CounterFlow Pipeline Processor (CFPP) Architecture

A Seminar By Marco Della Torre

Scope

- The CounterFlow Pipeline Processor (CFPP) Concept
- CFPP Architectural Features
- Overview of Operations
- Concept, not case study
- Life after CFPP : Rotary Pipeline

The Brains Behind CFPP

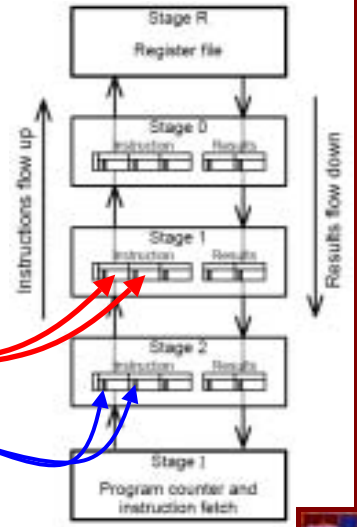
- Invented by Robert Sproull and Ivan Sutherland
- Also known as a Sproull Pipeline Processor
- Proposed in 1994
- Proposed as a “family” of microarchitectures
- Microarchitecture : internal processor implementation

CFPP : What’s New?

- Two pipelines which interact with each other – instructions flow “up” and results/operands flow “down”
- Either Single or multiple instruction issue depending on implementation
- Execution may occur out of issue order
- The register file has added significance
- Very different notion of a pipeline stage

Basic Organisation

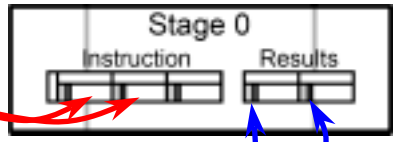
- Relocated register file
- Instructions move “up”
- Results move “down”
- Issue logic at “R” stage determines what is sent down the results pipeline
- Stages contain storage
- Storage units have validity tags



Stage Contents

- Instruction stages store information in bindings
- Each binding stores the intended operation, register names, data and a validity bit.
- The validity bit indicates whether the association (between register name and data) is valid

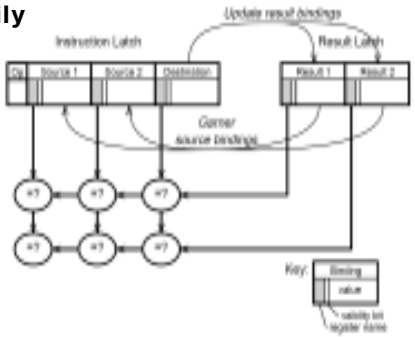
Instruction Pipeline: Three bindings



Results Pipeline: Two bindings

Operands and Execution

- Instruction stages also contain functional units
- Functional units are spread over the stages.
- A stage need not necessarily contain every type of functional unit present in the pipeline
- Operands are taken from the “downward” flowing results pipeline



Operands and Execution

- Each stage may or may not contain an instruction
- Once executed, the results is stored in the destination binding, marked valid and can proceed to the register file.
- The result is also entered into “downward” flowing pipeline binding
- The downward flowing pipeline is a forwarding mechanism
- At the top of the pipeline, destination bindings are committed to the register file
- Until committed, an instruction in the pipeline can be cancelled

Operands and Execution

- The register file is the primary source of bindings issued in the results pipeline, since it provides instruction operands.
- Algorithms for issuing operands must be carefully selected to maximise throughput
- If the register file does not rovide the correct operands in the results pipeline, an instruction may stall the pipeline indefinitely
- Bindings are also provided by executed instructions, which require only local logic to ensure the correct values are passed down the results pipeline

Pipeline Rules

- No overtaking : instructions proceed “up” the pipeline in order
- Algorithms for issuing operands must be carefully selected to maximise throughput
- If an instruction's operand bindings are valid and the stage it is in contains the required functional unit for it to execute, it may do so. Once executed, the result is entered into the destination binding, which is then marked valid
- Once an instruction is executed, at least one copy of its destination binding is entered into the results pipeline
- An un-executed instruction cannot pass the last stage able to execute it. Therefore, an un-executed instruction must stay in the pipeline until executed

When Pipeline Bindings Match

- If a valid result binding matches an invalid source binding, copy the result value to the source value and mark the source valid.
- If an invalid destination binding matches a valid result binding, mark the result binding invalid.
- If a valid destination binding matches a result binding, copy the destination value into the result value and mark the result valid.
- If an invalid destination binding matches a valid result binding, mark the result binding invalid.

