Embedded Processor Synthesis

RISC

Reduced Instruction Set Computer - is a processor architecture that utilizes a small, highly-optimized set of instructions, rather than a more specialized set of instructions often found in CISC.

A RISC-based processor design approach means significantly fewer transistors are required than in a typical CISC.

Such reductions are desirable for light, portable, battery-powered devices.

A simpler design facilitates more efficient multi-core CPUs and higher core counts at lower cost, providing improved energy efficiency for servers.

RISC ~~ra~~ vs CISC

The main characteristics of CISC microprocessors are:

① Extensive instruction sets
② Complex and efficient machine instructions of variable length and variable execution time
③ Micro-coding of the machine instructions
④ Extensive addressing capabilities for memory operations
⑤ Relatively few registers.

In comparison, RISC processors are more or less the opposite

① Reduced instruction set
② Less complex, simple instructions of fixed length, typically 1 clock cycle per instruction.
③ Simple control unit and no microcode.
④ Few addressing schemes for memory operands with only two basic instructions, LOAD and Store.
⑤ Many registers which are organised into a register file.

RISC advantages (2014/15)

① Simple hardware

Because the instruction set of a RISC processor is simple, it uses up much less chip space; extra functions, such as memory management units or floating point arithmetic units, can also be placed on the same chip.

Smaller chips allow a semiconductor manufacturer to place more parts on a single silicon wafer, which can lower the per-chip cost dramatically.

② Short design cycle

Since DISC processors are simpler than equivalent CISC processors, they can be designed more quickly, and can take advantage of other technological developments sooner than corresponding CISC designs, leading to greater leaps in performance between generations.

③ Low power consumption.

Energy has become the primary performance characteristic in embedded designs, especially those aimed at mobile consumer markets. Here RISC processors have a clear advantages over CISC designs: simpler hardware, smaller control logic are features that naturally lead to low energy consumption.

Mlticore

→ Power = C * Vdd^2 * F

Decrease frequency by 50% to have the same performance

→ Power = C * Vdd^2 * F/2

But now Vdd can be reduced by 50% because max frequency is roughly proportional to Vdd.

→ Power = 2 * C * $Vdd^2/4$ * F/2 less power, same frequency

Reasons to use multi-core processor

① Can exploit different types of parallelism
② Can exploit different (more energy efficient) communication mechanisms
③ Simplier cores → more speed
④ Simplier cores → easier to design and test
→ high yield (高产) → lower cost

Why is CISC still around?

Backward 向后
Compatibility 兼容.

ARM big.LITTLE architecture (2013/14)

ARM big-LITTLE is a heterogenerous (异构) computing architecture which ~~computes~~ couples slower, low-power processor cores with more powerful and power-hungry one. The original big LITTLE comprises the Cortex-A7 processor with its more powerful but architecturally ~~compta~~ computible counterpart Cortex-A15. The latest Cortex processors, A53 and A57 cores, are also computible with each other to allow their use in a big.LITTLE chip.

The main advantage is to create multi-core processor that can adjust better to dynamic computing needs and use less power than clock scaling alone. Big.LITTLE employs heterogenous multi-prossing (MP), which enables the use of all physical cores at the same time. Threads with high priority or computational intensity can in this case be allocated to the 'big' cores while threads with less priority or less computational intensity, such as background tasks, can be performed by the 'LITTLE' cores.

[Big.LITTLE energy performance]

The processors look like one multicore CPU to the OS.

User space software on a big.LITTLE Soc is identical to the software that would run on a standard processor.

How does the workload get scheduled?

① ARM has developed the Global Task Scheduling Software to handle automatic allocation of threads.
② GTS is a kernel patch that gives the OS awareness of the big and LITTLE cores, and the ability to schedule individual threads of execution on the appropriate processor based on dynamic - run-time behavior.
③ The software also keeps track of load history for each thread that runs, and uses the history to anticipate the performance needs of a thread the next time it runs.
④ The software reacts quickly to changes in load, and can move work to the big or LITTLE CPU cluster (簇) in less time than a DVFS state transition

RISC

RISC is a microprocessor that is designed to perform a smaller number computer instructions so that it can operate at a higher speed.

