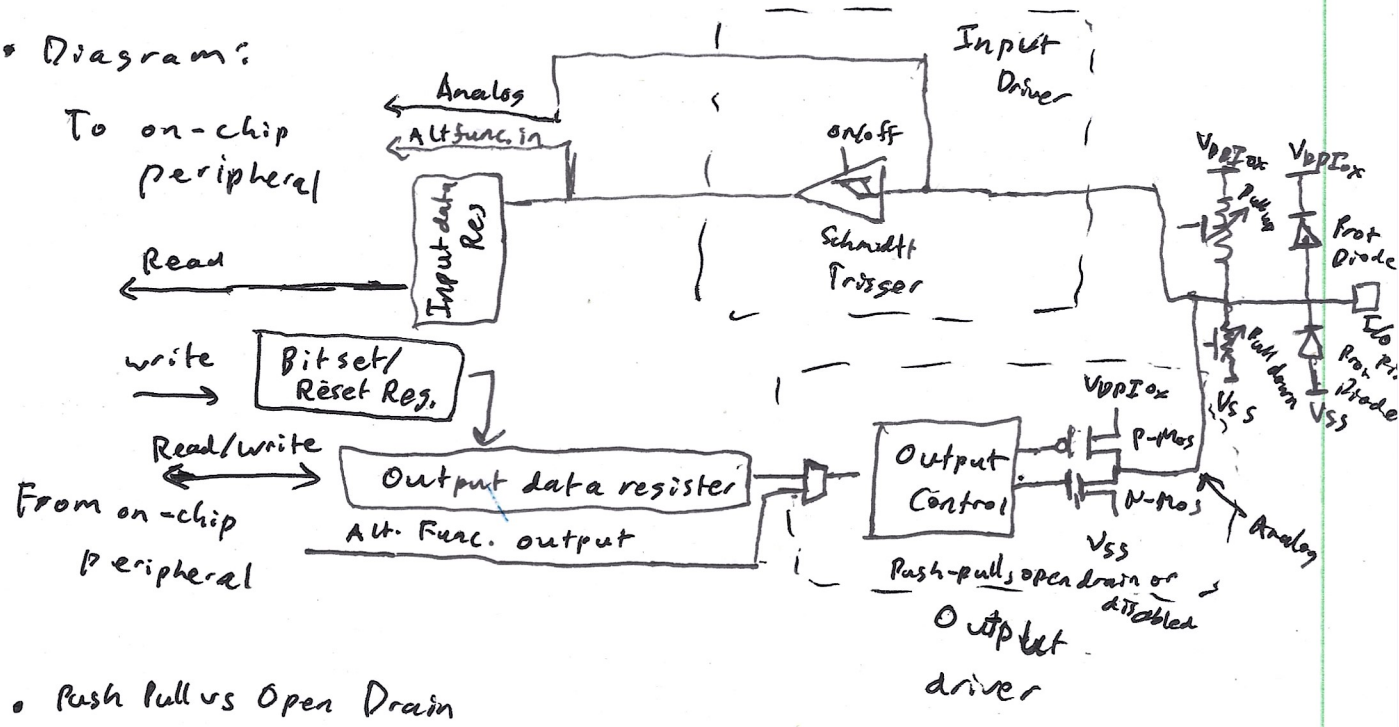
GPIO

- General Purpose Input/Output (GPIO)
  - can read & write w/ software
- Functions
  - Digital In
  - Digital Out
  - Other (ex: DAC or ADC)
- Diagram:



- Push Pull vs Open Drain
  - PP = both P-mos & N-Mos activate
  - OD = only N-mos operates (when 1 is written, pin has high imp.)
- Output Speed Control - rising/falling speeds
  - ↑speed = ↑ EMI Noise and ↑ power.
  - Consis. based on peripheral speed.
  - Slew Rate: $\frac{\Delta V}{\Delta t}$ (amount of time under max voltage to change)
- Pullup & down resistors to drain the buffer.
- Memory Mapped IO:

System	→	Sys. NVIC, Timers
Ext. Device	→	SD card
Ext. RAM	→	off-chip memory
Peripheral	→	Timers, GPIO
SRAM	→	on chip RAM
code	→	on chip flash.

- Use |= to change specific bits of a reg.

ECE 362 | Lecture 2

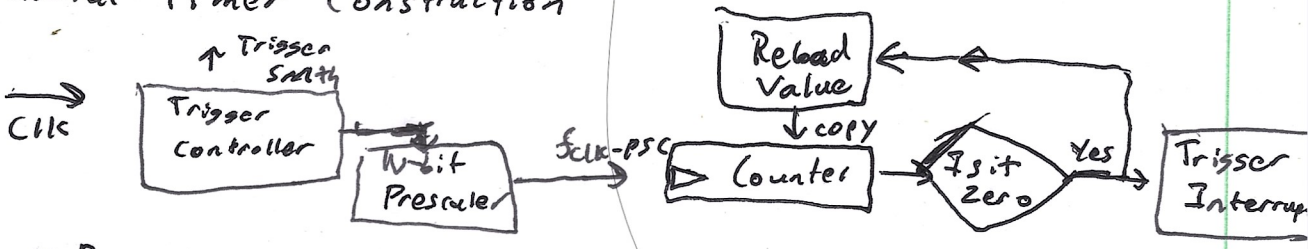
Interrupts

- Interrupts standard program sequence
  - Specific type of an exception (hardware invoked sub-routine call)
- Code is translated from C to assembly to machine code.
- Program Counter keeps track where you are in code.
- Microcontrollers are single threaded (not simul. ops.)
- Basics
  - Peripherals can interrupt
  - Can be interrupted by more than one thing.
    - Calls diff function Interrupt Service Routine (ISR) for each one.
  - Can be enabled and disabled
  - Have priorities, ↑ can skip over lower to go first
  - Pending interrupt = Raised but not handled interrupt.
    - Handler must tell periph. it acknowledges the request
- Flow of interrupt
  1. Peripheral raises particular interrupt
  2. CPU checks if its enabled
  3. If EN: mark "N" as pending
  4. checks priority level of "N" vs curr. CPL.
  5. If Prio-N > CPL: CPL = Prio-N, CPU state → Stack
     N the entry → PC of vec. table
  6. Running ISR.
- Exceptions will not stop unless higher prio one is raised.

