Lecture Summary

Contents

1 Organization and Introduction 2
2 Binary Numbers 2
3 EE Perspective 2
4 Combinational Circuits Theory 3
5 Combinational Circuits Design 4
6 FPGA Systems, Experimental Board 5
7 Verilog Combinational Circuits 5
8 Sequential Circuits Design 6
9 Verilog Sequential Circuits 7
10 Sequential Circuits: Timing 7
11 Arithmetic Circuits 8
12 Number Systems 9
14 Verilog Testbenches 10
15 MIPS Assembly 10
16 Memory Systems 11
17 Microarchitecture: Single Cycle Architecture 12
18 MIPS Programming 13
19 Cache Systems 14
20 IO Systems 15
21 Microarchitecture: Multi-cycle Architecture 15
22 Microarchitecture: Pipelined Architectures 16
23 Advanced Processors 17
Review 17

1 Organization and Introduction

- The art of managing complexity: abstraction, disciple, hierarchy, modularity, regularity
- Bit = **B**inary dig**it**

	Abstraction Levels	Examples
This Course	**Application Software**	Programs
	Operating Systems	Device drivers
	Architecture	Instructions, Registers
	Micro architecture	Datapath, Controllers
	Logic	Adders, Memories
	Digital Circuits	AND gates, NOT gates
	Analog Circuits	Amplifiers
	Devices	Transistors, Diodes
	Physics	Electrons

2 Binary Numbers

- While in the decimal system every position i in a number (beginning from position 0) is multiplied by 10^i, in the binary system every digit is multiplied by 2^i.
- Up to 2^{15}:
  $1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768$; $2^{10} \approx 10^3, 2^{20} \approx 10^6, 2^{30} \approx 10^9$
- Convert decimal to binary (roughly): find next larger power of two, set that position to 1 (if your number is $\leq$ than that power, else 0), calculate the difference and repeat
- Convert binary to decimal: multiply every position by 2^i
- Hexadecimal numbers have 16 different digits (base 16), ranging from 0 to F and are used express long binary numbers in a shorter format; conversion is done using 16^i.
- $\underbrace{1}_{\substack{\text{most}\\\text{significant}\\\text{bit}\\\text{MSB}}}\ 001011\ \underbrace{0}_{\substack{\text{least}\\\text{significant}\\\text{bit}\\\text{LSB}}}\ ,\overbrace{1001\underbrace{0110}_{\text{nibble}}}^{\text{byte}},\ \underbrace{CE}_{\substack{\text{most}\\\text{significant}\\\text{byte}}}\ BF9A\ \underbrace{D7}_{\substack{\text{least}\\\text{significant}\\\text{byte}}}$
- Addition in base 2 works just like in base 10 using carries[1]
- Overflow: since digital system operate on affixed number of bits, it can happen the result is too big to fit into the available bits
- Range of a N digit number: $[0, 10^N - 1], [0, 2^N - 1]$
- Signed binary numbers: sign/magnitude numbers, one's complement numbers, two's complement numbers
- **Sign/magnitude numbers**: 1 sign bit, $N-1$ magnitude bits, the sign bit is the MSB, 0 being positive, 1 being negative; range $[-(2^{N-1}-1), 2^{N-1}-1]$; problems: addition doesn't work, two representations of 0 (± 0)
- **One's complement**: A negative number is formed by reverse the bits of the positive number (MSB still indicates the sign); range $\left[-\left(2^{(N-1)}-1\right), 2^{N-1}-1\right]$; addition is done using binary addition with end-around carry i.e. if there is a carry at the MSB of the sum, this bit must be added to the LSB
- **Two's complement (commonly used)**: addition works, single representation for 0, no end-around carry; a negative number is formed by reversing the bits of the positive number and adding 1 (MSB still indicates the sign); range $[-2^{N-1}, 2^{N-1}-1]$; example: flip the sign of $8_{10} = 11000_2$: invert the bits $\rightarrow 00111_2$, add one $\rightarrow 01000_2$
- Increasing bit width (assume $M > N$) from N to M: either sign extension or zero extension
- Sign extensions: sign bit copied into most significant bits; number value remains the same, correct result for two's complement; $\underbrace{0011}_{N=4} \rightarrow \underbrace{\mathbf{0000}0011}_{M=8}$
- Zero extension: zeroes are copied into the most significant bits; value will change for negative numbers; $-5_{10} = 1011_2 \rightarrow \mathbf{0000}1011_2 = 11_{10}$

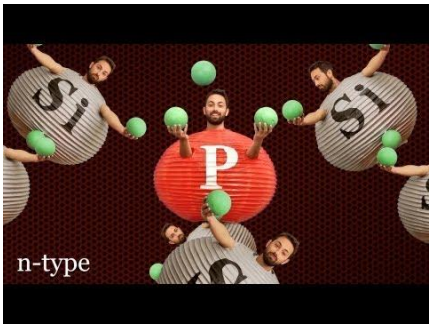
3 EE Perspective

- The goal of circuit design is to optimize: area, speed/throughput, power/energy, design time
- Frank's Principles: be lazy, ask why, engineering is not a religion, KISS

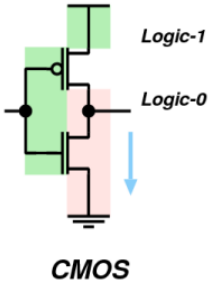
Digital Circuits	AND gates, NOT gates
Analog Circuits	Amplifiers
Devices	Transistors, Diodes
Physics	Electrons

[1] "Behalte", "Übertrag"

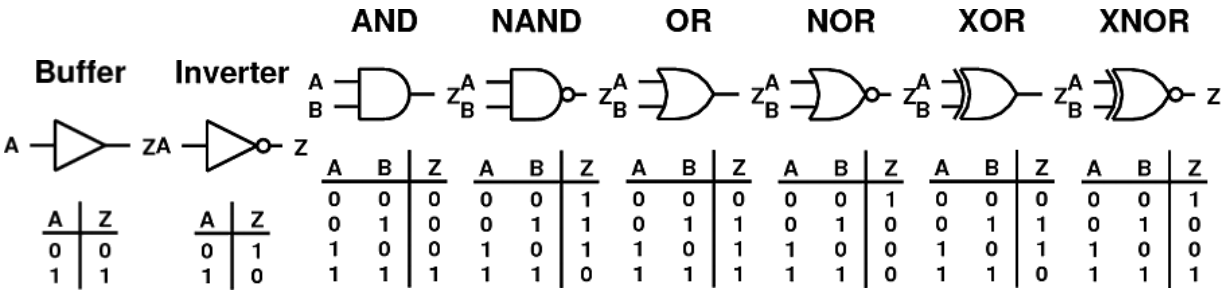
- Building blocks for microchips: conductors (Al, Cu), insulators (SiO_2, air), semiconductors (Si, Ge)[2]
- Semiconductor doping: N-type: extra electrons, negatively charged, e.g. P; P-type: missing electrons, positively charged, e.g. B[3]
- P and N regions can be combined and can be used to build diodes, transistors, …
- pMOS is a P type transistor, nMOS an N type transistors; combined they are a CMOS
- aside: explanation of transistors[4]



- **CMOS**: has no input current, has no current when output is at logic levels, current is only needed for switching, electrical properites determined directly gemoetry (double the size, double the current), very simple to manufacture
- CMOS gate structure: the general form used to construct any inverting logic gate (NOT, NAND, NOR): input $\twoheadrightarrow \begin{Bmatrix} \text{pMOS, pull} - \text{up network} \\ \text{nMOS, pull} - \text{down network} \end{Bmatrix} \rightarrow$ output, where the network may consist of transistors in series or in parallel

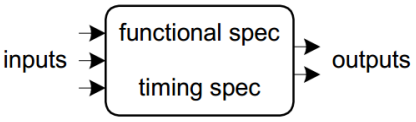


- Common logic gates:



- Multiple-input logic gates and compound gates exist, too
- Logic levels: define a range of voltages to represent 1 and 0 and define different ranges for inputs and outputs and allow for noise in the system (e.g. resistance in a long wire)
- **Moore's Law**: "The number of transistors that can be manufactured doubles roughly every 18 months"; one was able to keep it by manufacturing smaller structures, developing materials with better properties, optimizing manufacturing steps and new technologies
- Power consumption: dynamic and static power consumption; dynamic: power to charge transistor gate capacitances; static: power consumed when no gates are switching