☆ About 20% of instructions in a computer did 80% of the work

[Characteristic]

① Small instruction set {More complex operations are performed using a sequences of simple instr
② Fixed length instructions {May be fetched in a single operation
③ 1 clock-cycle instruction

Pipelining

Pipelining is a design ~~technology~~ technique where the computer's hardware processes more than one instruction at a time, and doesn't wait for one instruction to complete before starting the next.

Typical four stages: fetch, decode, execute and write

[RISC Disadvantage]

① Code quality

The performance of a RISC processor depends greatly on the quality of the code that it is executing.

RISC processors are harder to program efficiently than their CISC equivalents. If the instruction scheduling in a program is poor, the processor can spend quite a long time stalling: wait for the result of one instruction before it can proceed with a subsequent instruction.

Instruction scheduling rules can be complicated in RISC processors, hence the performance of a RISC application depends critically on the quality of the code generated by the compiler.

② System Design

They require more instructions, and hence memory, than CISCs to implement applications.

RISC processors require very fast memory systems to feed them instructions. RISC-based systems typically contain large memory caches, usually on the chip itself.

1-bit (Serial) Processors

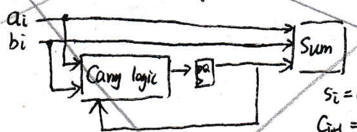
Several advantages that might be useful in many-core context.

- small physical size
- less power, also overall energy for computation might be less

Advantages:

① Extreme simplicity
② Very low power consumption
③ Many-core 1-bit systems can be reconfigurable easily.

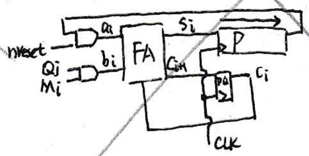
Single-bit serial adder for multi-bit addition



$S_i = a_i \wedge b_i \wedge C_i$

$C_{i+1} = a_i \& b_i | a_i \& c_i | b_i \& C_i$

1-bit multiplication



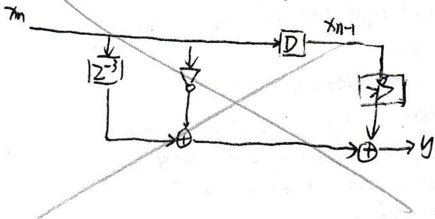
Use n^2 clock cycles

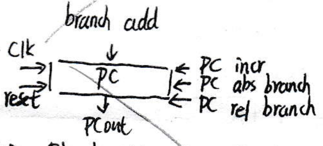
steps for M x Q

```
① Clear product P {nreset = 0
                   {Qi = 0
② for j=0 to n-1,
     for i=0 to n-1:
        clock carry register
              Product register
```

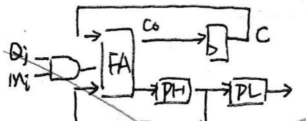
Bit-serial filter

$y(n) = -x(n) + x(n) \times ^{-3} + x(n-1) \cdot 2^{-1}$





```
logic ··· Rbranch // temp variable for addition operand
comb
   if (PC incr)
      Rbranch = 1
   else
      Rbranch = BranchAddr.
ff
   if (reset)
      PCout <= '0'
   else if (PC incr | PCrelbranch)
      PCout <= Pcout + Rbranch;
   else if (PCabsbranch)
      PCout <= BranchAddr.
```



