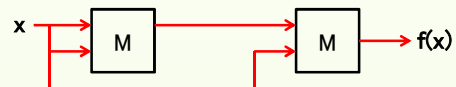


Basic Pipelining



Primer on pipelining

- Consider the cubing function: $f(x) = x^3$
 - We can write $f(x) = M(M(x, x), x)$ where $M(x, y) = x \cdot y$
 - This decomposition of f allows us to implement f with two serially connected multipliers



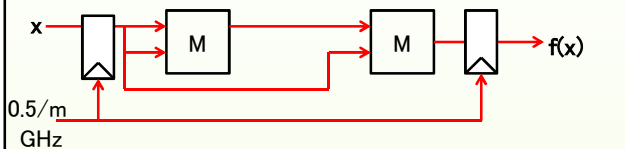
- Suppose the multiplier latency is m ns
 - How frequently can a new input be sent to this hardware?
 - How frequently should the output be sampled?

MAINAK CS422

2

Primer on pipelining

- Combinational cube computation



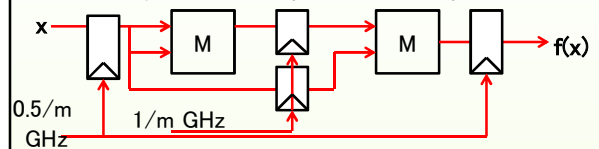
- Frequency is $0.5/m$ GHz, throughput is one output every $2m$ ns
- Observation: exactly one multiplier is active at any point in time
 - Why not use just one multiplier?
 - Will require some sequencing logic and offer slightly worse throughput at almost half area cost

MAINAK CS422

3

Primer on pipelining

- Multi-cycle cube computation (conceptual)



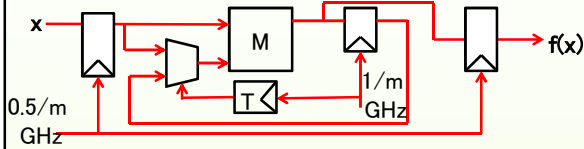
- Frequency is $0.5/m$ GHz, throughput is one output every $2m$ ns; in reality, throughput will be slightly worse
- Half of the work is wasted unless each multiplier is gated in alternate internal cycles
- Solution: use one multiplier

MAINAK CS422

4

Primer on pipelining

- Multi-cycle cube computation (physical)



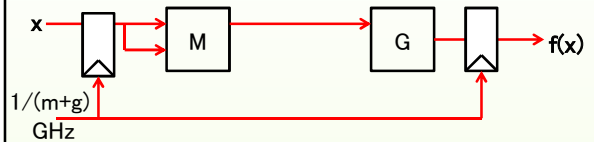
- Multi-cycle design makes sense only if there is a chance of reusing hardware so that we can save area
 - Otherwise we lose in throughput and increase area compared to a combinational design
 - Consider $f(x) = G(x*x)$; latency of G is g ns

MAINAK CS422

5

Primer on pipelining

- Combinational implementation



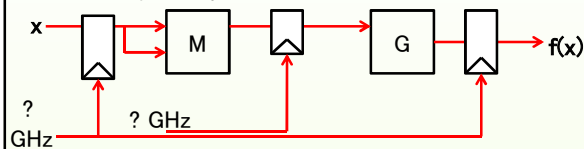
- Frequency is $1/(m+g)$ GHz, throughput is one output every $(m+g)$ ns

MAINAK CS422

6

Primer on pipelining

- Multi-cycle implementation



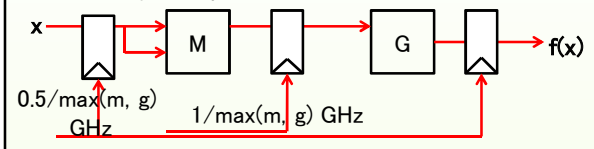
- What is the throughput of this implementation?
- Can we fold this implementation to save area?

MAINAK CS422

7

Primer on pipelining

- Multi-cycle implementation



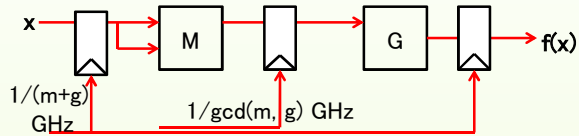
- Lot of wasted work and energy
- No improvement at all over combinational design
 - Worse in all departments: throughput, area, energy

MAINAK CS422

8

Primer on pipelining

- Multi-cycle implementation with same throughput as combinational implementation
 - Can't be any better without pipelining