4 Combinational Circuits Theory



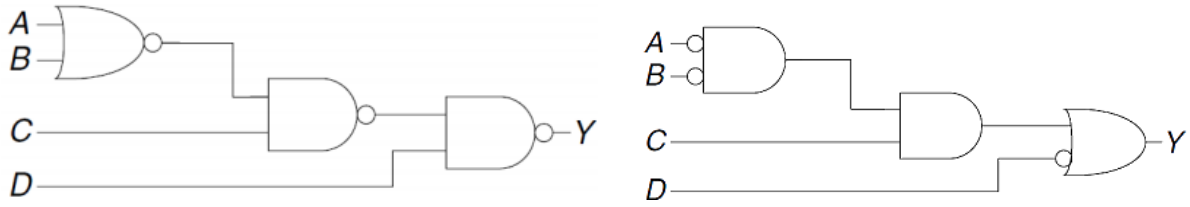
- To count the nodes (wires) in a circuit look at the outputs of every circuit element and the inputs of the entire circuit
- **Combinational (combinatorial) logic**: memory less, outputs determined by current values of inputs
- **Sequential logic**: has memory, outputs determined by previous and current values of inputs
- Rules of combinational composition: every circuit element is itself combination; every node is either designated as an input or connects to exactly one output terminal of a circuit element; the circuit contains no cyclic paths: every path through the circuits visits each node at most once
- Boolean algebra[5]: the axioms and theorems are dual i.e. if ANDs and ORs are interchanged and 0s and 1s are interchanged, they stay correct

[2] Aluminum, copper, glass, Silicon, Germanium
[3] Phosphorus, boron
[4] https://www.youtube.com/watch?v=IcrBqCFLHIY
[5] Again, this is also part of "Diskrete Mathematik" where it is dealt with more in depth

	Axiom		Dual	Name		Theorem		Dual	Name
A1	$B = 0$ if $B \neq 1$	A1′	$B = 1$ if $B \neq 0$	Binary field	T1	$B \cdot 1 = B$	T1′	$B + 0 = B$	Identity
A2	$\overline{0} = 1$	A2′	$\overline{1} = 0$	NOT	T2	$B \cdot 0 = 0$	T2′	$B + 1 = 1$	Null Element
A3	$0 \cdot 0 = 0$	A3′	$1 + 1 = 1$	AND/OR	T3	$B \cdot B = B$	T3′	$B + B = B$	Idempotency
A4	$1 \cdot 1 = 1$	A4′	$0 + 0 = 0$	AND/OR	T4		$\overline{\overline{B}} = B$		Involution
A5	$0 \cdot 1 = 1 \cdot 0 = 0$	A5′	$1 + 0 = 0 + 1 = 1$	AND/OR	T5	$B \cdot \overline{B} = 0$	T5′	$B + \overline{B} = 1$	Complements

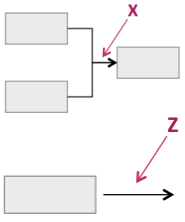
	Theorem		Dual	Name
T6	$B \cdot C = C \cdot B$	T6′	$B + C = C + B$	Commutativity
T7	$(B \cdot C) \cdot D = B \cdot (C \cdot D)$	T7′	$(B + C) + D = B + (C + D)$	Associativity
T8	$(B \cdot C) + (B \cdot D) = B \cdot (C + D)$	T8′	$(B + C) \cdot (B + D) = B + (C \cdot D)$	Distributivity
T9	$B \cdot (B + C) = B$	T9′	$B + (B \cdot C) = B$	Covering
T10	$(B \cdot C) + (B \cdot \overline{C}) = B$	T10′	$(B + C) \cdot (B + \overline{C}) = B$	Combining
T11	$(B \cdot C) + (\overline{B} \cdot D) + (C \cdot D) = B \cdot C + \overline{B} \cdot D$	T11′	$(B + C) \cdot (\overline{B} + D) \cdot (C + D) = (B + C) \cdot (\overline{B} + D)$	Consensus
T12	$\overline{B_0 \cdot B_1 \cdot B_2 \ldots} = (\overline{B_0} + \overline{B_1} + \overline{B_2} \ldots)$	T12′	$\overline{B_0 + B_1 + B_2 \ldots} = (\overline{B_0} \cdot \overline{B_1} \cdot \overline{B_2})$	De Morgan's Theorem

- Simplify Boolean expressions: apply axioms and theorems, bubble pushing (and Karnaugh[6] maps)
- Bubble pushing: pushing bubble backward (from the output) or forward (from the inputs) changes the body of the gate from AND to OR or vice versa; rules: begin at the output and work toward the inputs, push any bubbles on the final output back towards the inputs, draw each gate in a form so that bubbles cancel



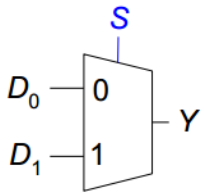
5 Combinational Circuits Design

- Minterm: product (AND) that includes all input variables; maxterm: sum (OR) that includes all input variables
- **Sum-of-products form (aka DNF; SOP)**: each row in truth table has a minterm, a product of literals (AND), and each minterm is true for that (and only that) row; formed by ORing the minterms for which the output is TRUE
- **Product-of-sums (aka CNF; POS):** each row in truth table has a maxterm, a sum of literals (OR), and each minterm is false for that (and only that) row; formed by ANDing the maxterms for which the output is FALSE
- **Karnaugh Maps (K-Maps)** minimize Boolean equations graphically; work well for up to four variables; useful on logic synthesizers to produce simplified circuits
- K-Map rules: special order for bit combinations – only one bit change from one to next; every 1 in a K-map must be circled at least once; each circle must span a power of 2 squares in each direction; each circle must be as large as possible: a circle may wrap around the edges; a "don't care" (X) is circled only if it helps minimize the equation; after forming all the circles, examine each circle and check which variables *don't* change, these variables (with their respective value) form the minterms (-> SOP).
- **Circuit schematics**: inputs: left (top); outputs: right (bottom); circuits flow from left to right; straight wires are better than wires with multiple corners; wires always connect at a T junction; a dot at an intersection indicates a connection (and the lack of a dot means there is no connection)
- Compress a truth table by using **"don't cares" (X)**
- **Contention** (**X**)[7]: when a signal is being driven to 1 and 0 simultaneously; not a real level; usually a problem since two outputs drive one node to opposite values; normally there should be only one direr for every connection
- **High-impedance** or **tri-state** or **floating** (**Z**): when an output is disconnected, not a real level; e.g. tr-sitate buffer; floating nodes are used in tri-state busses where many diferent



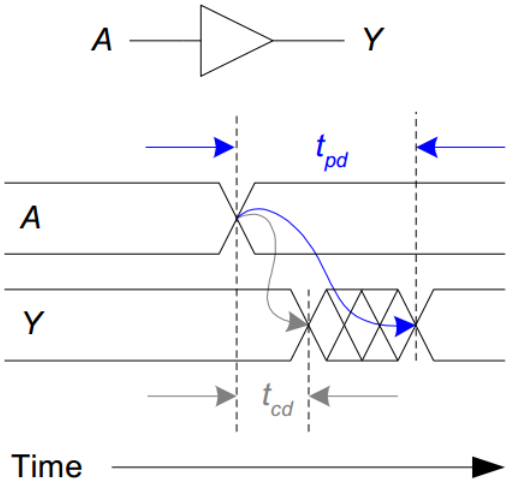
[6] See lecture 5
[7] WARNING: "don't care"s and contention are both called "X", but are NOT the same; Verilog uses "X" for both

dirvers share one common connection but only exactly one dirver is active at any time, the others are disconnected and said to be floating yet more than one onput can listen to the sgared bus wothout problems
- Combinational buildng blocks: combinational logic is often grouped into alrger building blocks to buld more complex systems and hides unnecessary gate-level details to emphasize the function
- **Multiplexer (mux)**: selects between one of the N inputs to connect to the output; needs $\log_2 N$-bit contol input; a 4:1 mux can be implementedusing two-level logi, using tri-state buffers, or using a tree of 2:1 muxes; a mux can be programmed perfrom any N-input loginc using a 2^N-input mux – it's a lookup table (!)



