you. So what you need to do is the following. First, don't delete the inverters already existing in the given schematic and also don't change the sizes of them. Second, when you add an inverter to your design, use the smallest inverter, with Wn = 0.36 um and Wp = 0.72um.

Timing is important in any system containing feedback. The *bitcell* cell contains a Verilog delay command that prevents the RAM input from racing through the entire datapath when the transmission gates turn on and off. To take advantage of the delay, use one terminal of the bitcell as an input and the other as an output. You are given more detailed schematics for the RAM than for other blocks, and you should understand them.

Please use the *tx_gate* included in your ece425mp2 library, as it contains a Verilog workaround. Don't create your own tx_gate cells. Also don't modify the schematic of *tx_gate*, which is a completed schematic, although it seems a little weird to you. **Using a single instance of NCSU_Digital_Parts *tx_gate* will break your Verilog simulation.**

As for the clock inputs of RAM and also Q register, the following is quoted from The *Am2900 Family Data Book.* "*The Q register and register stack outputs change on the clock LOW-to-HIGH transition. The clock LOW time is internally the write enable to the 16x4 RAM which compromises the master latches of the register stack. While the clock is LOW, the slave latches on the RAM outputs are closed, storing the data previously on the RAM outputs. This allows synchronous master-slave operation of the register stack.*" To put it simply, the master latches of both RAM and Q register should be open at clock LOW, while the slave latches should be open at clock HIGH.

Carry chain

We use a carry chain, which violates a couple rules to improve area and performance. It is similar to the Manchester carry chain, except it replaces the dynamic clock input with a static "kill" input. See Fig. W11.1 and section 11.2.2.4 in Weste & Harris Web (www.cmosvlsi.com). Each stage of the chain works as follows:

1. Carry-out is always connected to Vdd, gnd, or carry-in (static pass-transistor logic)
2. Carry-out connects to Vdd if a=b=1 (generate = a AND b)
3. Carry-out connects to gnd if a=b=0 (kill = a NOR b)
4. Otherwise, carry-out connects to carry-in through a transmission gate (propagate = a XOR b)

Two of our style rules are broken, so you can ignore them as a special case:

1. The chain is four transmission gates long. Including one transistor driving the carry-in pin outside the chip, this is a fan-in of five. Our rule limits fan-in to three.
2. The generate, kill, propagate, and negative propagate signals all go into the carry stage. A glitch or transient short circuit may occur, for example, if the propagate XOR turns off slowly while the generate NAND turns on quickly. Usually we avoid such situations by using fully complementary static CMOS, as explained earlier. The performance gain is worth the small cost in power in this case.

RTL code

Besides the datapath structures, Figure 1 also shows other components such as those labeled "decode." Although not all are related to each other, these blocks are collectively known as the *control unit*. Since performance is generally measured in terms of the time between operands going into the datapath and results coming out, and since these structures will only appear once in the final layout (contrast with the bitslice being copied four times), we will not manually optimize the control unit. You need only to specify its functionality in Verilog. This Verilog will automatically be converted to working layout in MP3.

The style of Verilog you will write is called *RTL code*. This stands for "register transfer language." Although our code contains no registers, it is a common term referring to anything that describes desired functionality regardless of the exact implementation.

The datapath will be most efficient if kept small, using the minimum number of transistors. Wherever possible, you are encouraged to keep functionality out of the datapath and inside the control unit. In particular, it is wasteful to decode every instruction word I<8:0> using identical transistors in each bitslice of the datapath.

Our RTL is split into two files:

- *controller.v*: describes the control unit
- *Am2901.v*: integrates the datapath with the control unit

Cadence will generate Verilog for the datapath, as done for your cells in MP1. You can copy the *inv_s* signal and use it as a template when adding your own control signals. Be sure to follow all these steps:

1. Add pins to the *datapath* cell in Cadence. You should only add inputs.
2. Propagate the pins and wires down through the hierarchy as appropriate, updating schematics and symbols.
   a. Check out the trick in the given *datapath* schematic using symmetrical top and bottom pins for wires connecting to all the bitslices in parallel, such as *cp*. If you do copy a given pin, note that the new top and bottom pins must both be renamed.
3. Add the pins to the list in the module declaration at the top of controller.v. **Use the same name for the pin in both the controller and the datapath.**
4. Declare the pins as "output" below the module declaration in controller.v.
5. Add logic code to assign the desired functions to the new pins.

6. In Am2901.v, add a wire to carry the signal from the controller to the datapath.
7. Connect the wires to the controller and the datapath using inv_s as an example. Note that the controller and datapath instances both contain ".inv_s(inv_s)".

We do not expect you to become a Verilog expert for this MP. Describing your decode logic should not be too hard given the examples in Am2901.v and prior knowledge of VHDL from ECE385. Note that Verilog offers C-like bitshift operators "<<" and ">>".