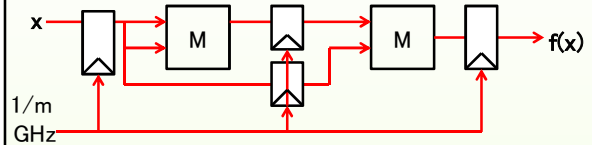
- Lot of wasted work and energy due to frequent switching of the register at the center

MAINAK CS422

9

Primer on pipelining

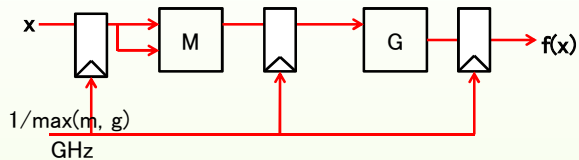
- Pipelined cube computation



- Frequency is $1/m$ GHz, throughput is one output every m ns; in reality, throughput will be slightly worse
- Fundamental requirement of pipelining
 - Should be able to decompose the function to be computed into a series of functions
 - $f(x) = f_1 \circ f_2 \circ f_3 \circ \dots \circ f_k(x)$
 - Ideally one would expect a k times faster clock and k_0 times higher throughput

Primer on pipelining

- Pipelined $G(x*x)$ computation



- Frequency is $1/\max(m, g)$ GHz, throughput is one output every $\max(m, g)$ ns; in reality, throughput will be slightly worse, but still better than combinational implementation

MAINAK CS422

11

Primer on pipelining

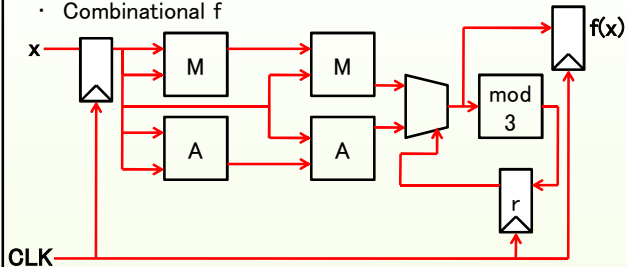
- Imagine feeding a series of inputs x to a k -stage pipelined implementation of some computation $f(x)$
 - Usually, one would expect the computation on two inputs to be independent of each other
 - True for stateless or memoryless functions
 - Consider the following function with a global state r
- $f(x)$ returns y
- if $(r == 0)$ $y = x * x * x$
 else $y = x + x + x$
 $r = y \bmod 3$
- How to pipeline this function?
 - Assume two multipliers and two adders
 - Computation of one input depends on the previous one
 - Pipeline hazard

MAINAK CS422

12

Primer on pipelining

- Combinational f

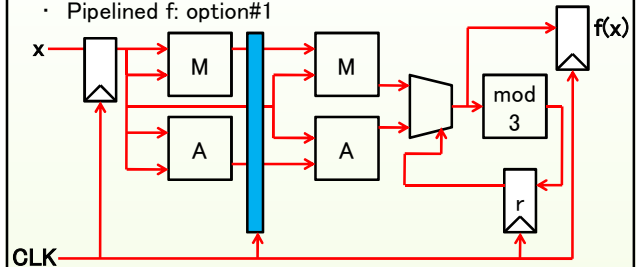


- Latency: $2 * \max(m, a) + \text{mux} + (\text{mod})$
 - Determines throughput and I/O sampling rate
- Lot of wasted work, but somewhat easy to pipeline

13

Primer on pipelining

- Pipelined f: option#1



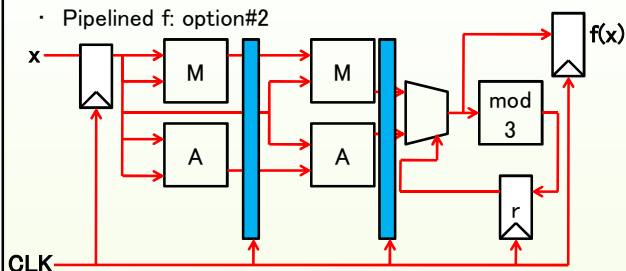
- Does it produce correct result?
- What about performance?

MAINAK CS422

14

Primer on pipelining

- Pipelined f: option#2

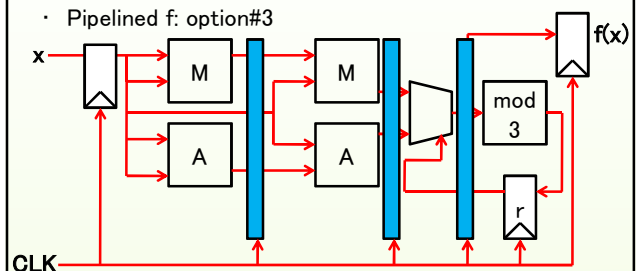


- Does it produce correct result?
 - Reading from and writing to the same register in the same pipe stage
- What about performance?