P = Q x M

Bits PH[0], PH[1] ... are fed to the Full Adder via a MUX or by rotating PH

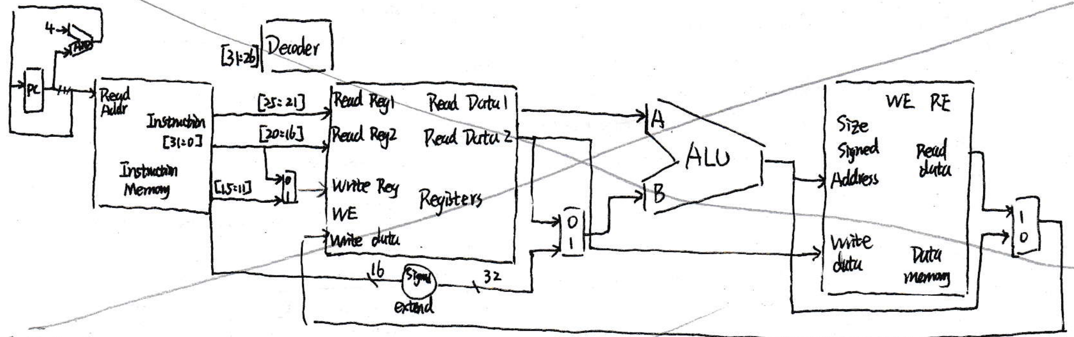
C needs to be shifted into PH via the FA or ~~the~~ a MUX

The ARM thumb instruction set is a subset of the most commonly used 32-bit ARM instructions.

Thumb instructions are each 16 bits long, and have a corresponding 32-bit ARM instruction that has the same effect on the processor model.

MIPS Architecture

[Basic MIPS Architecture]



- 32-bit instructions, 6-bit opcode (bits 31:26)
- 32 general purpose registers
- RAM data memory
- ALU operands: 3 registers or 2 registers + immediate operand

[Instruction formats]

Type						
					shift amount field	
R	opcode (6)	rs(5)	rt(5)	rd(5)	shamt (5)	funct (6)
I	opcode (6)	rs(5)	rt(5)	immediate (16)		
J	opcode (6)	address (26)				

- Instructions are divided into three types: R, I and J
- Every instruction starts with a 6-bit opcode
- R-type instructions specify three registers, a shift amount field, and a function field
- I-type instructions specify two registers and a 16-bit immediate value;
- J-type contain a 26-bit jump target address

[MIPS Pipeline]

① Fetch (F)
- Read next instruction from memory, increment address counter
- Assume 1 cycle to access memory

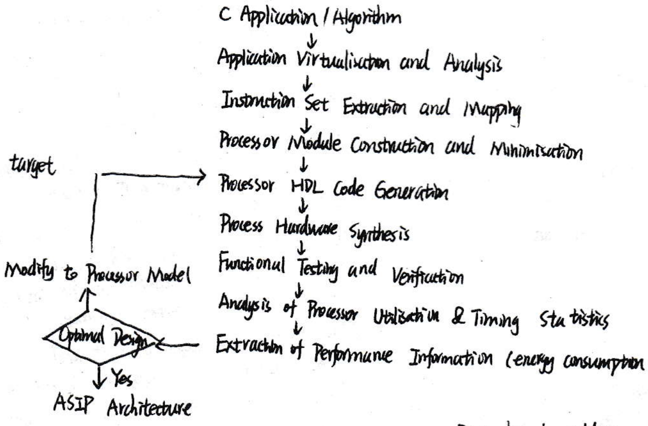
② Decode (D)
- Read register operands, resolve instruction in control signals, compute branch target