ECE 362 | Lectures 3 & 4

Timers

- Used to periodically do things
  - SysTick is special, but several timers do exist.
    - Generates SysTick interrupts in a fixed intervals.
- General Timer Construction



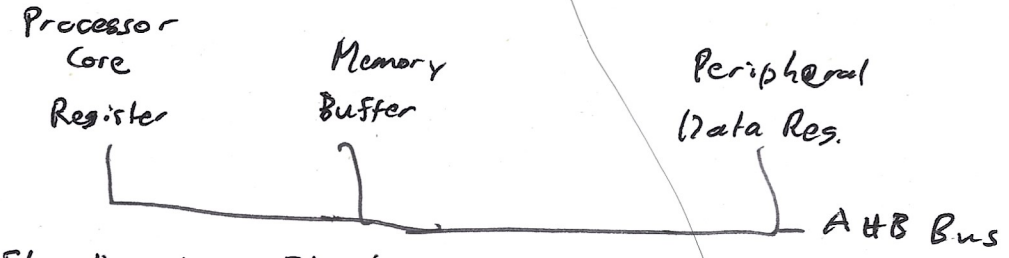
  - Prescaler divides clk.
  - Reload val. decides how far Counter counts.
  - $f_{clk-psc} = \frac{f_{clk}}{prescaler+1}$ - $f_{out} = \frac{f_{clk}}{(psc+1)(reload+1)}$
- We can use these timers to execute code regularly.

Multiplexing & Debouncing

- Electrical Debouncing is easy but hard for large # of buttons.
- For keypads, we can apply voltage to each row fast and then read out the columns that recieve voltage.
  - we can use a Timer ISR.
- Debouncing a matrix can be done through software.
  - We can keep a history of outputs from the matrix.
  - For 8 history byte, 00000001 - key press
    11111110 - key release
    11111111 - press + stable
    00000000 - release + stable
  - We want to scan the keypad faster than human reaction to record seperate key actions while slower than total bounce time so multiple incorrect bits in a row are not recorded.
- We can multiplex displays too, apply power to each digit and ground the segments to light up.

Direct Memory Access (DMA)

- DMA allows you to move data without the CPU actively taking part. This speeds up communication rate.
- Can move data between Memory buffer & Peripheral after cpu read/write.



- DMA Flow-through vs Fly-by
  - Flow through stores data as an intermediate buffer (to change data size or smth.)
  - Fly-by only connects bus between memory buffer and peripherals.
- DMA controller sits on bus, stores source & destination, etc.

Digital-to-Analog Converter (DAC)

- Converts digital data into a voltage signal w/ a N-bit Dac:

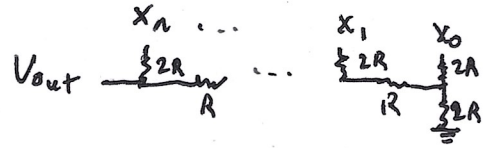
$$V_{DAC\,out} = V_{res} \cdot \frac{\text{Digital Value}}{2^N - 1}$$

(operates as a digital potentiometer)

  - Can be used for digital audio & waveform generator
  - Can optimize speed & resolution & power dissapation
- Binary fractions in fixed pt. notation is inexact
  - 16-bit value

$\overline{128}$ $\overline{64}$ $\overline{32}$ $\overline{16}$ $\overline{8}$ $\overline{4}$ $\overline{2}$ $\overline{1}$ • $\overline{1/2}$ $\overline{1/4}$ $\overline{1/8}$ $\overline{1/16}$ $\overline{1/32}$ $\overline{1/64}$ $\overline{1/128}$

- DAC arch. (op. at speed of light)



- Limits
  - Resistor tolerance
  - Minor errors become enlarged
- Wave table of digital values to convert can operate well to generate desired waves.
- $f_{wave\,table} = \frac{\text{samples/sec}}{\text{samples/cycle (of wave)}}$

ECE 362 | Lecture 7

Analog to Digital Conversion (ADC)

- ADC
  - Resolution: Number of binary bits in ADC output
    - X $V_{out_{ADC}} = \text{round}\left((2^N - 1) \cdot \frac{V_{in}}{V_{Ref}}\right)$
  - Quantization Error: max error from digital value vs analog.
    - X $\pm \frac{V_{Ref}}{(2^N - 1)(2)}$ = Max error Quantization (1/2 the lsb (least sig bit))
  - Sampling rate: how often the analog value is recorded.
    - X $f_{sampling} \geq 2 \cdot f_{signal}$
- How it works
  - Converges to value by adding bit by bit and checking if > or < actual value.
  - Input values are clipped at V_{Ref} and V_{SS}
- How to beat jitter in analog signals:
  - Box car averaging: Take lot of samples, average the last few for the new value at a slower update rate

ex. 1kHz update = 128k samp/sec + average last 128 samp.

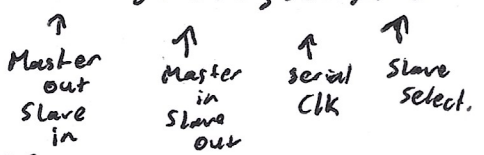
ECE 362 | Lectures 9 to 11

Advanced Timers + Pulse Width Modulation (PWM)

- Advanced timers include a compare and capture register (CCR) where it triggers an output when CCR = timer counter.
  - Active high mode = output is high signal
  - Toggle mode = output switches from high ↔ low.
  - PWM mode = is high < CCR, is low > CCR.
  - Also can have up count / down count or both.
- PWM Duty Cycle: $\frac{CCR}{Reload+1}$, $Period = (1+ARR) \cdot Clk\ period$
  - wave form generation with high and low digital signals
  - can program duty cycle + frequency
- GPIO as a DAC using PWM
  - varying duty cycle can do this
  - averaging the area under can generate + done w/ a low pass filter.
  - wave freq needs to be way higher than the target signal.
  - analog control of a transistor through DAC is inefficient, (power wise) PWM is only I/O so it removes this issue.
- No low pass filter for human perception and/or other signal dampeners.

Serial-Parallel Interface (SPI)

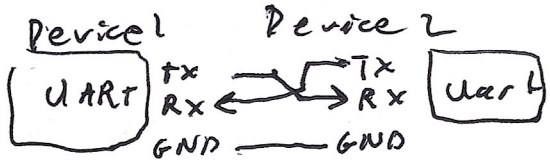
- Serial data transfers reduces wires.
- Synchronous: SPI, I^2C, USART, Async: UART
- SPI operates with min. one master + multiple slaves
  - master controls clk.
  - shifts in bits each clk cycle.
  - one clk. pulse per bit.
- Baud rate: bits/s (speed of transfer)
- Master initializes all data transfer, drives clk + slave select pins.
  - Pins MOSI, MISO, SCK, NSS (Master will have multiple slave selects if multiple slaves exist)



- Shift regs can be hooked up to a master device for data storage.
- Bidirectional SPI
  - Each transaction exchanges 2 words of 4-16 bits each between two devices
    - Usually master sends command + slave device responds + is ignored,
  - Master instead sends something to strobe SCK.