15

Primer on pipelining

- Pipelined f: option#3



- Does it produce correct result?
 - A back edge crosses a pipeline register

MAINAK CS422

16

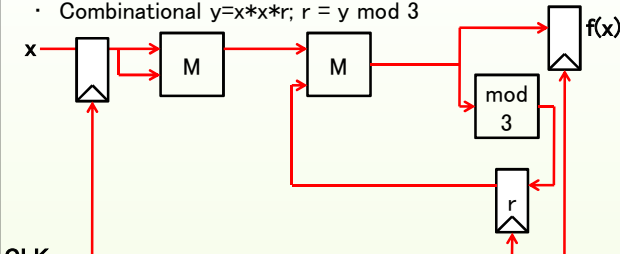
Primer on pipelining

- Another hazard of different nature
 $f(x)$ returns y
 $y = x * x * r$
 $r = y \bmod 3$
- The nature of computation is known, but the required data may not be available
- Another similar example
 $f(x)$ returns y
 $y = x * r * r$
 $r = y \bmod 3$
- How to pipeline this one?

MAINAK CS422

17

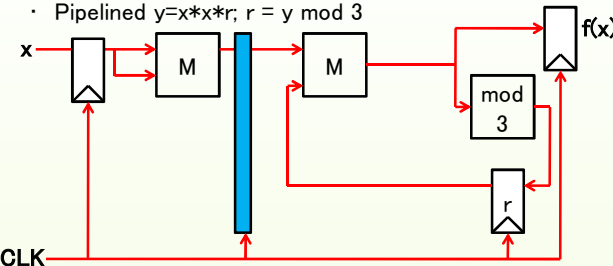
Primer on pipelining

- Combinational $y = x * x * r$; $r = y \bmod 3$
- 
- Latency: $2 * m + (\bmod)$
 - Determines throughput and I/O sampling rate

MAINAK CS422

18

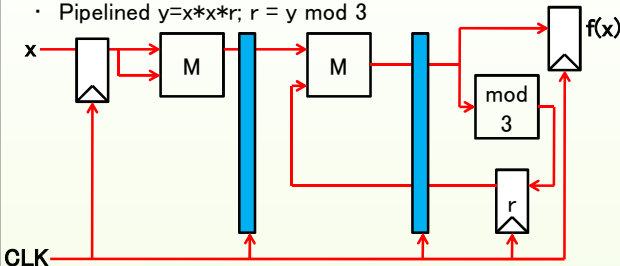
Primer on pipelining

- Pipelined $y = x * x * r$; $r = y \bmod 3$
- 
- Are we good?

MAINAK CS422

19

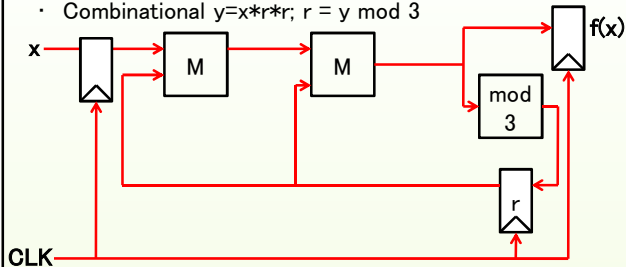
Primer on pipelining

- Pipelined $y = x * x * r$; $r = y \bmod 3$
- 
- Are we good?
 - Hazard rule: source stage of a data comes after the destination stage
 - No easy solution that can win back the lost performance²⁰

20

Primer on pipelining

- Combinational $y = x * r * r$; $r = y \bmod 3$



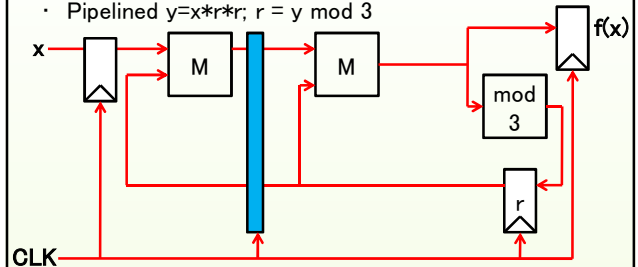
- Latency: $2 * m + (\bmod)$
 - Determines throughput and I/O sampling rate

MAINAK CS422

21

Primer on pipelining

- Pipelined $y = x * r * r$; $r = y \bmod 3$



- Are we good?