- **Decoders**: N inputs, 2^N outputs; one-hot outputs: only one output HIGH at once; decoders use minterms for logic

Timing



- The **propagation delay**, t_{pd}, is the maximum time from when an input changes until the output or outputs reach their final value.
- The **contamination delay**, t_{cd}, is the minimum time from when an input changes until any output starts to change its value.
- Delay is caused capacitance rand resistance in a circuit and the limitation of the speed of light[8]
- Reasons for t_{pd} and t_{cd} to be different: different rising and falling delays; multiple inputs and output, some of which are faster than other; circuits slow down when hot and speed up when cold
- **Critical (long) and short paths**: the critical path should be kept as short as possible
- **Glitch**: when a single input change causes multiple output changes yet they don't cause problems because of synchronous design conventions; noticing a glitch: generally speaking, a glitch can occur when a change in a single variable crosses the boundary between two prime implicants in a K-map

6 FPGA Systems, Experimental Board

- FPGA = Field Programmable Gate Array
- The FPGA is/can/has/…: array of configurable logic blocks (CBLs); performs combinational sequential logic; programmable internal connections (CLB-IOB); IOBs (I/O buffers) to connect to the outside world
- CLBs are composed of: LUTs (lookup tables) for combinational logic, flip-flops for sequential functions and multiplexers to connect LUTs and flip-flops

7 Verilog Combinational Circuits

Note The goal (in any of the Verilog lectures) is not to provide a syntax reference.

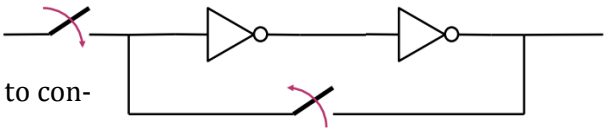
- Verilog is an **HDL** (hardware description language) just like VHDL, both exist
- An HDL is a convenient way of drawing schematics, it's standard, it's not proprietary, it's machine readable and can be used for simulation and synthesis
- Module: **module** *name* (**input** *[***[***bus width***]***[9]*] *name*, …, **output** *[***[***bus width***]***]* *name*, …) *body* **endmodule**
- Verilog is case-sensitive; names cannot start with numbers; whitespace is ignored; comments work Java/PHP-style (// for single line, /* … */ for multi-line)
- Use little-endian style for busses, i.e. MSB to LSB ([31:0]) + the usual good practices
- There are two main styles of HDL (which, in practice, are combined): **structural**: describes how modules are interconnected, each module contains other modules (instances thereof), and it describes a hierarchy; **behavioral**: the module body contains a functional description of the circuit, and contains logical and mathematical operators

[8] Let's take a minute to remember the lecture where Frank was complaining how slow the speed of light is! :)
[9] **[***range_start* : *range_end***]**; you can parameterize modules with "**#**(**parameter** *name* **=** *value*)"

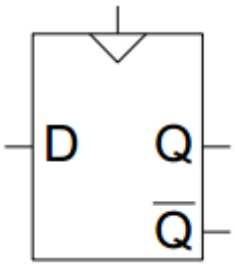
- Module instantiation: *module_name instance_name* **(***pin1, pin2, …***)**; alternate style (safer choice, pin order explicitly defined): *module_name instance_name***(.***pin_name***(***cable_name***)***, …***)**
- **Bitwise operators**: AND: *a&b*, OR: a|*b*; XOR: a^*b*; NAND: ~*(a&b)*; NOR: ~*(a|b)*
- **Reduction operator**: assign *a* to *y*: "assign y = &a"
- Conditional assignment (**ternary operator**): "assign y = *condition* ? *then* : *else*;"; if/else/elseif: use brackets for conditions "then" after if/elseif and if you have multiple lines, every block needs to be wrapped in "begin … end"
- **Numbers** are expressed like this: $N'Bxxx$, where N is the number of bits, B the base (b for binary, h for hexadedecimal, d for decimal and o for octal), and xxx is the number expressed in that base[10], can also contain X and Z, and underscores for readability
- In addition to the operators above and common ones, there are (arithmetic) shifts: <<,>> (<<<,>>>)
- **Bit concatenation**: use "{…}" for concatenation and "[i]" to address a single bit (or a range) of a value; you can precede a "{…}" block with an integer to have multiples of that value
- Timing relations defined in Verilog can only be used for simulation, not for synthesis

8 Sequential Circuits Design

- The outputs of sequential logic depend on current and prior input values – it has memory
- Definitions: state: all the information about a circuit necessary to explain its future behavior; latches and flop-flops: state element that store one bit of state; synchronous sequential circuits: combinational logic followed by a bank of flip-flops
- **Sequential circuits**: give sequence to events, have (short-term) memory, and use feedback from output to input to store information
- **State elements**: the state of a circuit influences its future behavior, the state is stored in state elements; bi stable circuits: SR latch, D latch, D flip-flop
- **Bistable circuits** can have two distinct state and remain in a state once they're in a state; to change the state a switch is sued to break the loop and another switch is used to connect the circuit to an input

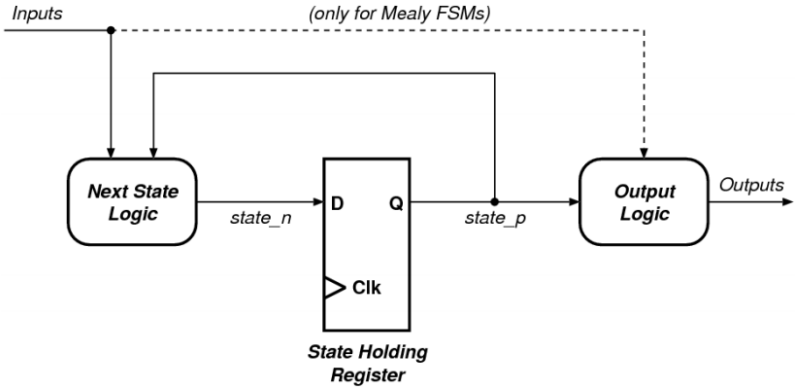


- **D Latch** is the basis bistable circuit in modern CMOS wher the clock controls the switches (only one is active at a time); the input is called D (data), the output Q; it can be in either latch mode (where the state is kept) or transparent mode (where the state changes)
- A rising edge triggered **D Flip-Flop** has two inputs, CLK and D; it samples D on the rising edge of the clock and D passes through to Q, otherwise Q holds its previous value (Q changes only on the rising edge); it's called an **edge-triggered** device since it is activate on the clock edge; internal a D flip-flop are two back-to-back latches controlled by complementary clocks



- **Registers** are multiple parallel flip-flops and can store more than one bit
- **Enabled flip-flops** are controlled by an additional input EN (enable) when new data is stored
- **Resettable flip-flops** have an additional RESET input which, when activated, forces Q to 0; there are two different types: synchronous (reset at the clock edge only) and asynchronous (resets immediately)
- **Settable flip-flops** set Q to 1 when SET is activated, otherwise behaves like normal
- Sequential circuits are all circuits that aren't combinational
- **Synchronous** sequential logic **design**: breaks cyclic paths by inserting registers (since their state changes at the clock edge, the system is called synchronized to the clock); every circuit element is either a register or a combinational unit; at least one circuit element is a register; all registers receive the same clock signal; every click path contains at least one register; two common synchronous sequential circuits. FSMs (finite state machines) and pipelines

[10] Heads-up: One hex number takes up 4 bits, not 1 (source of a nasty bug), e.g. 1'hf is wrong, 4'hf is correct



- An **FSM** consists of: state register (sequential circuit) to store current state and load the next state at the clock edge; next state logic (combinational circuit) which determine what the next state will be; output logic (combinational circuit) generates the outputs; name is derived from the fact that a circuit with k registers can bin in one of a fnite number (2^k) of unique states
- **Moore & Mealy** FSMs: are two different FSM types and differ in the output logic; while in a Moore FSM the outputs depend only on the current state, the outputs in a Mealy FSM depend on the current state and the inputs[11]
- *Example: see slides 30 – 47 (traffic light)*
- **FSM design procedure**: prepare: identify inputs and outputs, sketch a state transition diagram, write a state transition table, and select state encoding; for a Moore: rewrite the state transition tale with the selected state encoding, and write the output table; for a Mealy: rewrite the combined state transition and output table with the selected state encodings; for both: write Boolean equations for the next state and output logic and sketch circuit schematic