ECE 362 | Lecture 12

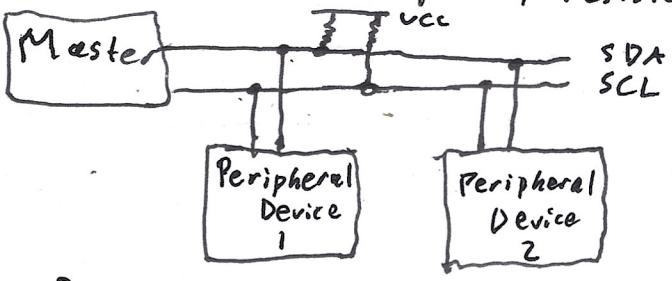
Universal Async. Reciever + Transmitter (UART)

- Programmable + Async
- 

- Use a start bit + stop bit for framing
- use ~~parity~~ parity bit to prevent interference (all bits + par. is even/odd)
  - Format: {start, 0xXX, par., stop}
  - Theres more advanced codes than parity
  - Baud rate is time for transaction
    - ~~Bits~~ Data Rate = Baud Rate / Size of Trans. (11 in this case)
    - Standards:
      - Idle high, start bit ↓, stop bit ↓
      - Sent LSB first, parity can be even/odd/none
      - stop bit can be .5, 1, 1.5, 2 bits long
      - word size 7-9 bits, parity not req.
      - Almost everyone uses 8N1 (8 bits, no parity, one stop bit)
  - RX → RX, devices can transmit simultaneously
    - each devices have sep clocks to read, but should be similar (5% tolerance)
- Errors
  - Framing Error: Didn't see stop bit (clk out of tol. or disagree in packet format)
  - Receiver Overrun: Didn't read received byte before new one started (Sys is too slow to read??)
  - Parity Error: bits don't add prop. (Noise)

Inter-Integrated Circuit (I^2C)

- Characteristics (useful bc only 2 wires)
  - Serial, byte-oriented
  - Multi-master, multi-slave
  - Two bidirectional open-drain lines: Serial Data Line (SDA), Serial Clock Line (SCL)
    - They need pull up resistors



- Devices pull low for 0, pull-up res. keep it high for 1.
- Baud rate:
  - Stand. 100 kb/s, 400 kb/s (fast), 1 mbit/s (fast+)
    10 kb/s (low speed mode), 3.4 mbit/s (high speed) (not prop.)
  - Limited by bus capacitance, pullup res. strength, length of network
- Clk pulse for every bit (two lines)
- Devices can rec., transmit, or both
- Slower than SPI, but more convenient.
- SDA transaction (MSB first)

  {Start bit, 7'b (slave addr), R/W, ACK (from sl), 8' (command), Ack, 8'(Data), Ack, P}

  R/W: 0, write; 1, read

  (0)

  or

  no command, all data replaces command w/ data

  2nd Ack is from master and last Ack is NAck from master

- Clk can be stretched inbetween Ack and Msb of next byte.
  - if slave can't keepup w/ master.

ECE 362 | Lectures 14-15

Computer Abstraction

- Basic Division of Hardware
  - Space Division

  - Time Division
    - fetch, decode, read operand, perform operation, write, det. next instruction
  -

- Instruction Set Architectures (ISAs)
  - Connects Hardware to software w/ assembly
    - we use RISC-V

ISA Basics

- Registers - storage within the core, seperate from memory
  - Word size storage (8-64 bit) storage
  - RISC-V 32I ISA has 32-bit reg. x0-x31
  - Conventions for RISC-V

x	x0/zero - Hard-wired zero
x	x1/ra - return address
x	x2/sp - stack pointer
x	x3/gp - global pointer
x	x4/tp - thread pointer
x	x5-7/t0-2 - temporaries
x	x8/s0/fp - saved reg/frame pointer
x	x9/s1 - saved reg
x	x10-11/a0-1 - Function arguments/return vals.
x	x12-17/a2-7 - Function arguments
x	x18-27/s2-11 - saved registers
x	x28-31/t3-6 - Temporaries

- ISAs have limited, expensive registers
  - Memory solves this. w/ byte addressing
- Addressing Mode - How operand gets its values
  - Register Addressing - read value from register
  - Immediate Addressing - read value from constant in instruction
  - Offset Addressing - read value from memory. Address from imm + reg.

ISA Basics cont'd

- Big endian (MSB @ 0), Little endian (LSB @ 0)
- RisC Instructions are 32 bits or 4 bytes long
  - R-Type (Reg/Reg) – funct7, rs2, rs1, funct3, rd, opcode
    7bit, 5bit, 5bit, 3bit, 5bit, 7bit
    - X opcode partially specs instr., funct7+3 defines operation
    - X rs2 is 2nd operand, rs1 is 1st op., rd is dest. reg
  - I-Type (imm arith + load) – immediate, rs1, funct3, rd, opcode
    12 bits, 5bit, 3bit, 5bit, 7bit
    - x rs1 – source/base address, imm. – const operand or addr offset w/ 2's compliment sign extended
  - S-Type (stores) – imm[11:5], rs2, rs1, funct3, imm[4:0], opcode
    7bits, 5bit, 5bit, 3bit, 5bits, 7bit
    - x rs1: base address added to address, rs2 – source op. reg #, imm – offset split to maintain rs2, rs1 positioning
- RISC can do arithmetic, load 8b, 16b, 32b, shift, and do logical ops. store
- Final uncovered instr type for now:
  - U-Type (upper imm. format) – imm[31:12], rd, opcode
    20bit, 5bit, 7bit
- Examples: (rd, rs1, rs2)
  - add x7, x6, x5
  - lw x8, 0(x9) (rd, imm, rs1)
  - sb x5, 0(x6) (rs2, imm, rs2)

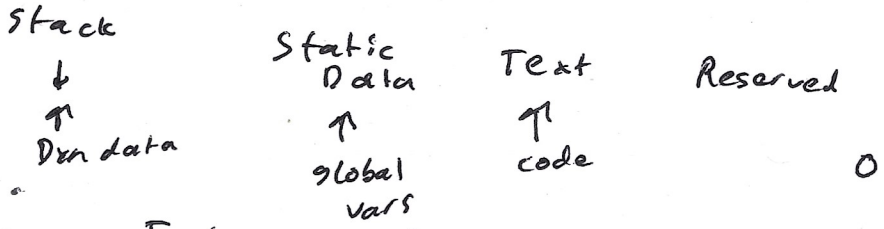
ISA Control Flow

- We can create blocks to jump to
  - block1: ...
    ...
    block2: ...
- Using branch instr. we can jump based on values in regs.
  - bne x0, x5, block1, beq x0, x5, block2 (≠ and =)
  - bge x0, x5, block1, blt x0, x5, block2 (≥ and <)
- Branching can be used for if/else, loops, etc.