Operation Example

$$\begin{array}{ll} PC = 101 & A := B + C \\ PC = 102 & B := A + B \\ PC = 103 & D := C - 1 \end{array}$$

Stage	*Instruction pipe*	*Result pipe*	*Remarks*
R			Registers contain: $A[14]B[2]C[3]D[21]$
0			
1			
2			
1	$PC = 101$ ↑		Fetch, send source names B, C to reg file

Operation Example Continued

Stage	*Instruction pipe*	*Result pipe*	*Remarks*
R		$B[2]C[3]\downarrow$	Registers contain: $A[14]B[2]C[3]D[21]$
0			
1			
2	$A[] := B[] + C[]\uparrow$		
I	$PC = 102\uparrow$		Fetch, send source names A, B to reg file

Stage	*Instruction pipe*	*Result pipe*	*Remarks*
R		$A[14]B[2]$	Registers contain: $A[14]B[2]C[3]D[21]$
0		$B[2]C[3]$	Musta't swap with instruction below.
1	$A[] := B[] + C[]\uparrow$		Musta't swap with result above.
2	$B[] := A[] + B[]\uparrow$		
I	$PC = 103$		Fetch delayed due to cache miss.

Operation Example Continued

Stage	*Instruction pipe*	*Result pipe*	*Remarks*
R		$A[14]B[2]$	Registers contain: $A[14]B[2]C[3]D[21]$
0	$A[] := B[2] + C[3]$	$B[2]C[3]\downarrow$	Garner B, C; execute
1	$B[] := A[] + B[]$		
2			
I	$PC = 103\uparrow$		Fetch, send source name C to reg file

Operation Example Continued

Stage	*Instruction pipe*	*Result pipe*	*Remarks*
R		$A[14]B[2]$	Registers contain: $A[14]B[2]C[3]D[21]$
0	$A[5] := B[2] + C[3]\uparrow$	$A[5]$	Insert result
1	$B[] := A[] + B[2]\uparrow$	$B[2]C[3]$	Garner B
2	$D[] := C[] - 1\uparrow$		Literal −1 held in binding value
I	$PC = 104$		Fetch, send source names to reg file

Stage	*Instruction pipe*	*Result pipe*	*Remarks*
R	$A[5] := B[2] + C[3]\uparrow$	$A[]B[2]$	Registers contain: $A[5]B[2]C[3]D[21]$
0	$B := A[5] + B[2]\uparrow$	$A[5]$	Garner A, execute
1	$D := C[3] - 1\uparrow$	$B[2]C[3]$	Garner C, execute
2	. . .		
I	$PC = 105$		Fetch, send source names to reg file

Handling Traps

- The instruction that generates the trap sends a special trap binding down the results pipeline, thereby affecting subsequent instructions
- When the special trap binding reaches the “bottom” of the pipeline, it is interpreted as such by the program control logic
- When setting the trap or branch binding, information about the instruction can also be held, such as its program counter value

Handling Branches

- Similar to traps
- When a branch instruction enters the pipeline, it has the condition code register as a source
- The processor makes a branch prediction and continues to issue instructions up the pipeline
- If the prediction is correct, execution is not altered
- If a misprediction occurs, the branch instruction sends a wrong-branch-result binding down the results pipeline, which disables subsequent (now invalid) instructions.