9 Verilog Sequential Circuits

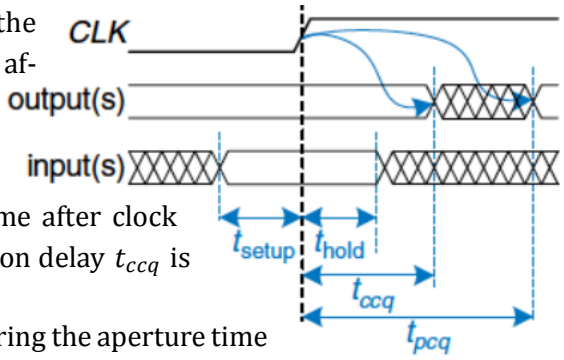
- Memory blocks (flip-flops, latches, FSMs) need to be defined; **sequential logic is triggered by a CLOCK event**; new constructs are needed (always, initial)
- always: **always @ (***sensitivity list***) begin** … **end**
- E.g. flip-flop: always @ (posedge clk) q <= d;
- "assign" is not used within an always block, "<=" describes a non-blocking assignment; the assigned variables need to be declared as **reg**
- E.g. D latch: always @ (clk, d) if (clk) q <= d;
- If, in an always block, the stamen define the signal completely, nothing is memorized and the block becomes combinational
- Switch/case: **case (***variable***)** *val1*: **begin** … **end**[12]**;** … **default:** … **endcase**; always use a default clause; casez works with don't cares
- Non-blocking vs blocking **assignments**; non-blocking (<=) assignments assign the values at the end of the block, in parallel thus the process flow is not blocked; blocking (=) assignments assign the value immediately, the process waits for the first assignment to complete thus it block progress; **rule**: use always blocks and non-blocking (<=) for synchronous sequential logic and continuous assignments (assign…) for simple combinational logic; use always blocks and blocking assignments (=) to model more complicated logic; do not make assignments to the same signal in more than one always/assign statement

10 Sequential Circuits: Timing

- If the data D sampled by a flip-flop isn't stable but changes when it is sampled, **metastability** can occur

[11] Mnemonic help (from the VIS forum): Meeealy = iiiinput
[12] As always, begin…end is only used for non-oneliners

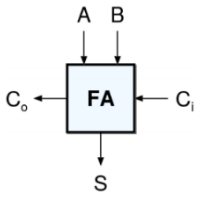
- **Input timing constraints**: setup time t_{setup} is the time before the clock edge that data must be stable; hold time t_{hold} is the time after the clock edge that data must be stable; aperture time t_a is the time around the clock edge that data must be stable, $t_a = t_{setup} + t_{hold}$



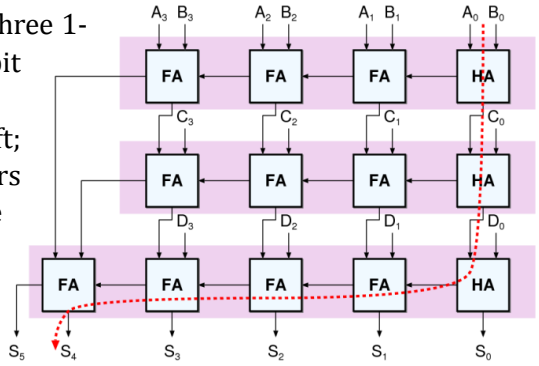
- **Output timing constraints**: propagation delay t_{pcq} is the time after clock edge that the output Q is guaranteed to be stable; contamination delay t_{ccq} is the time after clock edge that Q might be unstable
- The input to a synchronous sequential circuit must be stable during the aperture time around the clock edge; the delay between register has a minim and maximum delay, dependent on the delay of the circuit elements
- The clock period or cycle time T_C is the time beween rising edges of the clock signal, its reciprocal $f_c = 1/T_c$ $[Hz]$ is the clokc frequency, measured in Hertz or cycles per second; increasing the frequency increases the work that can be accomplished per unit of time
- **Setup time constraint**: $T_C \geq t_{pcq} + t_{pd} + t_{setup}$; $t_{pd} \leq T_C - (t_{pcq} + t_{setup})$
- **Hold time constraint**: $t_{hold} < t_{ccq} + t_c$; $t_{cd} > t_{hold} - t_{ccq}$
- **Clock skew**; the clock doesn't arrive at all register at the same time; worst case has to be determined to guarantee the dynamic discipline is not violated for any register
- The dynamic disciple can also be violated by asynchronous (user) input
- Any bistable device has two stable states and a metastable state in-between; if a flip-flop lands in the metastable state, it could stay there for an undetermined amount of time, yet if you wait long enough it will (most likely) have resolved to a stable state
- **Synchronizers**: since asynchronous inputs D are inevitable (UI, etc.) the goal of synchronizers is to reduce the probability of the output Q being metastable; it can be built with two back-to-back flip-flops; the MTBF (mean time between failures) indicates how often a synchronizer fails
- **Parallelism**: spatial (duplicate hardware) and temporal (split up the task, aka pipelining) parallelism; parallelism increases throughput
- Definitions: token: a group of inputs process to produce a group of outputs; latency: tome for one token from start to end; throughput: number of tokens that can be produced per unit time

11 Arithmetic Circuits

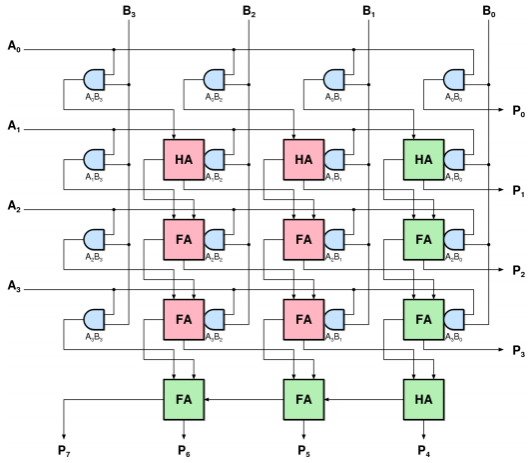
- Better arithmetic circuits, better performance
- Different types (from least to most complex): shift/rotate, compare de-/increment, negate, add/subtract, multiply, divide, square root, exponentiation, logarithmic/trigonometric functions; addition being the most important one
- **Half-adder** is the simplest arithmetic block adding two 1-bit numbers, producing a 2-bit number (sum and carry-bit)



- **Full-adder** (pictured) is the essential arithmetic block, adding three 1-bit (2 numbers and 1 carry; A, B, C_{in}) numbers, producing a 2-bit number (sum and another carry; S, C_{out})
- To add multiple digits,, the carry from the right is added to the left; done in a **ripple carry adder** (RCA; pictured) in carry save adders since multiple fast adders aren't that efficient; the curse of the carry: the most significant outputs of the adder depends on the least significant inputs



- **Multipliers** (pictured) are the largest common arithmetic block and consists of three parts: partial product generation, carry save tree to reduce partial products and a carry propagate adder to finalize the addition; since performance matters, there are (again) many optimization alternatives



- Several artihmetic operations are based on adders: negators, icnrementer, subtracter, adder subtracter, comparator
- **Negating**: (2' complement) all bits are inverted, one is added to the result
- **Incrementer**: C_{in} used as the incrementer input, input B is zero; decrementer is similar
- **Subtracter:** B input is inverter, C_{in} is used to complement B
- **Comparator:** based on asubtractor
- Functions not realized using adders: shoft/rotae; binary logic
- **Shifters**: logical shofters (<<,>>) shitf vlaue to the left/right and fill empty spaces with 0s; artimetic shifters (<<<,>>>) work similar, but on right shift, they fill empty spaces with the old MSB; can be used as multiplies and divieders, logic left shift multiples by 2^N, arttmatic right shoft divides by 2^N
- **Rtators** rotates buts in a ciurcle
- Division is often implemented by using other hardware iteratively; exp/log/trig is (less common)using dedicated hardware or uses numerical approximations or lookup tables (more common=
- **ALU** (arithmetic logic unit) defines the basic operations a CPU can perform directly; mostly a collection of resources that work in parallel