ECE 362 | Lecture 17

Procedures

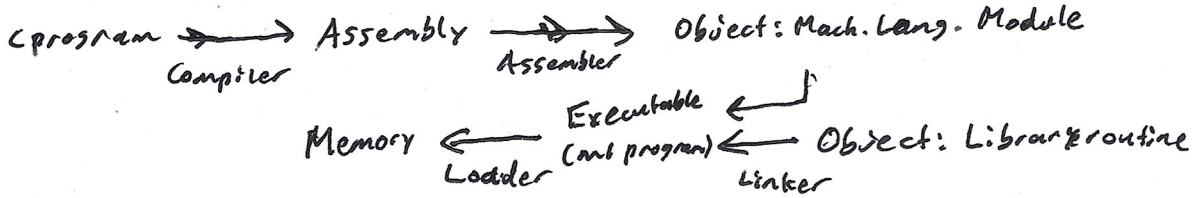
- How to call a function.
  - Pass in arguments w/ x10 - x17, return w/ x10 - x11.
  - Call jal, stores where to ret. to and jumps.
  - After finishing call jalr to load prev addy and jump.
- Since only 8 temp regs exist, use stack in memory to store data.
  - To use stack you subtract 4 for each new reg
  - Use sw/lw and offset w/ the stack pointer
  - add back stack usage at the end.
- Frame pointer shows where each function's data besins
- Memory space division:



- Application Binary Interface (Defines the assembly → binary protocal)

PC-Relative

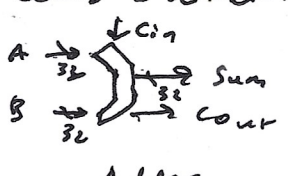
- Translating a C program:



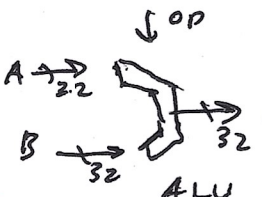
  - ASCII is used to encode string chars.
  - Assembler translates to ml, and provides info for building program.
  - Object Modules produces an executable image that will be loaded later

Single Cycle Processor

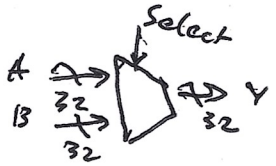
- Datapath (single cycle)
  - uses Muxs, ALU, register files, etc.
    - Comb Elements



Adder

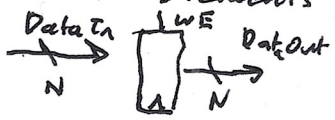


ALU

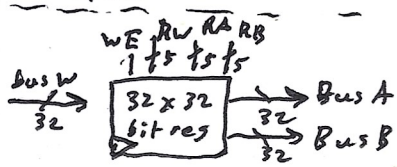


Multiplexer

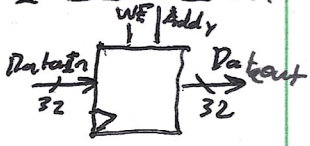
    - Storage Elements



Register

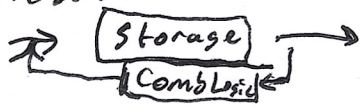


RegisterFile



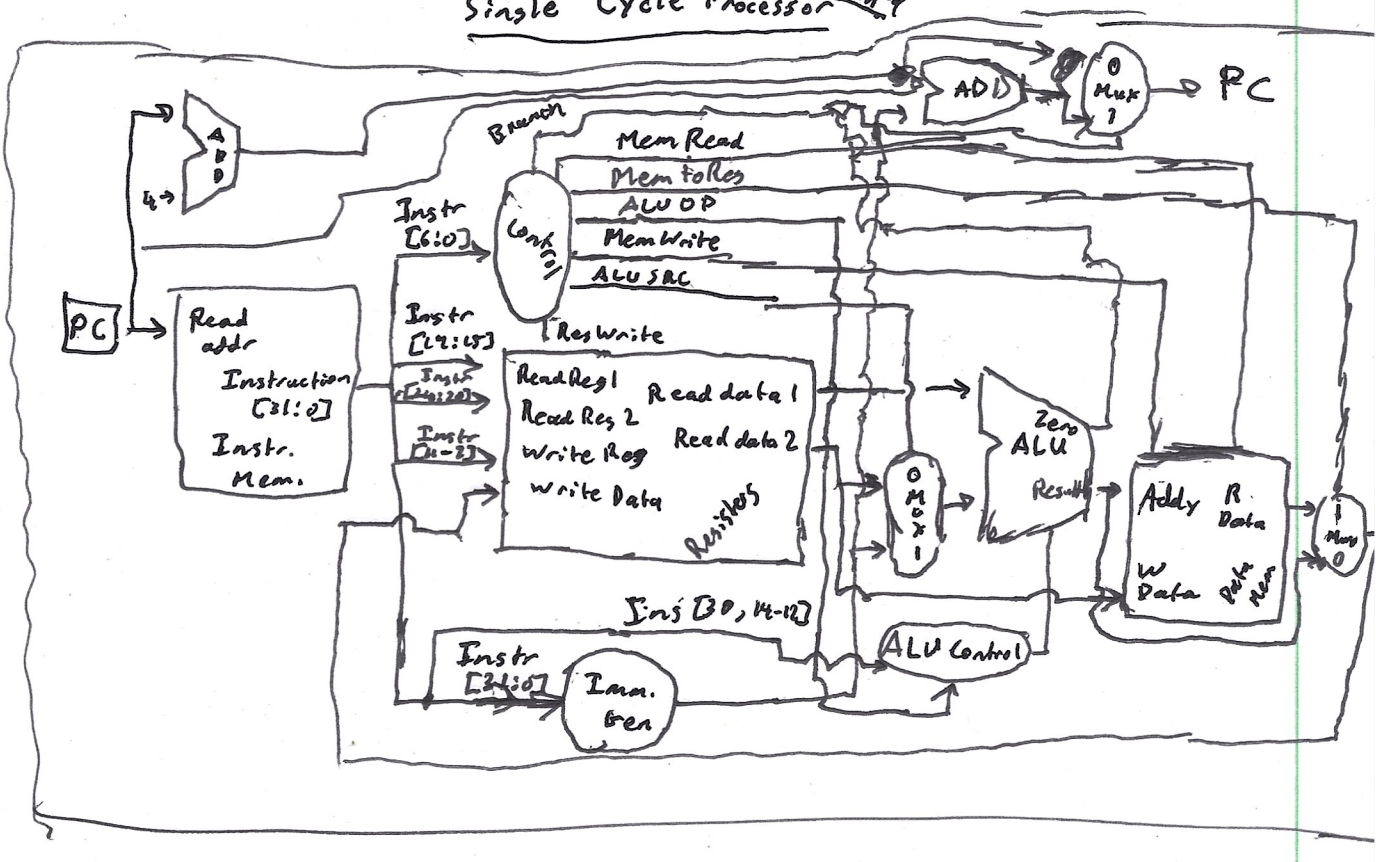
Memory

      - WE - write enable, RW - write to rw, Put ra reg on bus A, Put RB reg on bus B.
  - Computer Outline:

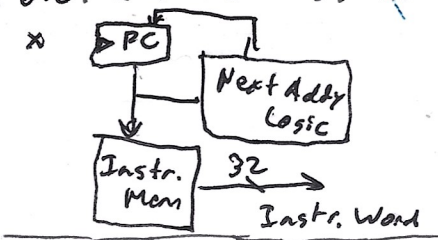


- Processor Implementation
  - Impl. def. Clocks per instruction (CPI) and clock cycle time (Cycle time).
  - ISA + compiler det how many instructions in a program
- Datapath - Single cycle: Get one instruction done in one long clock cycle.
  - steps: Fetch, decode operands, execute, memory, write back

Single Cycle Processor cont'd



- Steps
  - Fetch Instructions; then update PC at end of every cycle



- ALU Instructions: $R[rd] \leftarrow R[rs1]\ op\ R[rs2]$
- Load Instructions: $R[rd] \leftarrow Mem[R[rs1] + SignExt[imm12]]$
- Store Instruction: $Mem[R[rs1] + SignExt[imm11:5 \,||\, imm4:0]] \leftarrow R[rs2]$
- Cond. Branch Instruction: $IR = Mem[PC]$

  if $(R[rs1]\ op\ R[rs2])$

  $PC \leftarrow PC + (SignExt(imm12)\ cc\ 1)$

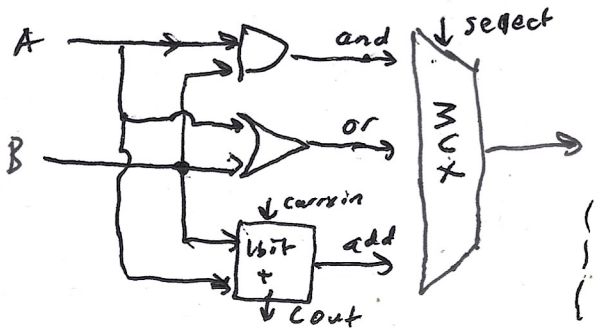
  else

  $PC \leftarrow PC + 4$

ECE 362 | Lecture 21

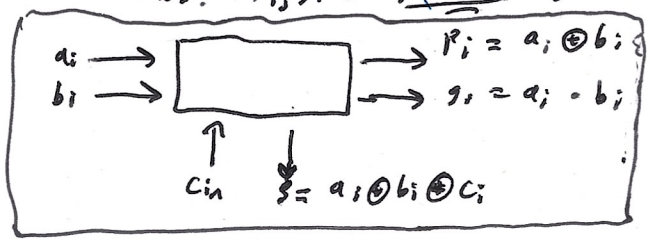
Arithmetic

- Unsigned vs signed
  - Unsigned - normal + range of $[0, 2^{32}-1]$
  - Signed # representation - $[-2^{31}, 2^{31}-1]$ range (All reps have MSB as -)
    - × Sign Mag (MSB is -) $[-3, 3]$, ±0 for 3 bits
    - × Ones Compliment (- is opposite of +) $[-3, 3]$, ±0 for 3 bits
    - × Twos Compliment (invert +1) $[-4, 3]$ for 3 bits
  - × Sign extension copies msb for new bit.
- ALU bit-slice - and, or, add

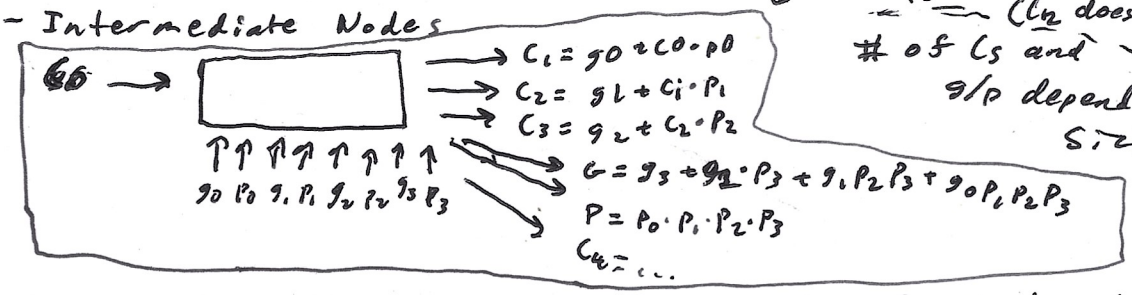


- 1 - For sub, add a signal to xor w/ B.
- Overflow detection w/ xor of MSB carry in & out.
- → Ripple carry adder chains together 1 bit full adders, but is inefficient.

- Carry-lookahead adder - efficient adding. (Propagate or gen carry ahead)
  - Ripple through tree of k-bit blocks (logarithmic vs linear)
  - Leaf Nodes: P_i, g_i = 1 gate delay, S = 2 gate delays



- Intermediate Nodes

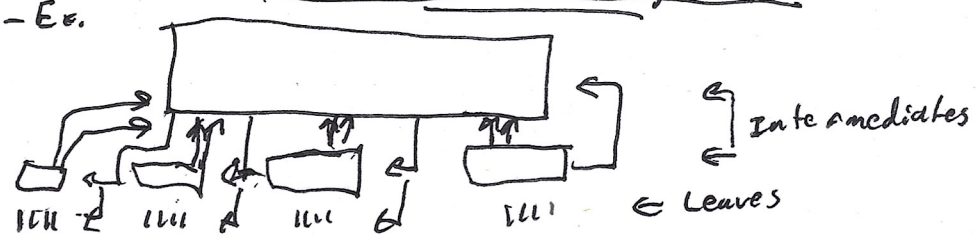


↙ usually expand C expressions to minimize delay (Cn doesn't connect to # of Cs and Ci) g/p depend on block size

Cs only depend on gs & Ps

2 gate delay for Cs, G and P

- Ex.