③ Execute (E)
- Execute arithmetic/resolve branches

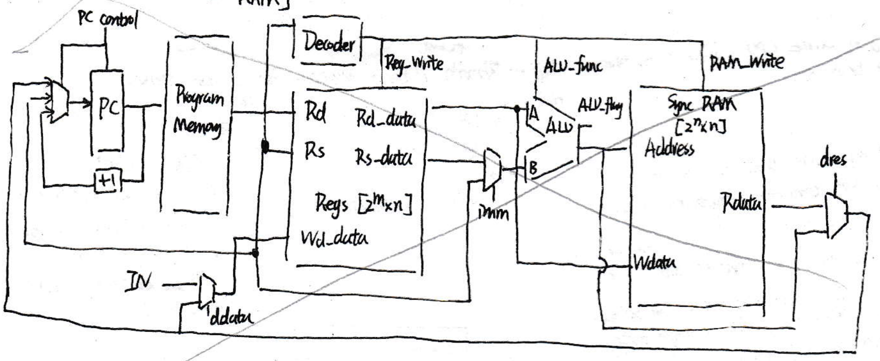
④ Memory (M) - Load/Store instructions only
- Perform load/store access to memory
- Access 1 cycle to access memory

⑤ Write back (W)
- Write arithmetic result to register file

[Processor synthesis flow]


C Application / Algorithm
Application Virtualization and Analysis
Instruction Set Extraction and Mapping
Processor Module Construction and Minimization
Processor HDL Code Generation
Process Hardware Synthesis
Functional Testing and Verification
Analysis of Processor Utilization & Timing Statistics
Extraction of Performance Information (energy consumption, size)
Optimal Design
Modify to Processor Model
Yes
ASIP Architecture

[picoMIPS with branches and RAM]



Immediate Jump { PC ⇐ branch address Absolute Jump
PC <= PC + imm Relative Jump (One Adder)

Indirect Jump { PC <= Rd_data

Conditional Jump { PC <= condition ? branch address : PC+1 Absolute
PC <= condition ? PC + imm : PC+1 Relative

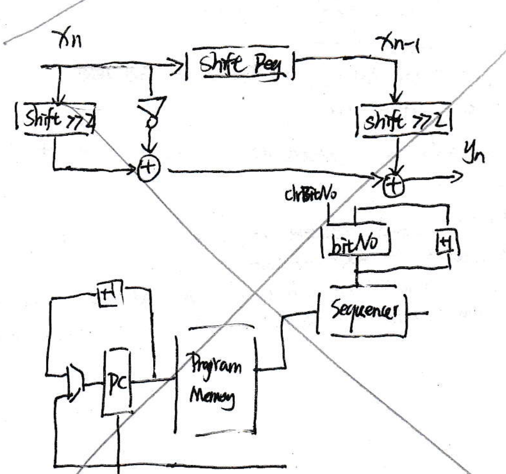
[RAM instructions]

① LW - load word from RAM
- Syntax LW %d, %s, imm
- Operation: %s = RAM [%d + imm]

② SW - store word in RAM
- Syntax SW %d, %s, imm
- Operation: RAM [%d + imm] = %s

Load must be performed in two clock cycles if the RAM is synchronous.

{ ① LW1 - selects registers, program ALU and clocks RAM
② LW2 - writes RAM output into registers



```
`Store: { RAMwrite = '1;   // Write to RAM
          ALUfunc = `RB ;  // ALU outputs memory address from its port B
          Imm = '1;        // Select immediate for port B, RAM address
`LOAD1: { ALUfunc = `RB;
          Imm = '1;
`LOAD2: { ALUfunc = `RB;
          Imm = '1;
          ddata = '1;
          ~~RAM~~ Regs write = '1;   // Write to general purpose register
```

Processor Synthesis

[ALU Carry logic]

A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1 ←
1	0	0	1	0
1	0	1	0	1 ←
1	1	0	0	1 ←
1	1	1	1	1 ←

When coding addition as A+B
We don't know Cin at MSB. but we know Sum[7]

Carry is set at MSB slice if:

A[7]&B[7] | B[7] & ~Sum | A[7] & ~Sum[7]

A-B, complement B[7]

[Detection of 2's complement overflow]

A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0 ←
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1 ←
1	1	1	1	1

Overflow occurs in addition if the signs of both operands are the same, but the result sign is different.