Logic verification

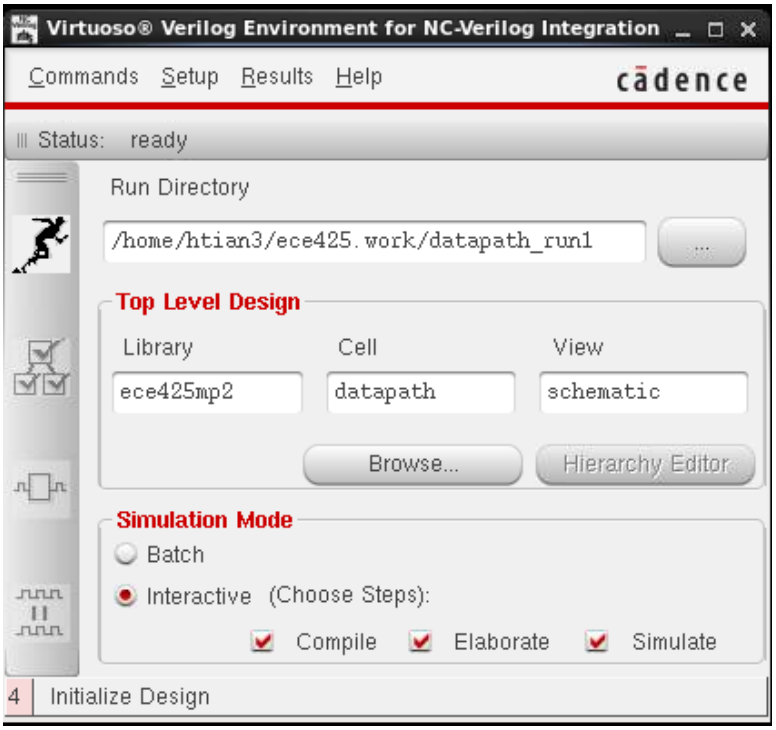
We will use NC-Verilog to run an Am2901 program through our schematics. It is important to check that schematics do the right thing before moving onto layout!

There are three major components to our Verilog code:

1. "Netlist" code: generated from your schematics
2. "RTL" code: written by hand and expresses functionality we "know will happen"
3. "Stimulus" code: drives the input pins according to a test program

First launch NC-Verilog as follows:

1. Select "*Launch=>Plugins=>Simulation=>NC-Verilog*", a window like below should pop up. In the Library, fill in "ece425mp2", in Cell, fill in "datapath", and fill in "schematic" in View. Then, click the icon with a running man to initialize the design. A folder named "*datapath_run1*" should be automatically created under your working directory (ece425.work/).



2. In MP1 we used only netlist and stimulus files. To add the RTL files to your setup, execute the following commands in your working directory (ece425.work):

```
cp ece425mp2/Am2901bench.v  datapath_run1/testfixture.template
cp ece425mp2/Am2901test.v  datapath_run1/testfixture.verilog
```

3. To disable overwriting the stimulus and test bench when netlisting the schematics, in the Verilog integration window, select *Setup =>Netlist* and deselect *Generate Verilog Test Fixture Template*. Click OK. You should only have to perform these steps once.

If the following window pops up complaining "Top Scope Mismatch", just click "close". It should not affect your simulation results.

You will do comparison with the golden file *"~/class/ece425/mp2/datapath_gold/datapath_gold.trn*". Your implementation will glitch differently from everyone else's, so we only want to compare at the falling clock edges:

1. Choose "*Setup => Simulation Compare...*" from the NC-Verilog window
2. Specify the golden file *"~/class/ece425/mp2/datapath_gold/datapath_gold.trn*" and compare file "*datapath_run1/shm.db/shm.db.trn*", or where you specify as the run directory. Proceed with your comparison.
3. The simulation comparison result is in compare.out. Don't worry if you find 'cout_high' and 'g_lo' mismatch. There may be some difference between designs, but should not be huge. But **you need to make sure your output 'y' matches**.

Debugging latches will be easier if you select *Edit => Preferences* in SimVision, open the preferences for Verilog, and select *Show Strength* and *Show colors by strength*. Signals driven by the bitcell will then appear light blue, so you can see when a latch is being read versus written.