MAINAK CS422

22

Primer on pipelining

- A somewhat easier to pipeline computation

$f(x)$ returns y

$$y = x * r * r$$

$$r = x \bmod 3$$

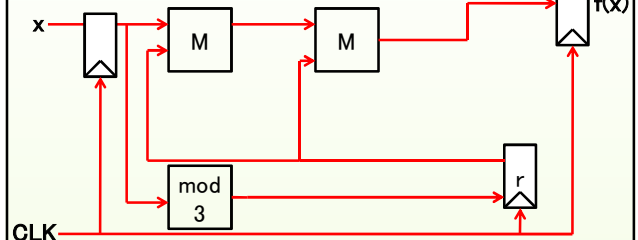
- Notice that the computation of one x still depends on the previous computation
- Can we pipeline it?

MAINAK CS422

23

Primer on pipelining

- Combinational $y = x * r * r$; $r = x \bmod 3$



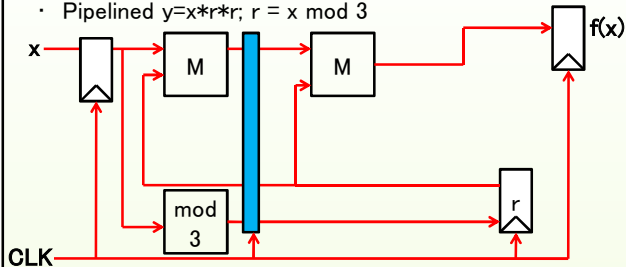
- Latency: $\max(2 * m, \bmod)$
 - Determines throughput and I/O sampling rate

MAINAK CS422

24

Primer on pipelining

- Pipelined $y = x * r * r$; $r = x \bmod 3$



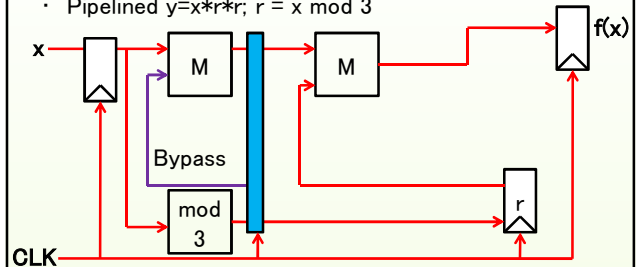
- Are we good?

MAINAK CS422

25

Primer on pipelining

- Pipelined $y = x * r * r$; $r = x \bmod 3$

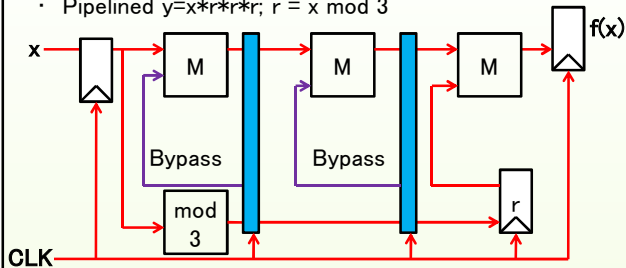


- Are we good?
 - Source stage of data comes before (or equal to) the destination stage even if the data is written to storage element at a later stage
- Can be handled with a bypass path

26

Primer on pipelining

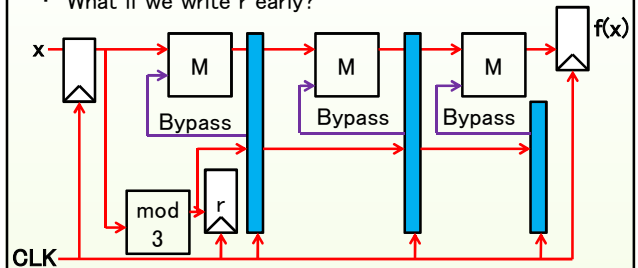
- Pipelined $y = x * r * r * r$; $r = x \bmod 3$



- Keep bypassing from pipeline registers until the value is written to storage element
 - In a particular stage S, the register before it contains r produced by the current computation and the register after it contains r produced by the previous computation

Primer on pipelining

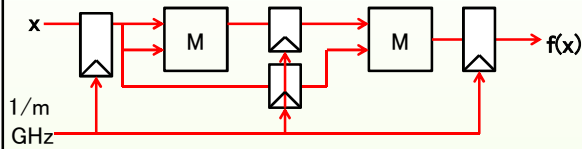
- What if we write r early?



- In the first stage, could read directly from r
- For later stages, need to carry r forward
 - In a particular stage S, the register before it contains r produced by the current computation and the register after it contains r produced by the previous computation

Primer on pipelining

- How to simulate the pipelined cube computation?



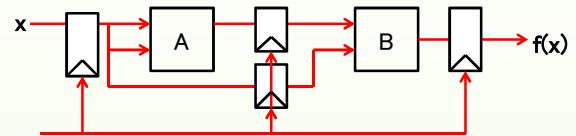
- A two-stage pipe: call them M1 and M2
- In a sequential simulator, in which order would you simulate these to capture the correct behavior within each cycle?