A[7]&B[7]& ~Sum[7] | ~A[7]&~B[7]&Sum[7]

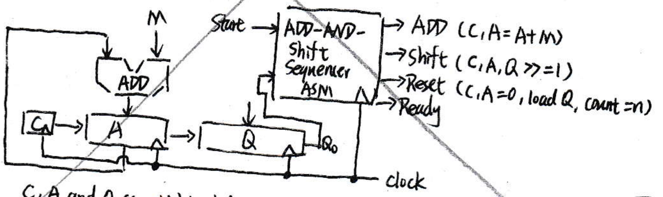
A-B, complement B[7]

Multiplier

For sequential hardware implementation

M = 111 (7)

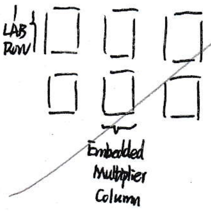
C	A	Q=5	
0	000	101	Initial Value
0	111	101	ADD: A:=A+M
0	011	110	Shift
0	001	111	Shift
1	000	111	ADD: A:=A+M
0	100	011	Shift



C, A and Q can right shift by one bit

Embedded multipliers in FPGAs

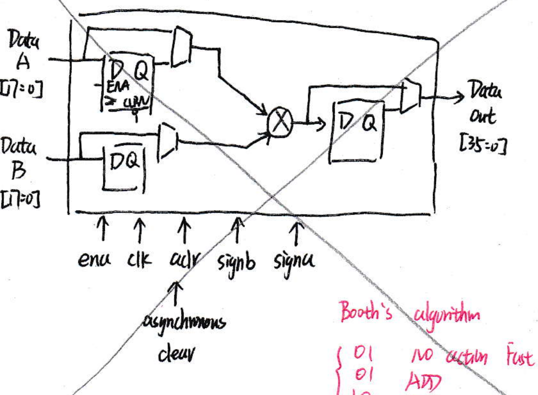
Altera Cyclone: multipliers arranged in columns sourrounded by Logic Array Blocks (LABs)



Each multiplier is configuarted as one 18x18 multiplier or two 9x9 multipliers

For multiplications greater than 18x18, multiple multipliers are blocked together, There is no restriction on the data width

[Multiplier block architecture]



Booth's algorithm

01	No action Fast
01	ADD
10	SUB
11	No action

FPGA Processor Cores

Can be classified as 'Hard' or 'Soft'

- 'hard' core is an embedded processor, e.g. ARM, surrounded by FPGA's memory and programmable logic
- 'soft' core is synthesised from HDL code using standard FPGA's memory and logic

[Soft cores]

~~Using~~ Use existing FPGA logic elements
Flexible, feature rich, reconfigurable
Customized memory size, data path width, ALU functionality, number and types of peripherals
Typically have slower clock rates

[NIOS II windowed register file]

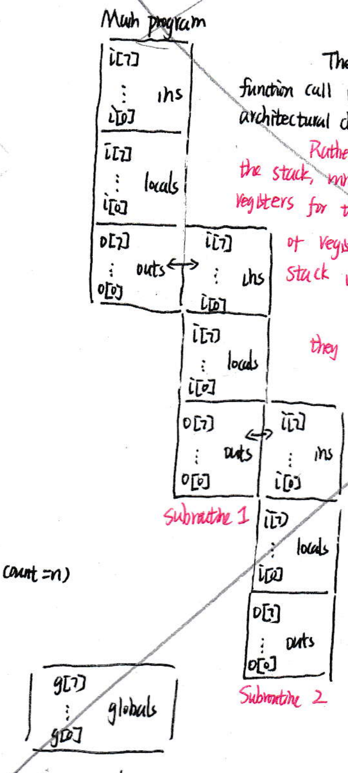
Common Technique used by High-performance CPUs
- Provide fast subroutine calls

Up to 512 general-purpose registers
Movable window with access to 32 registers
- 24 register window (movable)
- 8 global registers (fixed)

Automatically used by C compiler.

(Wiki)

Every part of a program wants registers for its own use, several sets of registers are provided for the different parts of the program



The overhead associated with saving registers to the stack during conventional function call was believed to be very large, or at least significant enough to warrant architectural changes to speed this process.