12 Number Systems[13]

- **Fractions** can either be represented with fixed-point where the binary point is fixed (e.g. for 4 integer and 3 fraction bits: $0110(.)110 = 2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$)), or floating-point where the binary point floats to the right of the most significant 1 and an exponent is used
- **Fixed-point**: the binary point is not a part of the representation but implied (and be agreed upon); negative fractions can be expressed either using sing/magnitude or two's complement
- **Floating-point**: similar to decimal scientific notation: general form: $\pm M \times B^E$, where M is the mantissa, B the base, and E the exponent; e.g. $273_{10} = \underbrace{2.73}_{M} \times \underbrace{10}_{B}^{\overbrace{2}^{E}}$
- Storing a 32-bit float: 1 bit for the sign, 8 bits for the exponent, and 23 bits for the XXX-part (see below)
- First representation: store everything like it is, naïve; XXX is the mantissa
- Second representation: omit the first bit of the mantissa, since it's always 1 (by definition); XXX is the fraction
- Third representation (IEEE standard): XXX is the fraction, the exponent becomes a biased exponent; the biased exponent is the bias + the exponent; the bias for 8 bits is 127_{10}; this representation has the following special cases defined (format: sign/exponent/XXX-part): 0: $[X|0|0]$, ∞: $[0|1|0]$, $-\infty$: $[1|1|0]$, NaN[14]: $[X|1|\text{non} - \text{zero}]$
- **Precision:** single: 32 bits, bias is 127; double: 64 bits, bias is 1023, 1 sign bit, 11 exponent bits, 52 fraction bits
- **Rounding**: problems: underflow – number is too small, overflow – number is too big; rounding modes: down, up, toward zero, to nearest
- **Addition**: 1) extract exponent and fraction bits, 2) prepend leading 1 to form mantissa, 3) compare exponents, 4) shift smaller mantissa if necessary, 5) add mantissas, 6) normalize mantissa and adjust exponent if necessary, 7) round result, 8) assemble exponent and fraction back into floating-point format[15]

[13] If you want to delve deeper in that subject, http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html seems to be a good place to start (what I've heard, haven't read it myself (yet)).
[14] Not a number
[15] Example: see slides 27 – 29

14 Verilog Testbenches

- HDL code to test another HDL module, the device/unit under test (dut/uut)
- Verilog: *#time* makes the program wait/pause in simulation for that time in nanoseconds
- Verilog: **initial** has the same syntax like **always** but is only executed once
- **Simple testbench**: you write a few tests, apply a series of inputs, observe and compare the outputs (in a simulator program); the statements have to be blocking
- **Self-checking testbench**: includes a statement to check the current state, use **$display** to output a message; a lot of work and you make mistakes when writing the test, too
- **Testbench with testvectors**: most elaborate; uses a high-level model (golden model) to product the correct input and output vectors; the testbench generates the clock for assigning inputs and reading outputs, reads the testvectors into an array, assign inputs, gets the outputs form the DUT, and compares the actual with the expected outputs and reports error
- Verilog: assign a vector to multiple variables: **{***var1, var2, …***} =** *vector***[***i***]**
- Verification needs to happen automatically and is difficult and takes a long time for all possible cases; formal verification methods are needed and critical cases need to be checked

15 MIPS Assembly[16]

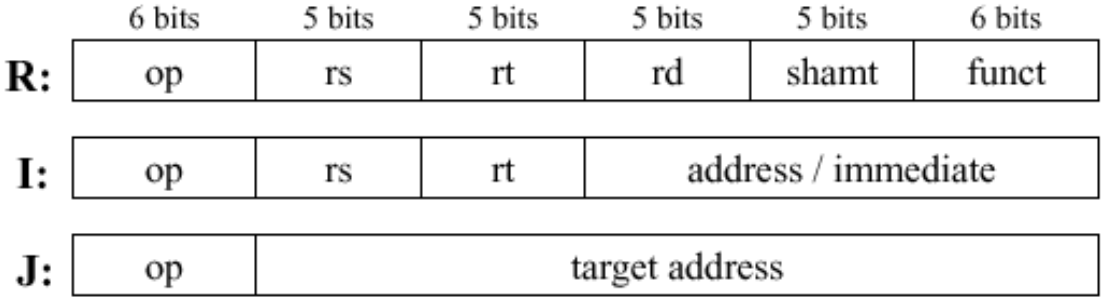
Abstraction Levels	Examples
Application Software	Programs
Operating Systems	Device drivers
Architecture	Instructions, Registers
Micro architecture	Datapath, Controllers
Logic	Adders, Memories
Digital Circuits	AND gates, NOT gates

- **Architecture design principles**: simplicity favors regularity; make the common case fast; smaller is faster; good design demands good compromises
- **Main types of MIPS instructions**: R(egister), I(mmediate), J(ump)
- The **Assembly language** consists of **instructions** taken from an **instruction set** and is human-readable; it is translated into machine language, readable by the computer
- Some examples: $a = b + c$: **add** a, b, c; $a = b - c$: **sub** a, b, c
- **RISC** and **CISC**: reduced instruction set computer: small number of simple instructions (e.g. MIPS); complex instruction set computers: larger number of instruction (e.g. x86)

Name	Register Number	Usage
$0	0	the constant value 0
$at	1	assembler temporary
$v0-$v1	2-3	procedure return values
$a0-$a3	4-7	procedure arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved variables
$t8-$t9	24-25	more temporaries
$k0-$k1	26-27	OS temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	procedure return address

- **Operands**: a computer can retrieve operand from either registers, memory or constants (immediates)
- **Register**: since main memory is slow, most architectures have a small set of fast register (MIPS: 32)
- **Memory**: since there's too much data for only 32 registers, more data is stored in (slow) memory while commonly used variables are kept in register; reads are called loads (lw), writes are called stores (sw)
- **lw/sw (lb/sb)**: for the memory address to still fit into a 32-bit instruction, a special method is used to calculate the address; say you have the instruction "lw $s3, 1($0)" you have the base address "$0", the offset "1" which together form the address: $\$0 + 1 = 1$; the offset can be decimal (default) or hexadecimal
- Both, **byte-addressable** and **word-addressable** memory exists, MIPS uses byte-addressable memory; every 32-bit word has 4 bytes thus the word address increment by 4
- **Big- and little Endian** refers to whether byte numbers start at the LSB (little) or MSB (big)
- **Constants/immediates**: immediates don't require register for memory access, they are directly available; e.g. addi (add immediate; subi is not necessary)
- R-type: add, sub; I-type: addi, lw, sw;
- Below: overview of the different instruction types[17]

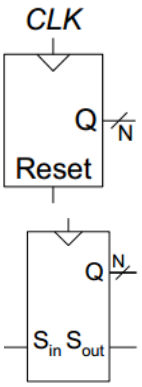
[16] Great. Resource. http://en.wikipedia.org/wiki/MIPS_instruction_set#MIPS_assembly_language
[17] http://www.cise.ufl.edu/~mssz/CompOrg/Figure2.7-MIPSinstrFmt.gif



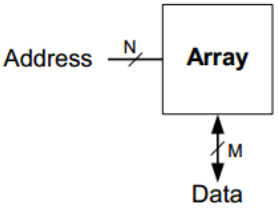
op: basic operation of the instruction (opcode)
rs: first source operand register
rt: second source operand register
rd: destination operand register
shamt: shift amount
funct: selects the specific variant of the opcode (function code)
address: offset for load/store instructions (+/-2^{15})
immediate: constants for immediate instructions

- **Stored program**: 32-bit instruction data stored in memory, a sequence of instructions; no re-wiring required for a new programs – program is fetched (read) from memory in sequence
- **Program counter**: to keep track of the current instruction; in MPIS it starts at 0x00400000
- To interpret machine language code, read the opcode (if all 0, it's an R-type, use function bits)

16 Memory Systems



- Common sequential building blocks: various counters; serial/parallel converters, serial in- serial out is a shit register, parallel in – parallel out is normal register
- **Counters** (aside) are incremented on each clock edge; can be used for digital clock displays and program counters
- **Shift register** (aside): shifts a new value in on each clock edge, shift a vlaue out on each clock edge; serial-to-parallel converter $S_{in} \rightarrow Q_{0:N-1}$); with parallel load: if load = 1, it acts as a normal N-bit register, when load = 0, it acts as a shoft register; it can now act as a serial-to-parallel ocnverter or a parallel-to-serial converter ($D_{0:N-1} \rightarrow S_{out}$)
- Memories are large blocks, they are practical tools for system design, and they allow you to store data and work on soterd data
- Different ways to store data:
- **Flip-flops** are very fast, allow for parallel access but are expensive[18]
- **SRAM** (s for static) are relatively fast and not that expensive but allow for only one data word at a time; stores data by cross couples inverters
- **DRAM** (d for dynamic) are slower, reading destroy content, they can only read one data word at a time, they need a special process BUT they are even cheaper; sotres data by charging asmall capactor which slowly discharges (i.e. forgets the value) thus needs to be refreshed every now and then; the larger the capcatior, the longer it takes to forget
- Even slower but non-volatile: HDD, SSD, etc.