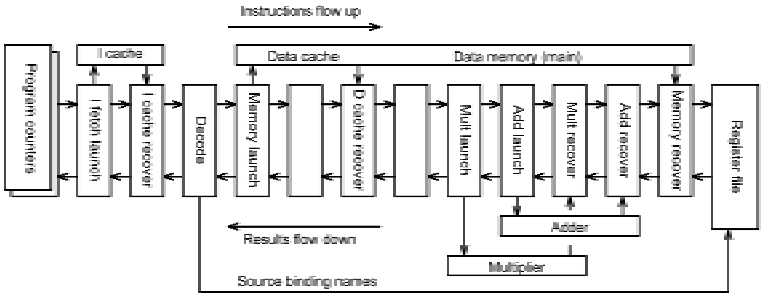
Handling Branches

- When the result reaches the control logic at the bottom, the program counter is adjusted to the correct value.
- Stages such as the instruction decoder and program control logic can also be separated by stages, with required communication occuring via the results pipeline.
- Last Pipeline Rule : If a result binding is either trap-result or wrong-branch-result, mark the instruction invalid. The instruction may proceed up the pipeline, but it will have no side effects on the results pipeline or the register file.

Functional Units

- Different stages can be capable of different types of processing
- Multiple-cycle operations may be implemented as sidings
- Instructions “drop off” the operands and retrieve the result further up the pipeline. Can be a functional unit or specialised coprocessor
- This procedure removes the need for stalling while waiting for lengthy operations to complete, if sidings are pipelined
- An invalidated instruction using a siding must still retrieve the results in order to ensure the sidings and pipeline are coordinated

Functional Units



Register Files and Caches

- Multiple register files can be supported, for example a floating point register file located after all the floating point functional units and an integer register file after all integer units
- Any instruction which may alter a register file must execute before passing it, so the results are not lost
- To reduce the latency involved in fetching instructions from instruction register and passing them down the pipeline, register caches can be used
- Locating a register cache above the instruction decoder enables bindings to be filled by the results pipeline

Register Files and Caches

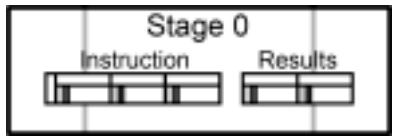
- Source bindings are filled by the register cache if the register has a valid entry in the cache
- Results which pass the cache are entered as valid
- Any instruction passing the register cache which may alter a valid entry must invalidate it.
- Traps and mispredicted branch instructions must invalidate cache entries modified by instructions after the trap or branch
- Rather than keep track of such instructions, the entire cache is swept clean.

Register Files and Caches

- Requests for source operands need only be issued when required registers are not valid in cache
- This reduces the number of operand requests sent to the register file
- The path from the register file to the decode and register cache stages is long.
- Communications are pipelined and the register cache is considered as a regular pipeline stage, governed by the pipeline rules, in order to speed up the connection between the ends of the pipeline
- Cache bindings are then maintained by the pipeline rules

Proposed Implementations

- Many things can be modified – number of stages, number of sidings, where sidings pick up operands and return results, caching, etc
- Terminology : “Packet” refers to the bindings held at a stage in either the instruction or results pipeline. In the first example, the packet size of the results pipeline was two
- The number and contents of the bindings in the instruction pipeline bindings and results pipeline bindings can be varied for different implementations
- If the instruction pipeline has a packet size greater than one binding, then the implementation is like a conventional multi-scalar design



Proposed Implementations

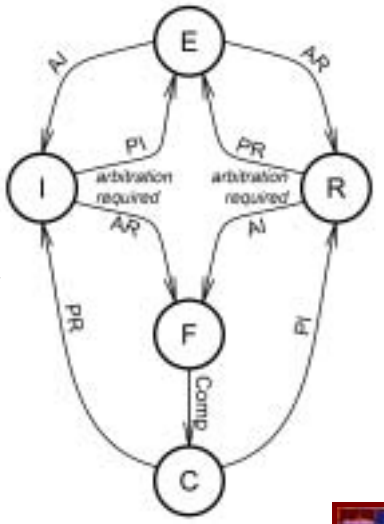
- Local control is used between stages, since only local knowledge is required for transferring data between pipelines and for stalling instructions
- Proposed control described as “elastic” - i.e. the number of packets in the pipeline can vary (within limits). Packets can be removed or added from the result pipeline
- To transfer packets from stage to stage, space is required at the destination stage. Maximum throughput therefore occurs when a pipeline is half full (No need to wait for the above stage to copy binding out first)
- If the instruction pipeline has a packet size greater than one binding, then the implementation is like a conventional multi-scalar design

Overview of Control

- At each stage:

E = Empty
I = Instruction Present
R = Result Present
F = Both Instruction and Result
C = Pipeline rules have been enforced and both the instruction and result are ready to move on.