MAINAK CS422

29

Primer on pipelining

- How to simulate a general pipelined computation?

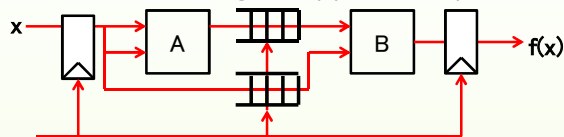


- Suppose the latency of B depends on the value of A(x) and the maximum latency of B is much higher than the fixed latency of A
 - Highly non-deterministic
 - Clocking the pipe at $1/\max(a, b)$ offers the worst-case performance always
 - What if I clock it at $1/a$? Replace registers by queues
 - How to properly simulate such a pipeline?

30

Primer on pipelining

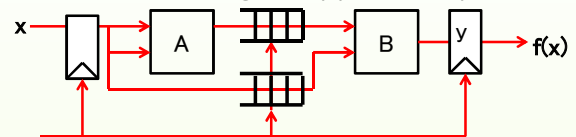
- How to simulate a general pipelined computation?



- In a cycle, we first simulate B and then A
 - while (1) { B(...); A(...); cycle++; }
- Suppose the queue between A and B is full in some cycle
- B is invoked; it dequeues one element and processes it
- When A is invoked after B, it sees one queue slot free and therefore, enqueues one new element
 - This is an incorrect simulation; in a real system, A would not be able to enqueue into a full queue in this cycle

Primer on pipelining

- How to simulate a general pipelined computation?



```
while (1) {
    On posedge {
        Sample queue states (full/empty) and store internally;
    }
    On negedge {
        if (!sampled empty queue) dequeue b from head; y = B(b);
        if (!sampled full queue) a = A(new input x); enqueue a at tail;
        Update full/empty state of queue;
        cycle++;
    }
}
```

32

Pipelining a processor

- Input to a processor pipeline is program counter
 - Each pass through the pipeline involves a new PC
- Processor pipeline is not stateless
 - Each instruction modifies the state in a different way
 - Each instruction depends on the state in a different way
 - All instructions may not be independent
 - This is what makes processor pipelining difficult
- Caveat
 - I will focus primarily on a pipelined implementation of the 32-bit MIPS ISA
 - Different kinds of delay slots are very specific to MIPS
 - Not present in other processors or x86
 - Basic pipelining principles will hold across different ISAs

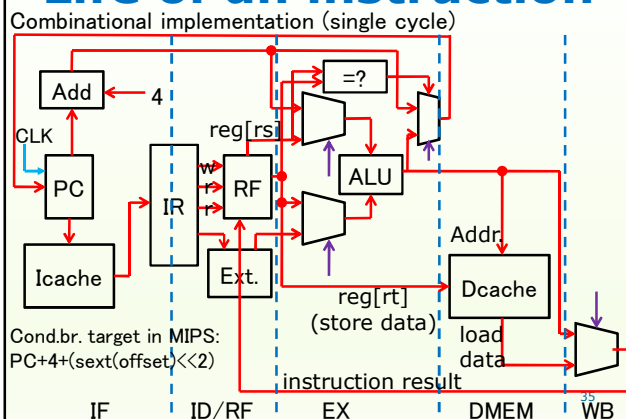
Life of an instruction

- A RISC instruction can normally be completed in five phases
 - Instruction fetch (IF): Fetch the instruction pointed to by program counter (PC) and increment PC to point to the next instruction
 - Instruction decode / register fetch (ID/RF): Decode fetched instruction, send the source register addresses (rs and rt) to the register file and fetch the contents
 - Execute (EX): Compute addresses for memory operations, compute ALU operations; target for branches
 - Memory access (DMEM): Load/store instructions access memory for reading or writing
 - Writeback (WB): Send the result of computation and the destination register address to the register file