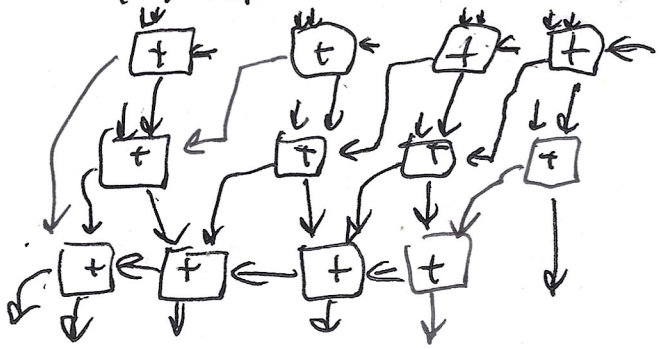


↑ Multi Layer CLA

- Sum bit delay = (bit#/block size) * 2 + 1

Arithmetic cont'd

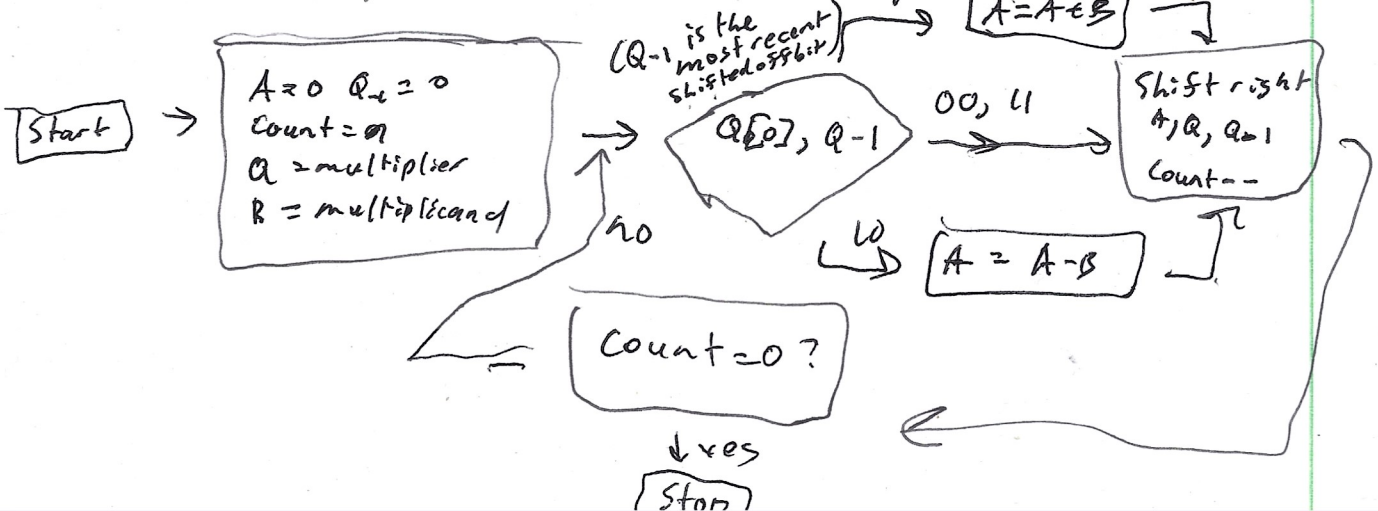
- Shift Logic: uses Multiplexers
  - Mux (shift[0], d, {d[6:0], 0})
  - Mux (shift[1], $tasc0, {stasc0[5:0], 00})
    etc.
- Addins Multople #s
  - Chain adders
  - or use carry save into a Trad adder
    * carry save is 3 nums added (1 in Cin) and Carry is input for next bit a line down



- Multiplication
  - Grade school (how you do multiplication mentally)