- **Array organzitation of memories** (aside) ; efficiently store large amounts of data: memory array to store data, address election logic (to select an array row) and a readout ciruity; an M-bit balue can be r/w'ed at ech nuique N-bit addressed: all values can be access, but only M bits at a time; acces restriciton allows more compact orignaization; the array is 2D with 2^N rows and M columns whereas the depth is the number of rows (words) and the width is the number of columns (size of a word)
- **Memory types**: volatile: loses data on power off, e.g. SRAM, DRM; non-volatile: keeps data even without power, e.g. ROM, flash memory

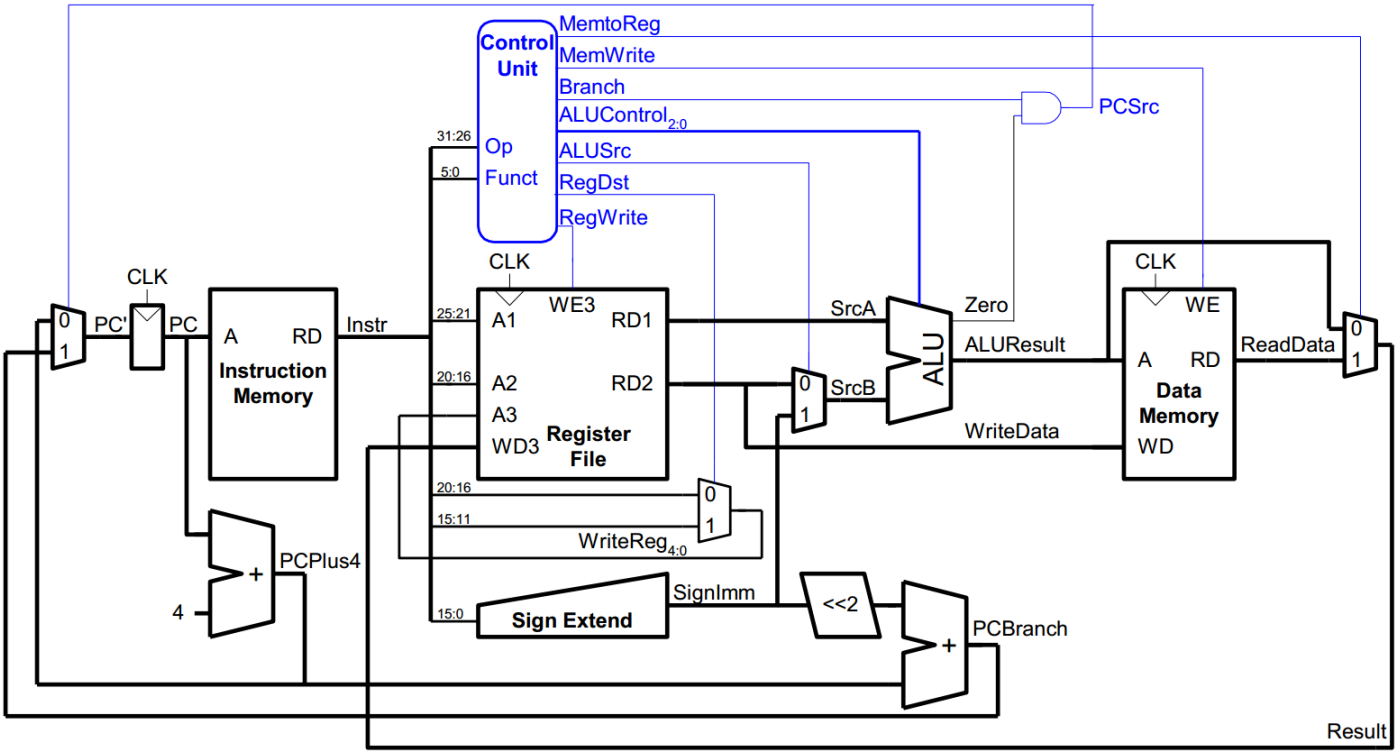
[18] As in numbers of transistors required

- **ROM** (read-only memories): can be made much denser since they are not only non-volatile but there's also no need to change the content; used in embedded system, for configuration data, lookup tables, …; re-writable (flash) memories are commonly used and since writing is very slow, they're (form an application
- **Logic with memory arrays**: called lookup table (LUTs) to look up values
- **Multi-ported memories**: a port is an address/data pair; small multi-ported memories are called register files

17 Microarchitecture: Single Cycle Architecture

- **Microarchitecture**: how to implement an architecture in hardware
- **Processor**: data path: functional blocks; control: control signals
- **Single-cycle**: each instruction executes in a single cycle
- The **basic execution** of a microprocessor is: read one instruction, decode the instruction, find the operands in memory/registers, perform the operation according to the instruction, (if necessary) write the instruction, go to the next instruction
- The **basic components of the MIPS**: main: program/instruction memory, registers, data memory, ALU for the core function; auxiliary: program counter (PC), instruction decoder, a mean to manipulate the program counter for branches
- Design recap: register store the current state; combinational logic determines next state: data moves along a data path by control signals which depend on state and input; the clock moves us from one state to another
- The program is stored read-only as 32-bit binary; memory addresses are 32 bits wide and the actual address is stored in the PC
- The **PC** needs to be incremented by 4 during each cycle (as long as there are no jumps)
- The **register file** sores 32 register, each being 32-bit[19] and since $2^5 = 32$ 5 bits are needed for addressing; every R-type instruction uses 3 register (read: RS, RT; write: RD); a special memory with 2 read ports (2x address, 2x data out) and 1 write port (address, data in) is needed
- The **data memory** is used to store the bulk of data, having up to 2^{32} bytes (implies requiring address to be 32-bit)
- For **I-type** instructions to work (lw), too, the following changes to the register file have to be made: RS + sign extended immediate is the address, RS is always read, sometimes RT is read as well, if one register is written, it is RT, not RD, the write data can come from the memory or the ALU, and not all I-types wrote to the register file; and the flowing changes to the ALU: it is now also used to calculate memory addresses, the result is also used as a data memory address, one input is no longer RT, but the immediate; sw additionally needs to be able to write to the memory
- The following **control signals** are now required: RegWrite: write enable for the register file, RedDst: write to register RD or RT, AluSrc: ALU input RT or immediate, MemWrite: write enable, MemtoReg: register data in from memory or ALU, ALUOp: the ALU's operation
- **BEQ** (branch if equal): needs to ALU for comparison: subtracts $RS - RT$, and cheks if $RS == RT$ and if so, outputs a zero flag; a second adder is needed for the immediate value to be added to PC+4; now the PC can either be PC+4 or the new branch target address (BTA); a new **control signal** is needed: Branch: determines whether jumping or not
- **J-type**: doesn't need the ALU, nor memory or the register file, but adds one more option for the PC;); a new **control signal** is needed: Jump: direct jump (yes/no)
- **Recap:** *see slides 47 – 68*

[19] Verilog: reg [31:0] R_arr [31:0];



- **Processor performance**: to get the performance of a program, one has to look at the instructions which can take one or more clock cycle, measured in **cycles per instruction (CPI)**; the time on clock cycles takes is determined by: the critical path determines how time on cycles requires, the **clock period** and the **clock frequency** $f = 1/T$ gives you how many cycles can be done each second; $1/1\text{ns} = 1$ GHz
- General formula for N instructions: $N \cdot CPI \cdot (1/f) = N \cdot CPI \cdot T\ [s]$
- To **make a program run faster** you can: reduce the number of instructions (more CISC, better compilers), use less cycles to perform the instruction (RISC, use multiple units/ALIs/cores in parallel), or increase the clock frequency (better manufacturing technology, redesign time critical components, adopt pipelining)
- In our example, lw is the critical path T_C

18 MIPS Programming

- **Branching**: beq (branch if equal), bne (branch not taken), j (jumping; unconditionally)
- **Labels**: branching statements work with labels which indicate instruction locations and are followed by a colon
- When you transform high-level code to MIPS assembly code, sometimes (e.g. while loop) a beq is easier than a bne; transforming if/else, for, and while should be fairly easy, when using **arrays**, load the base address into a register and use that as an address base, e.g. `0($t1), 4($t1), …`
- Compute absolute value:

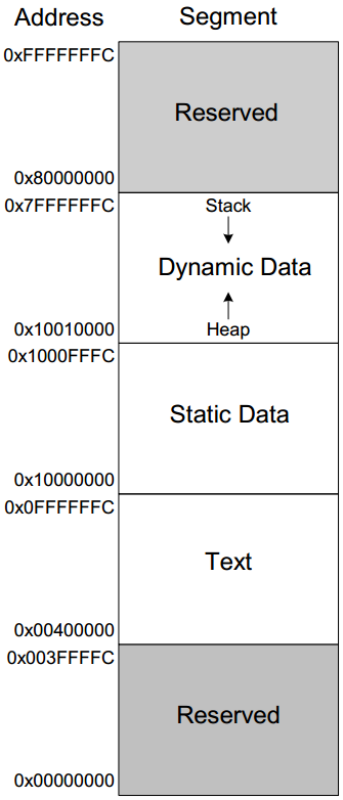
```
# input and output in $t0
sra $t1,$t0,31
xor $t0,$t0,$t1
sub $t0,$t0,$t1
```

Procedures

- **Procedures**: (caller: calling the procedure, callee: the called procedure) the caller passes arguments to callee and jumps there; the callee performs the procedure, returns the result to caller, returns to the point of call, and must not overwrite registers or memory need by the caller
- **MIPS procedure calling conventions:** call procedure: jal (jump and link; jumps and saves PC+4 in $ra); return from procedure: jr (jump register; jumps to $ra); argument values: $a0 – $a3; return value: $v0
- **The Stack**: memory used to temporarily save variables; it expands and contracts, thus adapting to memory requirements; it grows down (from higher to lower memory address); $sp (stack pointer) points to top of the stack; allows procedures to ensure they do not have any (unintended) side effects; e.g. sw $t0 8($sp)

- **Registers**: can be either preserved (i.e. saved to the stack on restored afterwards): $s0 – $s7, $ra, $sp, stack above $sp; or non-preserved: $t0 – $t9, $a0 – $a3, $v0 – $v1, stack below $sp

Addressing modes



- **Operands** can be addressed using: register only (operands found in registers), immediate (16-bit immediate used as an operand), base addressing (address of operand is: base address + sign extended immediate), PC-relative (beq), pseudo direct (jal)
- **Stored** in memory (aside) are: instructions (text), data (global/static: allocated before; dynamic: allocated within); the memory is at most $2^{32} \widehat{=}$ 4GB, address from $0x00000000$ to $0xFFFFFFFF$
- The MIPS also supports a few **pseudo instructions** which are like shortcuts; e.g. li, mul, clear, move, nop
- **Exceptions** can either be caused by hardware (interrupt, e.g. keyboard) or software (e.g. undefined instruction); if an exception occurs, the processor records the cause, jumps to the exception handler (0x80000180), and returns to the program; exception registers are not part of the register file and record the cause and the exception PC (EPC); coprocessor 0
- For add, addi, sub, mult, div, stl[20], slti, lh, lb unsigned variants (append a "u") exist
- **Floating-point instructions** are executed in coprocessor 1, have 32 32-bit float register ($f0 – $f31); double-precision floats are stored in two registers; there's a special **F-type** instruction format (name:width): op:6, cop:5, ft:5, fs:5, fd: 5, funct: 6

19 Cache Systems

- Until now, we assumed memory (on which performance depends on) could be accessed in 1 clock cycle (wrong)
- The **challenge** is to have fast, cheap and large memory but only two are possible, thus a hierarchy of memories is used
- **Locality** can be exploited to make memory access fast; either **temporal** (used recently, likely to use again soon) where recently accessed data is kept in higher levels of memory hierarchy or **spatial** (used recently, likely to use nearby soon) where data which is close to the data accessed is brought into higher levels, too.
- **Memory performance**: hit: found in that level; miss: not found, go to next level; **AMAT** (average memory access time)
- **Cache** is a safe place to hide things – it is the higher level in memory hierarchy, fast (around 1 cycle access time); it ideally supplies most of the data to the processor and holds most recently accessed data; **design questions**: What data is held in the cache? How is data found? What data is replaced?
- **Terminology**: capacity (C): the number of data bytes a cahge stores; block size (b): bytes of data brought into cache at once; number of block ($B = C/b$): the number of blocks in cahce; degree of associativity(N): the number of blokcs in a set; number of sets ($S = B/N$): each memory address maps to exactly one cache set
- **What data**: ideally the cache anticipates data (but prediction of the future is impossible) by using temporal and spatial locality
- **Find data**: cache is organized into S set, each memory address maps to exactly one set; caches are **categorized** by number of blocks in a set: direct mapped: 1 block per set (many conflict misses); n-way set associative: N blocks per set (associativity reduces conflict misses); fully associative: all cache blocks are in a single set (no conflict misses, expensive to build)

Organization type	**Number of ways (N)**	**Number of sets ($S = B/N$)**
Direct mapped	1	B
N-way set associative	$1 < N < B$	B/N

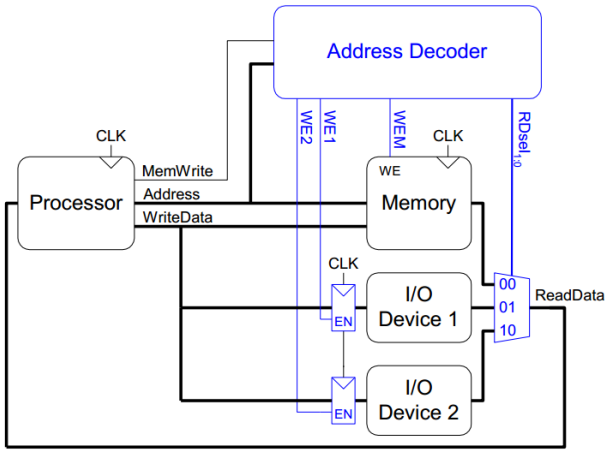
[20] $(A - B)[31]$

Fully associative	*B*	1

- By increasing block size (spatial locality) compulsory misses are reduced
- **Misses**: compulsory (first time data is accessed), capacity (cache too small), conflict (data maps to the same location; **miss penalty**: time it takes to retrieve a block from a lower level
- **Replaced data: LRU** (least recently used) data is replaced to reduce capacity misses
- (bigger) CPUs use multi-level caches (L1, L2, …)
- Additionally, there's **virtual memory** which gives the illusion of a bigger memory without the high cost DRAM; DRAM acts as a cache for the hard disk; in reality, every program uses virtual addresses and the CPU determines the physical address and each program has its own virtual to physical mapping (**memory protection**) and thus can't access data from other programs
- In virtual memory, one talks about **pages** instead of blocks, whereas the page size is the amount of memory transformed from HDD to DRAM at once and the page table is a lookup table to do address translation
- The **challenges** are: the page table is large; each load/store requires two main memory accesses(translation, access[21]); thus a **TLB** (translation lookaside buffer) is used to make sure page table access have a lot of temporal locality; the TLB is small, fully associative, and >99% hit rates are typical