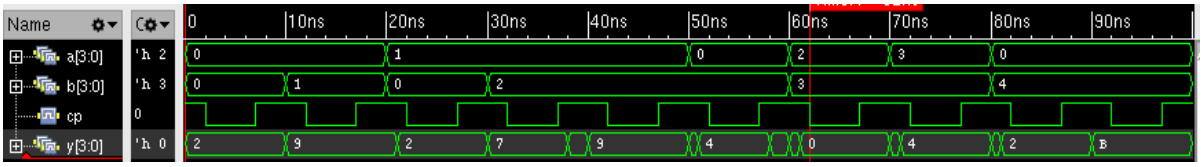
Besides only caring about the falling edges, we also will ignore mismatches on the carry signal when we're not performing addition or subtraction. (The AMD book specifies exactly what the carry signals do under all conditions, but we don't need to be that detailed. You might get different results depending which operands you negate to get each logical function.)

Some Hints:

1. Every time you add an input pin to your datapath, you add a control signal between the controller and the datapath, so you have to modify several verilog files to make this happen. To help you have

a better understanding about how this works, we added several example signals for you, like "inv_s" ('s' complement when it is equal to 1) and "reg_wr"(regfile write enable). Just follow these examples and add your own signals.

2. Most of the schematics we provided are not complete and you need to add more connections and/or even more pins to the schematic. After adding pins, when you "*check and save*" the schematic, you will get the error message saying that the schematic has pins not existing in the symbol view. Then, what you need to do is to "modify" your symbol instead of "replacing" your symbol, because the symbol is already used in the higher-level schematic. If you replace the symbol, then the pin positions will be totally different and you will mess up the higher-level schematics.
3. Your waveform should match the golden file. However, after the comparison, if you believe some errors are due to the problem of the software setup, there is another way to justify your waveform: output y[3:0] should have stable value at all cp's falling edges, which are (**2, 9, 2, 7, 9, 4, 0, 4, 2, B**) in sequence. A sample output is given as follows. Note that different implementations will have different glitches.



4. If you see wired behaviors of your simulation, such as "*ncsim: Path element could not be found: "datapath"*"", please delete the folder "datapath_run1", and perform all steps for logic simulation and run it again.
5. More tips and solutions can be found in FAQ document and Piazza.

PART II. Layout

In MP1, you have implemented an 8-bit adder. Likewise, our *bitslice* will be implemented as rows of cells with data "flowing" from left to right. The height of this structure is the *bit pitch*, and it is a major factor determining the area of the design. (The other factor is the length of the row.)

You will be graded based on the correctness as well as the area of your layout on total area. Area is a good indicator of power and general efficiency. A tight final layout requires prior planning and proper prioritization. Try to estimate what component will take the most area in the complete design, and find the most efficient implementation of that piece. If you can, reduce the bit pitch. Then try to fit your other cells to that bit pitch, even if some additional area is wasted (as long as it's less than the savings from making the larger structure smaller). Before integrating your cells into a final layout, sketch how they should fit together using their known sizes. The bit pitch may be brought as low as 10um before the memory is most compact.

Please layout your *bitslice* as nearly solid rows of cells. You cannot save total area by moving a few cells to the bottom or top as "extras." It is okay to route some wires above or below the rows of cells, adding a little height to the layout. This takes less area than adding a whole partly empty row!

To give everyone a fair head start, here are some guidelines:

1. The memory array will take up a lot of space.
2. The smallest layout is usually one, which packs the control wires tightest.
3. Control wires should go on metal 3, data operands on metal 2, and metal 1 should be for transistor terminals, with few long-distance signal routes. Keep **metal 3 routes vertical** and **metal 2 horizontal**.
4. Good layouts devote less space to wires and more to transistors.
5. Get a rough draft that works before trying to optimize.
6. Reduce, reuse, and recycle the cells you have already implemented.
7. To minimize area, if you could add an input pin instead of adding a gate, do it.

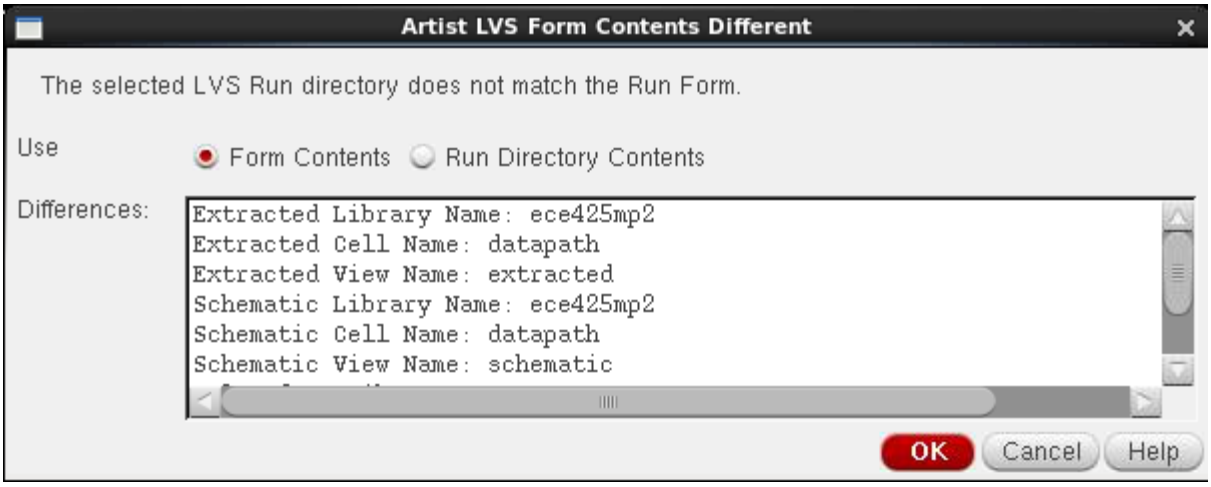
In addition to the guidelines and DRC rules, there are some class layout rules that **you will be penalized (lose points) for breaking:**