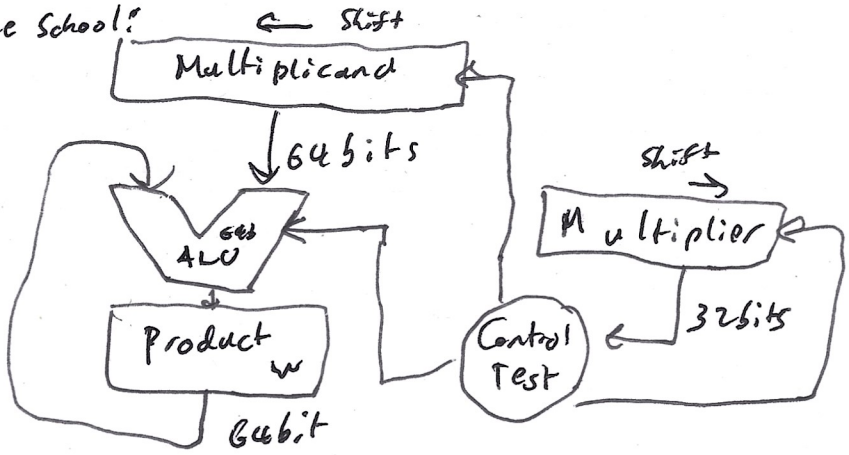
cont'd on next page

- Booth's Algorithm Flow chart

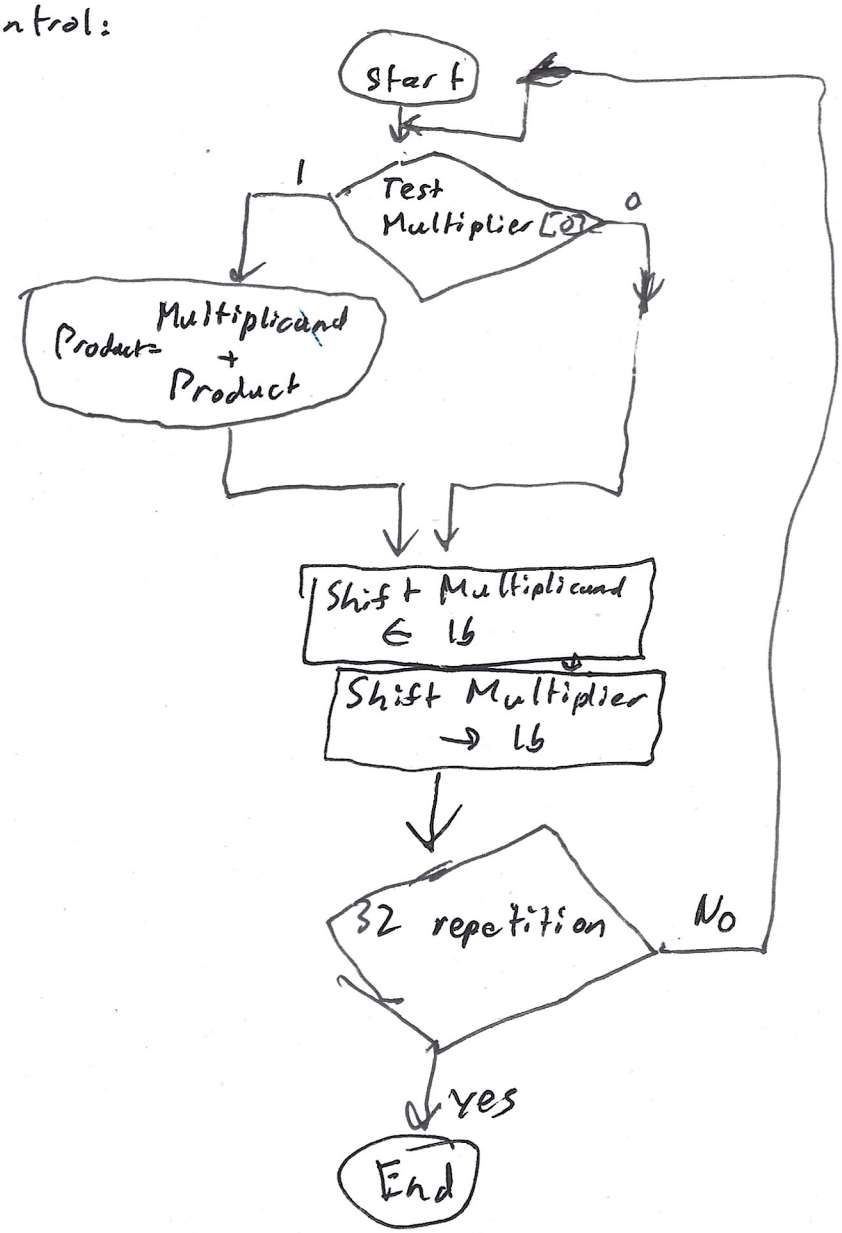


Arithmetic cont'd

- Multiplication
  - Grade School!

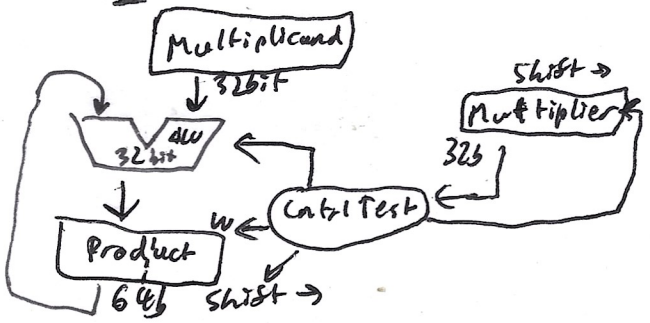


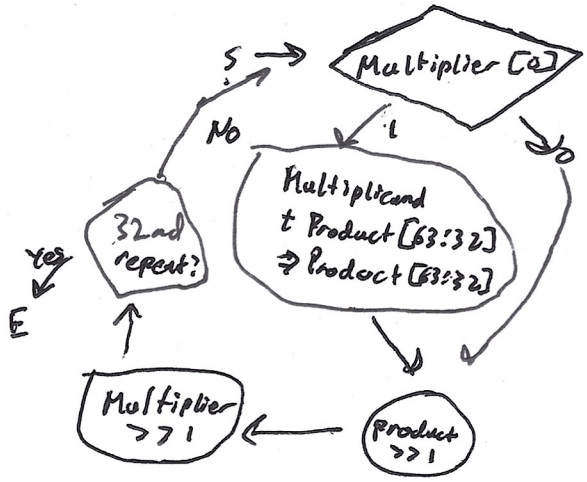
Control:



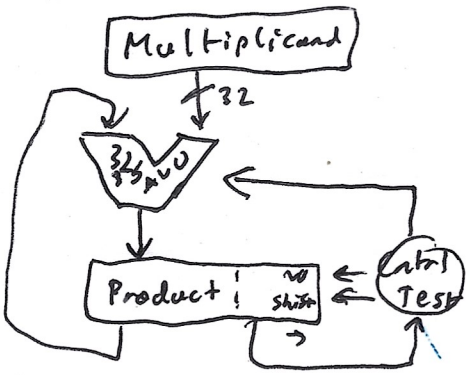
Arithmetic cont'd

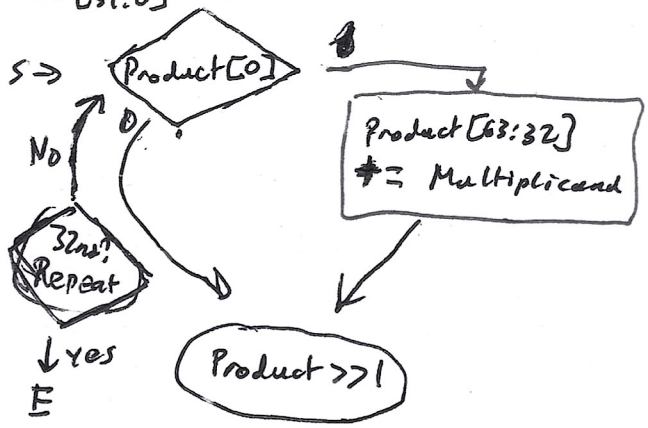
- Multiplication
  - V2:





  - V3 - insert multiplier in product[31:0]