20 IO Systems

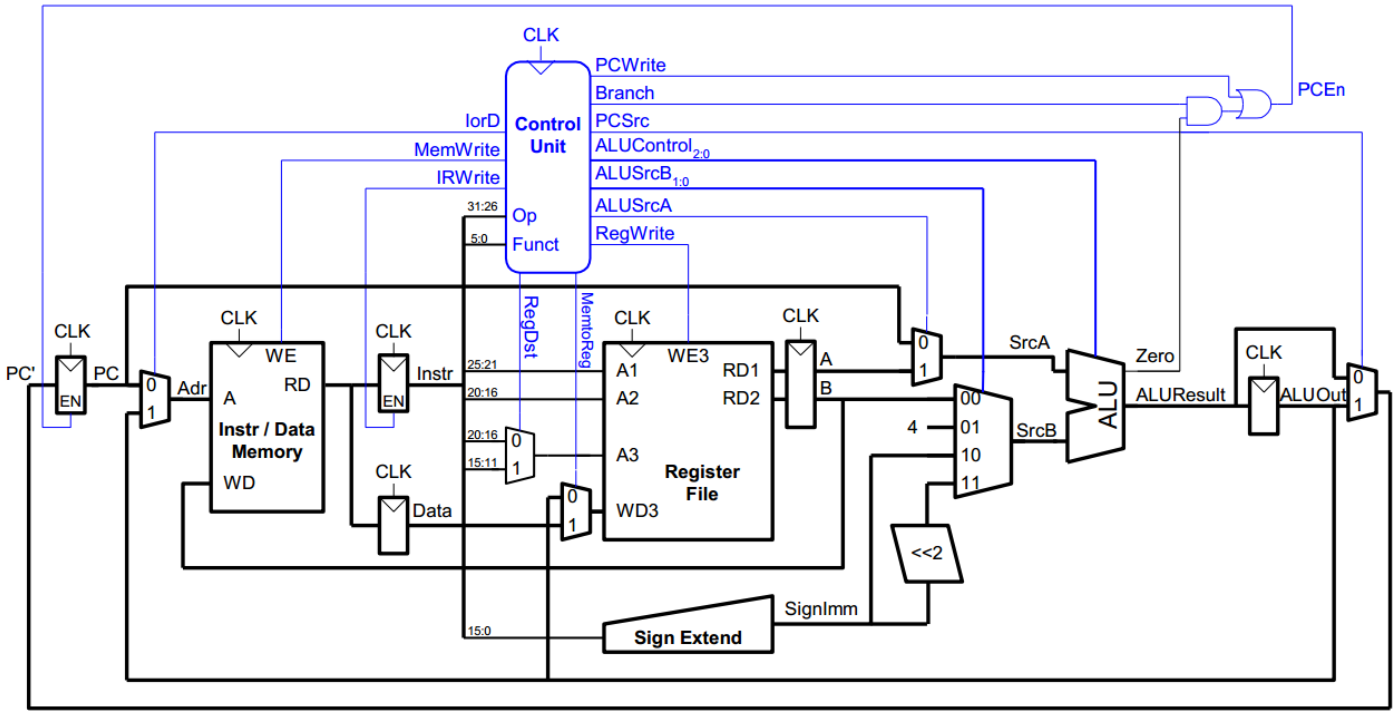
- **Memory-mapped I/O** access I/O devices just like it access memory with every I/O device being assigned to one or more addresses; a portion of the address space is dedicated to I/O devices (e.g. 0xFFFF0000 to 0xFFFFFFFF)
- Required **hardware**: address decoder to determine which memory communicates with the processor; I/O register to hold values written to I/OP devices; ReadData mux to select between memory and I/O



21 Microarchitecture: Multi-cycle Architecture

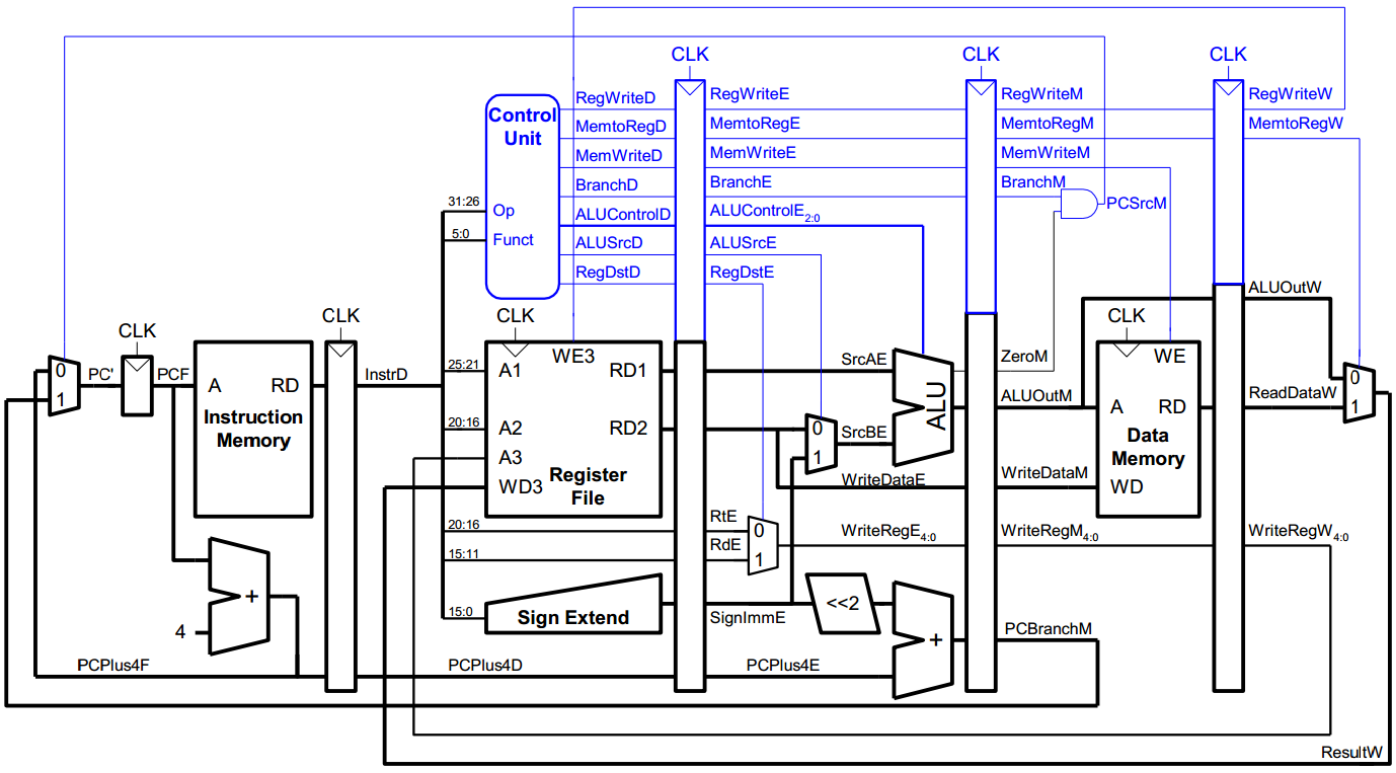
- While a simple-cycle microarchitecture is simple, the cycle time is limited by the longest instruction and it uses two adders/ALUs and two memories
- The benefits of a **multi-cycle** microarchitecture are higher clock speed, simpler instructions run faster and you can reuse expensive hardware in multiple cycles by the sequencing overhead is paid many time
- The things to optimize: use only one memory instead of two; use only the ALU instead of separate adders; divide all instruction into multiple steps
- *See slides 14 – 23 for data paths*

[21] Unless you use a TLB



22 Microarchitecture: Pipelined Architectures

- **Parallelism**: spatial (duplicate hardware) and temporal (split up the task, aka pipelining) parallelism; parallelism increases throughput
- Definitions: token: a group of inputs process to produce a group of outputs; latency: tome for one token from start to end; throughput: number of tokens that can be produced per unit time
- A pipelined MIPS processor deices the single-cycle processor into 5 **stages**: fetch, decode, execute, memory, writeback; it uses temporal parallelism and adds pipeline registers between the stages
- WriteReg must arrive at the same time as Result, thus it is slightly modified



- A pipeline **hazard** occurs when an instruction depends on result from a previous instruction that hasn't completed; this can be either a **data** hazard when the register value hasn't been written back yet or a **control** hazard where the next instruction is not yet decided (caused by branches).

- Example of **data hazards**: operations involving the register file have only half a clock cycle to complete; one instruction writes to a register and the next instruction reads from that register → RAW (read after write) hazard
- **Handling data hazards**: NOP (no operations); rearrange code at compile time; forward data at runtime (done by the hazard unit); stall the processor at runtime (hazard unit; add enable (EN) inputs to the fetch & decide pipeline regs and a synchronous CLR to the execute pipeline reg)
- **Control hazards**: beq: the branch is not determined until the fourth stage of the pipeline thus instruction after the branch are fetched before the branch occurs and thereby might have to be flushed (branch misprediction penalty); can be prevented by early branch resolution (consider history of previous branch yes/no; recognize loops)
- **FASTEST** MIPS

23 Advanced Processors

- **Deep pipelining**: pipeline all the things, up to 10-20 stages, limited by pipeline hazards, sequencing overhead, power, and cost
- **Branch prediction**: static branch prediction: check direction (backward/forward); dynamic branch prediction: keep history of branches in a buffer; ideal CPI = 1
- **Superscalar**: multiple copies of data path; dependencies make it tricks to issue multiple instructions; ideal IPC = 2
- **Out of order**: looks ahead multiple instruction to use as many as possible at once as long as there are no dependencies (RAW, WAR, WAR; W = write, A = after, R =read); instruction level parallelism (**ILP**); has a reorder buffer and a scoreboard table to keep track of what is available/needs to be executed/dependencies
- **SIMD**
- **Multithreading**: increase throughput (not ILP) by having multiple copies of the architectural state and multiple threads active at once which enables another thread to run immediately if one stalls and more than one thread can be active to keep all execution units busy

Review