MAINAK CS422

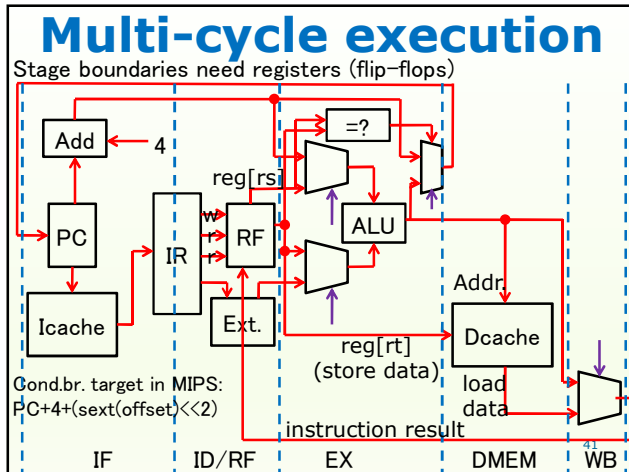
34

Life of an instruction



Single-cycle design

- Possible to implement all five stages as a big combinational logic
 - The input address to the memory elements can be presented any time within a cycle and the output data is available within the same cycle
 - Register file (in stages 1 and 4), icache (in stage 0), and dcache (in stage 3) are the memory elements
 - PC, icache[PC], reg[rs], reg[rt], dcache[sext(displ)+reg[rs]] are inputs
 - dcache[sext(displ)+reg[rs]] is a relevant input for loads
 - PC, reg[rd] or reg[rt], dcache[sext(displ)+reg[rs]] are outputs
 - dcache[sext(displ)+reg[rs]] is a relevant output for stores
 - The instruction opcode and function (for R format) fields decide the control of combinational logic e.g., which register operands are valid sources, which ALU op to invoke, etc.



Multi-cycle execution

- Example
 - IF: 2 ns, ID/RF: 1 ns, EX: 1 ns, DMEM: 3 ns, WB: 1 ns
 - Branch frequency: 20%
 - Store frequency: 10%
 - Multiply/divide frequency: 5%, latency: 30 ns
 - Total instruction count: 100
- Multi-cycle
 - Cycle time: 3 ns, frequency: 333 MHz
 - CPI: $0.2*3+0.1*4+0.05*10+0.65*5 = 4.75$
 - Execution time: $100*4.75*3 \text{ ns} = 1425 \text{ ns}$
- Single-cycle
 - Cycle time: 8 ns, frequency: 125 MHz
 - CPI: $1*0.95+\text{ceil}(30/8)*0.05 = 1.15$
 - Execution time: $100*1.15*8 \text{ ns} = 920 \text{ ns}$

42

Multi-cycle execution

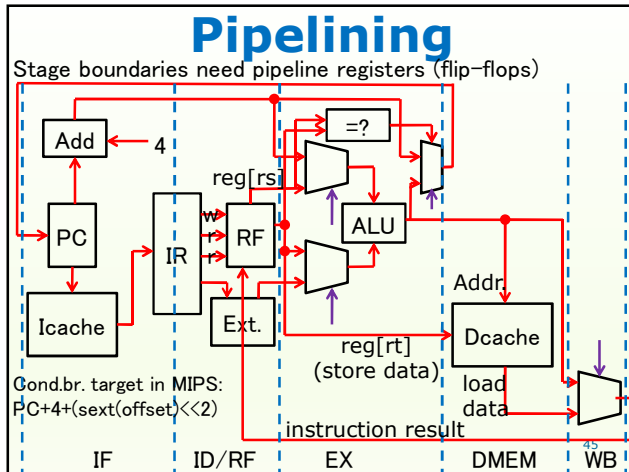
- Example with a balanced pipeline
 - IF: 2 ns, ID/RF: 2 ns, EX: 2 ns, DMEM: 2 ns, WB: 2 ns
 - Branch frequency: 20%
 - Store frequency: 10%
 - Multiply/divide frequency: 5%, latency: 30 ns
 - Total instruction count: 100
- Multi-cycle
 - Cycle time: 2 ns, frequency: 500 MHz
 - CPI: $0.2*3+0.1*4+0.05*15+0.65*5 = 5$
 - Execution time: $100*5*2 \text{ ns} = 1000 \text{ ns}$
- Single-cycle
 - Cycle time: 10 ns, frequency: 100 MHz
 - CPI: $1*0.95+\text{ceil}(30/10)*0.05 = 1.1$
 - Execution time: $100*1.1*10 \text{ ns} = 1100 \text{ ns}$ (worse than multi-cycle)

Pipelining

- Observations
 - In the second cycle, I know if it is a branch; if not, start fetching the next instruction?
 - When the ALU is doing an addition (say), the decoder is sitting idle; can we use it for some other instruction?
 - In summary, exactly one phase is active at any point in time: wastes hardware resources
- Form a pipeline
 - Process five instructions in parallel
 - Each instruction is in a different stage of processing (called pipe stage)
 - How to synchronize between pipe stages?