Rather than wasting valuable CPU cycles to copy register data to and from the stack, windowed register aims to ensure that a function call gets a private set of registers for the duration of the function. When the function completes, the previous set of registers return to existence with (in most cases) no interaction with the stack whatsoever.

The registers are divided into four groups based on the sort of data they are to contain, according to windowed registers:

① global registers: for ~~common~~ data common across function calls.
② Input registers: for incoming function parameters (including the frame pointer and return pointer.
③ Local registers: for general use
④ Output registers: for parameters to deeper called functions, the return value from deeper function calls, the stack pointer, and the saved program counter after a jump and link.

①, ②, ③ comprises a register window

```
//unsigned multiplier
module unsigned_mult (output logic [15:0] out, input logic [7:0] a,b);
    assign out = a*b;
endmodule

// signed multiplier
module signed_mult ( output logic signed [15:0] out, input logic signed [7:0] a,b );
    assign out = a*b;
endmodule

//signed multiplier
module signed_mult (output logic [15:0] out, input logic [7:0] a,b, input logic clk);
    logic signed [7:0] a_reg, b_reg;
    logic signed [15:0] mult_out;
    assign mult_out = a_reg * b_reg;
    always_ff @ (posedge clk)
    begin
        a_reg <= a;
        b_reg <= b;
        out <= mult_out;
    end
endmodule.
```

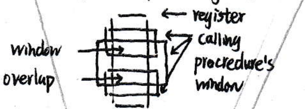
[multiply-add hardware] a*b+ c*d

```
always_ff
    if (aclear)
        { a_reg, b_reg, c_reg, d_reg, mult0, mult1, out <= 10;
    else
        { a_reg <= a;
          ...
          mult0 <= a_reg * b_reg;
          ...
          out <= mult0 + mult1;
```

[Windowed Register] Detailed

When a function is called, it allocates a new window for its specific use. The global registers are shared between the old and new window (meaning that any modification of global data in the callee will be visible in the caller). The callee receives a new group of local registers, as well as a new set of output registers - these registers are not accessible from the calling function.

Finally, the caller's output registers are rotated to be the input registers for the call function. Any changes the callee should make to its input registers will be visible to the caller as changes in the caller's output group of registers.



In this way parameters can be passed from one function to another without (usually) interaction with the stack. The caller's code need only put parameters in its output registers, then call a function.

The called function will have access to the caller's output registers in its own input registers. Return values are the reverse of this process; the called function leaves the return value in a particular input register, which then reverts to being an output register for the caller as soon as the function returns.

Nested function calls will create a chain of linked register window. Each function call will use the same group of eight global registers, but will have its own group of eight local regs for its own private use. The output registers from the first function will be the input registers for the second deeper function called; the outputs from the second will be the inputs for the third, and so on.

Obviously, this trend can't go on forever. Each register window involves 24 registers (8 input, 8 local, 8 output), a third of which are shared with the calling function and two thirds of which need to be allocated by the processor. (The global registers are not shifted). The processor will only have a limited number of register available - most modern processors provide enough for seven or eight windows.

Typical applications of NIOS

USB port, Ethernet, UART, Parallel I/O port
SPI ports, Timers

Pipelined CPUs

[Key features of RISC and pipelines]

- Small and simple instruction set
- Most instructions execute in single clock cycle (no microcode)

(在CISC结构下，一些复杂功能的指令执行时，被分解为一系列相对简单的指令来执行，这样的一系列简单执行就被 微程序)

- Memory access instructions only transfer data : mem ↔ reg
- ALU operations are register to register
- Large number of general purpose registers
- Piplines are not always used in embedded architectures

Hardware overheads are necessary to implement and control pipeline.