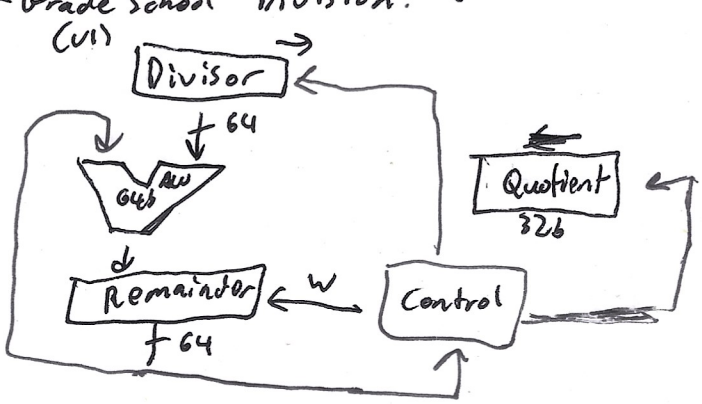


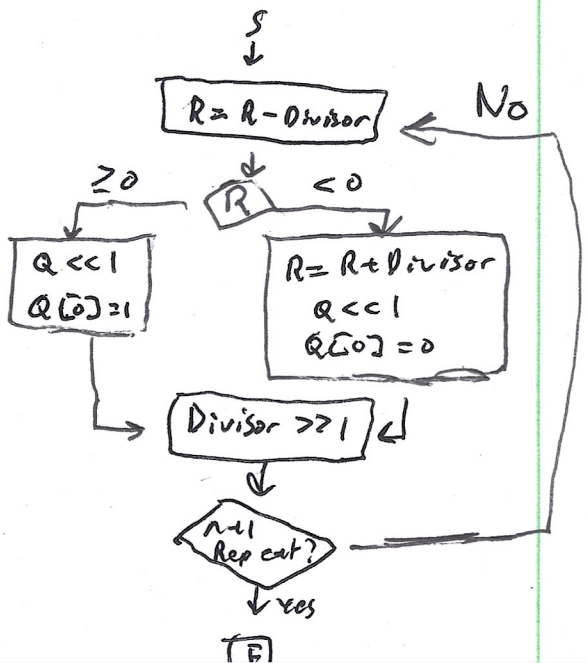


  - Signs - abs val and calc sign after. ~~(not always)~~

- Booth's Algorithm.
  - Booth encoding - go from LSB to MSB, 0→1 = T, 1→1 = 0, 1→0 = 1
  - Multiply while shifting other term left. T means x-1.

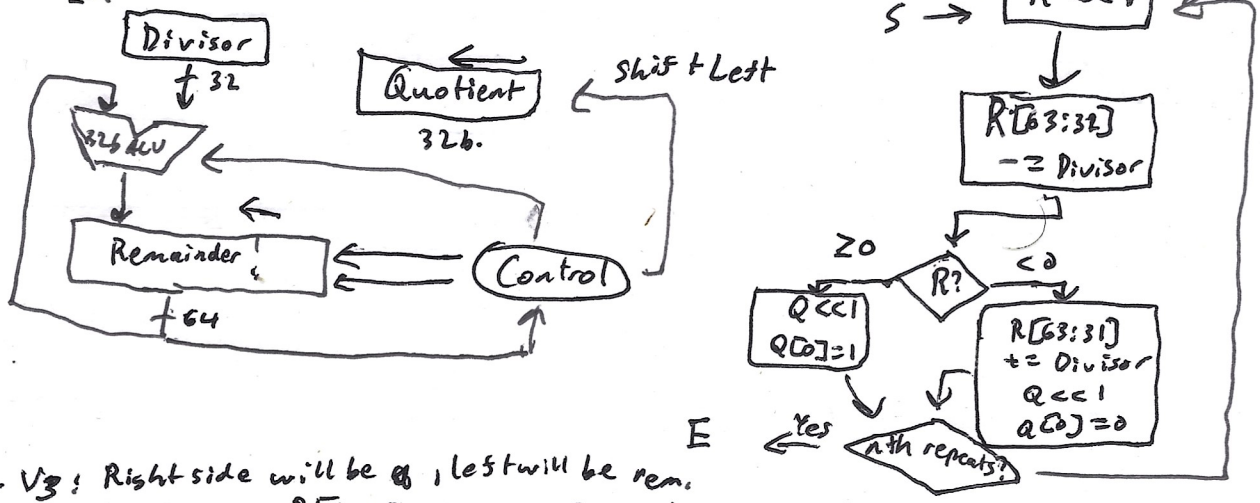
- Division
  - Grade school Division: (Dividend starts in remainder)



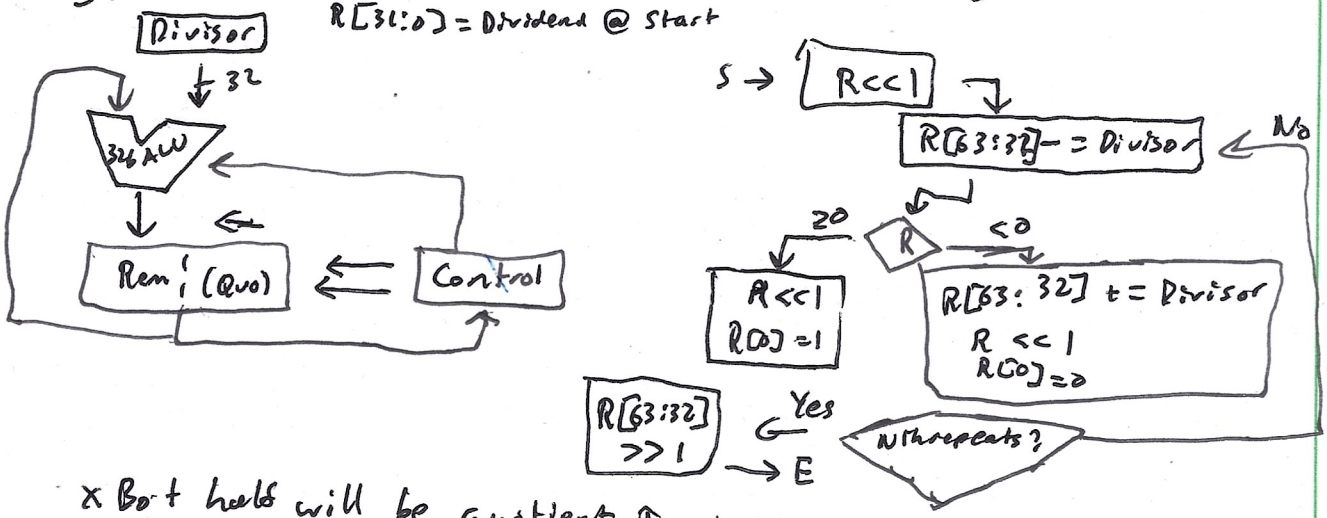


Arithmetic Cont'd

- Division
  - V_2: Rem[31:0] = Dividend to start



  - V_3: Right side will be Q, left will be rem. R[31:0] = Dividend @ start



  - x Bot half will be quotient, Top half will be remainder
  - All these methods are restoring division (undo remainder subtraction)
- Non-restoring Division
  - (+) Divisor
    - x subtract if (+)R, Add if (-)R
  - (-) Divisor
    - x Add if (+)R, Subtract if (-)R
  - If rem/ at end is - add divisor one more time.
- Floating point.
  - IEEE 754 - Store SEF (sign, exponent, float)
    - x $(-1)^S \cdot f \cdot 2^e$
    - x 16 for S
    - x 8b for float, 11 b for double (e)
    - x 23 b for float, 52b for double (f)
  - Further systems are needed for roundings + floating point addition, multiplication, division.