1. ***No connect by name***. You must connect power and ground wires together in the layout. Likewise for all the control signals. (This only applies to the top-level datapath cell – you will lose too much space if you tie Vdd and gnd in every register file cell!)
2. ***Poly routes***. Do not use polysilicon routes longer than 10um, measured from contact to terminus. Every polysilicon polygon should contain a transistor gate. Metal contacting poly should only connect to transistor source/drains. Transistor gates should be vertical.
3. ***Verified layout for every cell***. Layout the smallest units first, check it, and then move on. The smoothest design flow finishes first. Try not to find all the little bugs in the big circuit.
4. ***Body contacts in every cell***. Find room and add a ptap and an ntap in every cell.

Note since we have never talked about sizing the transistors, just keep the pmos width to 0.72 microns and the nmos width to 0.36 microns. And when doing LVS, uncheck "*Compare FET parameters*".

Hints:

1. If a window complaining "*Artist LVS Form Content Different*" appears, just choose "*Form Contents*" and fill in the correct value in the form.



2. More tips and solutions can be found in FAQ document and Canvas Discussion Page.

Submission Guidelines:

1. Similar to MP0 and MP1, you will be submitting the report in single merged PDF file. Use the commands introduced in former MPs.
2. The correctness of your datapath schematic will be considered in the MP2 final submission. Please include your comparison output or simulation waveform for your datapath schematic in your final report.
3. We will penalize violation of those "rules" mentioned in the document. However, don't worry about too much about the other design suggestions given in the document. As long as you follow the suggestion in most part of your design, you will be fine.
4. Try your best to optimize your area, designs larger than 20,000 units will be penalized. Full credit on MP2 are only given to correct designs with area smaller than 20,000 units.

MP2 Check Point (5 points): March 25

1. Printouts of your schematics
    a. *MUX*
    b. *latch*
    c. *ALU*
    d. *logic* part of ALU
2. List the *eight* ALU functions and explain how you implemented each of them.
3. Printouts of your top-level schematics.
    a. *bitslice*
    b. *datapath*
4. Simulation waveform from SimVision showing **only the pins** of the *top2901* instance executing the given test program
5. Simulation comparison result of the *top2901* instance (in compare.out: select *File => Save as…*, then convert the resulting file to PDF)

MP2 Final Report (20 points): April 8

6. Screenshot of *datapath* layout with width and height measurements using ruler. Calculate the area of your final layout and **write down on the first page of your submission!**
   (Ruler measurements won't show using print file option, so this is the only exception.)
   Penalty: Area larger than 20000 units will be penalized by 3 points.
7. Printouts of the following cell layouts, with only one level of hierarchy per layout. From *File->Print,* click *Display Options*, set *"Show Name of"* to "*Master"*, as explained in MP1, and under *Display Levels*, set *"from"* to 0 and *"to"* to 1.
    a. *ADD*
    b. *ALU*
    c. *MUX2-1*
    d. *MUX3-1*
    e. *MUX4-1* (If you didn't use MUX4-1, print your Q-register)
    f. *regbit*
    g. *datapath*
8. LVS result for the top-level *datapath* (with "Join Nets with Same Name" turned off and strict NCSU comparison rules, but unchecked "Compare FET parameters").
9. So far, you have finished the hand-made part of you AM2901 microprocessor. Are there any difficulties you have encountered in your design and layout? List three most important ones and simply describe what they are and how you solved them.
10. Briefly answer the following questions:
    a. What is the distance between the *vdd* and *gnd* in your layout? Why do you set this value?
    b. Some gates are sized to be larger than the minimum size (e.g., I55 and I56 in the *regfile* or I0 and I1 in the *logic*). Take one example to explain why it has to be set larger than the minimum size? (*Hint*: different examples may have different reasons.)
11. Please also submit all your design files in a zipped folder (such that anyone can evaluate your work in cadence).