[Pipeline hazards]

- Instruction cycle broken into n phases (one execution per phase) e.g. Fetch, Decode, Read OPs, Execute 1, Execute 2, WriteBack.
- A new instruction is fetched up at each phase.
- Maximum speed gain is n times
- Pipeline hazards reduce the ability to achieve a gain of n times

  Types of Hazards
  - Resources
    - Hazards occur when instruction needs a resource being used by another instruction
  - Data
    - RAW (Read after Write)
    - WAR (Write after Read)
    - WAW (Write after write)
  - Control
    - Hazard occurs when a wrong fetch decision at a branch results in an extra instruction fetch and a pipeline flush.(冲洗)

[Data hazards]

- Dependences between instructions may cause data hazards when $Instr_1$ and $Instr_2$ are so close that their overlapping(重叠) within the pipeline would change their access order to Reg.
- Three types of hazards

① Read After Write (RAW)
  $Instr_2$ tries to read operand before $Instr_1$ writes it

② Write After Read (WAR)
  $Instr_2$ tries to write operand before $Instr_1$ reads it

③ Write After write (WAW)
  $Instr_2$ tries to write operand before $Instr_1$ writes it

[Solutions to data access and data dependency hazards]

Software solutions (Compiler scheduling)

- Putting no-op instructions after each instruction that may case a hazard.
- Instruction scheduling : rearrange code to reduce no-ops

Hardware solutions

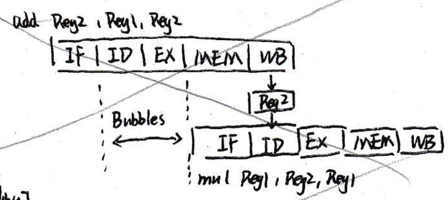
- Hazard detection hardware is necessary
- Interlocking : Stall pipeline for one or more cycles
- Forwarding : two types

  ① The ALU result of $instr_1$ in EX stage can immediately be forwarded back to ALU input of EX stage as an operand for $instr_2$

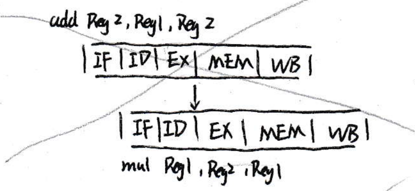
  ② The memory load data from MEM stage can be forwarded to ALU input of EX stage.
- Forwarding with interlocking

  Assuming that $Instr_2$ is data dependent on the load instruction $Instr_1$ then $Instr_2$ has to be stalled until the data loaded by $Instr_1$ becomes available.

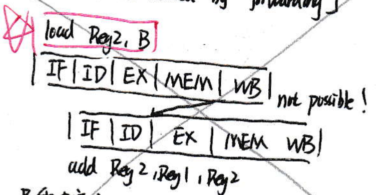
[Hazard solution to data dependency hazard - interlocking]



[Forwarding]

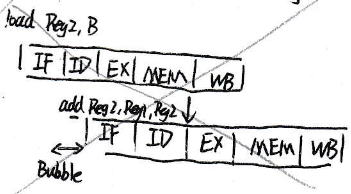


[Hazard can't be resolved by forwarding]



B的内容在MEM里

⇓ Resolved by forwarding and stalling



Control hazards

Control hazards occur when a wrong fetch decision results in a new instruction fetch and the pipeline being flushed.

Dealing with control hazards may require significant hardware overheads.

[Solutions]

- Multiple Pipeline streams
- Prefecting the branch target
- Using a Loop Buffer
- Branch Prediction
- Delayed Branch
- Reording of Instructions
- Multiple Copies of Registers (backups)

RISC-V features

- Small CPU
  - may have 16 registers
  - Basically version includes 33 instrs
- Idea borrowed from SPARC
  - Register 0 is a constant 0
- Ideas borrowed from MIPS
  - No status register
  - Conditional branches evaluate branch condition in the same cycle.
  - Memory mapped I/O
- Novel ideas
  - ALU intentionally lacks condition codes, branch conditions are evaluated by separate logic
  - Extra space is reserved for new instructions future extensions but also user defined instructions