AI = Accept Instruction
AR = Accept Result
PR = Pass Result
PI = Pass Instruction



Some advantages of CFPP

- Simple, regular structure
- Register feed-forwarding
- Branches and traps easy to manage
- Speculative execution and misprediction correction easy
- Flexibility and freedom of design

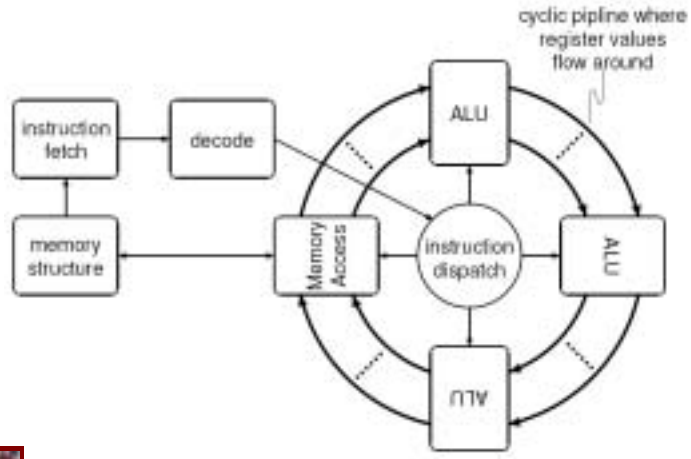
Some disadvantages of CFPP

- Complex arbitration
- Register must provide the best contents possible in the result pipeline bindings to maximise throughput
- Multiple instruction issue adds a large amount of dependency checking logic
- Stalled instruction can stall the entire pipeline indefinitely

Rotary Pipeline

- The results pipeline is converted to a loop so that registers circulate around the stages of the outer pipeline
- Pipeline stages are characterised by the functional unit stored there rather than data held there. Operands are sourced from the circulating registers
- This eliminates the need to stall if an instruction reaches the last stage able to execute it, since there is no longer any last stage.
- Registers/results are not kept in lock-step, but lapping is not allowed in order to maintain consistency
- If data dependencies exist, execution is prepared and begins as soon as the data is available

Rotary Pipeline Organisation



Rotary Pipeline Organisation

- Only the required registers are passed around the pipeline
- The number of required registers is proportional to the number of functional units
- This means that rotary pipeline processors can be very large structures if many functional units are used, since support for each “loop” must be added
- Sequential instructions are issued in the same direction as register pipeline flow
- If a register holds condition codes, it can be passed around as a special type of register to pass the signal to each stage

Rotary Pipeline Execution

- The pipeline has to be stalled if no appropriate stages (functional units) are available
- Speculative execution can either be achieved by postponing committing the result until the speculation is confirmed or by writing to temporary registers
- Sequential instructions are issued in the same direction as register pipeline flow
- If an exception is raised, instructions must be cancelled behind the current instruction, but care must be taken not to cancel instructions at another point in the loop which is actually ahead of the instruction.

Rotary Pipeline Execution

- In order to know when a stage has completed execution, a completion signal must be maintained, or the executions must be matched with delays
- Stage control is implemented in a similar fashion as in the CFPP example earlier, with states indicating what data is present and the current execution progress


Rotary Pipeline Aims

- Reduce delays down to only data rates only (remove stalling)
- Localise control
- Avoid clock synchronisation issues : self-timed logic
- Facilitate multiple instruction issue in a simple fashion
- Exhibit good exception handling and speculative execution properties
- Justify large structures required to maintain the architecture


Thank You!

References to appear on website….