MAINAK CS422

44



Pipelined design

- Individual instruction latency is five cycles, but ideally can finish one instruction every cycle after the pipeline is filled up
 - Ideal CPI of 1.0 at the clock frequency of multi-cycle design
 - Execution time is ideally one-fifth of the multi-cycle design
 - Instruction throughput improves five times (number of instructions completed in a given time)

10	IF	ID/RF	EX	MEM	WB
11		IF	ID/RF	EX	MEM
12			IF	ID/RF	EX
13				IF	ID/RF
14					IF

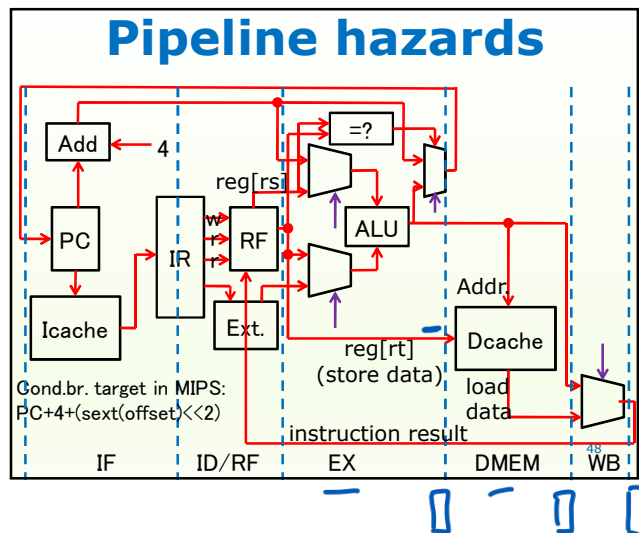
time →

46

Gain and loss?

- Parallelism
 - Extracting parallelism from a sequential instruction stream: known as instruction-level parallelism (ILP)
 - Can complete one instruction every cycle (ideally)
 - Ideally, CPI is five times smaller
- Loss
 - Each pipe stage may get lengthened a little bit due to control overhead (skew, setup time): limits the gain due to pipelining
 - Each instruction may take slightly longer
 - Resource conflicts? (RISC vs. CISC)
 - Bigger memory bandwidth
- Overall execution time goes down

47



Pipeline hazards

- Structural hazards
 - Arises due to resource conflicts
 - Happens if the same resource is accessed in at least two stages of the pipe
- Control hazards
 - Problems with branches
 - A branch does not resolve immediately after it is fetched
 - What to fetch in the next cycle?
 - Defines an important parameter called *branch penalty*
- Data hazards
 - Dependent instructions may not execute back to back if dependence does not resolve in time
- Speedup of pipeline = $\text{pipeline depth} / (1 + \text{stall cycles per instruction})$

MAINAK CS422

49

Structural hazards

- Fewer resources than needed
 - Unpipelined functional units
 - Memory or register file ports
- Why fewer resources?
 - Reduction in complexity (and power consumption)
 - Make the common case fast: pipelined divider may only waste silicon estate

MAINAK CS422

50

Structural hazards

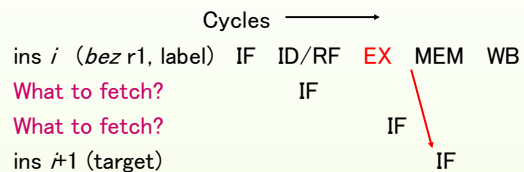
- All hazards introduce stalls or pipeline bubbles
 - Example: 40% data memory access; one memory port; adding second memory port slows down clock by k times; what is maximum k so that two-ported system sees some gain?
 - Solution: Whenever there is a data memory access, a stall cycle has to be introduced to handle shortage of memory ports in the single-ported memory design; so, 40% instructions would have a CPI of 2 and the remaining 60% would have CPI 1
 - Average CPI of single-ported design = $0.4 \cdot 2 + 0.6 \cdot 1 = 1.4$
 - Let the cycle time of single-ported design be t
 - Execution time of single-ported design = $1.4tN$ assuming N instructions
 - Execution time of dual-ported design = ktN
 - We want $ktN < 1.4tN$ i.e., $k < 1.4$

MAINAK CS422

51

Control hazard

- Branches pose a problem



- Two pipeline **bubbles**: increases average CPI
- Can we reduce it to one bubble?
- Define target and fall-through

MAINAK CS422

52

Control hazard

- Definition of fall-through and target of a branch instruction

```

beq $1, $2, label
ins
ins } FALL THROUGH
ins
label: TARGET
    
```

- Most often seen (but not always true):

```

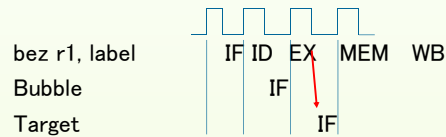
if (cond) {
    // FALL THROUGH
}
else {
    // TARGET
}
    
```

MAINAK CS422

53

Branch delay slot

- MIPS R3000 has one bubble
 - Called branch delay slot
 - Exploit clock cycle phases
 - On the positive half compute branch condition
 - On the negative half fetch the target



- The PC update hardware (selection between target and next PC) works on the lower edge

MAINAK CS422

54

Branch delay slot

- Can we utilize the branch delay slot?
 - The delay slot is always executed (irrespective of the fate of the branch)
 - Reason why branch target is $PC+4+(\text{sext}(\text{offset})\ll 2)$
 - Boost instructions common to fall through and target paths to the delay slot or from earlier than the branch

```

if (cond) {
    // fall through
}
else {
    // target
}
    
```

branch instruction

delay slot

55

Branch delay slot

- Can we utilize the branch delay slot?
 - Boost instructions common to fall through and target paths to the delay slot or from earlier than the branch
 - Not always possible to find by the compiler
 - Compiler will have to be careful also to not break anything
 - Must boost something that does not alter the outcome of fall-through or target basic blocks
 - If the BD slot is filled with useful instruction then we don't lose anything in CPI; otherwise we pay a **branch penalty** of one cycle

MAINAK CS422

56

Branch penalty

- To increase the clock frequency, designers may choose a deeper pipeline
 - Subdivide the longest stage further
- If the number of pipe stages increases between IF and EX, branch penalty also increases
 - Consider the following three example designs
 - Pipe1: IF ID/RF EX ... Br. penalty?
 - Pipe2: IF ID RF EX ... Br. penalty?
 - Pipe3: IF1 IF2 ID RF1 RF2 EX ... Br. penalty?
 - Assume no branch delay slot
 - Meaning that IF or EX cannot be accommodated in half cycle
 - Usually not possible for high-frequency designs

57

Branch penalty

- To increase the clock frequency, designers may choose a deeper pipeline
 - Subdivide the longest stage further
- If the number of pipe stages increases between IF and EX, branch penalty also increases
- Today all processors rely on branch predictors that observe the behavior of individual branches and learn to predict their future behavior
 - Makes it possible to infer the next instruction's PC even before the branch executes
 - Pipeline must be flushed on a wrong prediction

58

Branch prediction

- Branch prediction
 - We can put a branch target cache in the fetcher
 - Also called branch target buffer (BTB), the simplest predictor
 - Use the lower bits of the instruction PC to index the BTB
 - Use the remaining bits to match the tag
 - In case of a hit the BTB tells you the target of the branch when it executed last time
 - You can hope that this is correct and start fetching from that predicted target provided by the BTB
 - Later you get the real target, compare with the predicted target, and throw away the fetched instruction in case of misprediction; keep going if predicted correctly

MAINAK CS422

59

Branch target buffer

- BTB is looked up with the PC of every instruction in parallel with fetching the instruction
 - On a hit, it provides two pieces of information: this instruction is a control transfer instruction and the target of this control transfer instruction seen last time
 - This target will be used to fetch in the next cycle
 - On a miss, the fetcher has no option but to fetch from the fall through path (PC+4) in the next cycle
 - A control transfer instruction is inserted in the BTB after the EX stage when its target is known
 - A lookup at this point may hit in the BTB; if the branch is not taken, the BTB entry is invalidated; otherwise the entry is updated with the taken target
 - If a lookup at this point misses in the BTB, a new entry is allocated provided the branch is taken

Branch target buffer

- Assume for a program
 - 90% of all control transfer instructions hit in the BTB
 - 90% of outcomes provided by BTB hits are correct
 - 20% of control transfer instructions that miss in the BTB result in taken branches
 - Fraction of bubbles saved = BTB prediction accuracy = $0.9 \times 0.9 + 0.1 \times 0.8 = 0.89$
 - 11% branches suffer from mispredictions and will require some "recovery" mechanism for correct execution

MAINAK CS422

61

Branch prediction

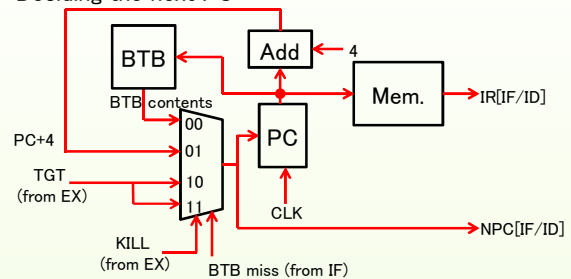
- BTB will work great for
 - Loop branches (how many misprediction?)
 - Subroutine calls
 - Unconditional branches
- What about *jalr*?
- Conditional branch prediction
 - Rather dynamic in nature
 - The last target is not very helpful in general (if-then-else)
 - Need a direction predictor (predicts taken or not taken)
 - Once that prediction is available we can compute the target
 - How does this co-exist with BTB? Target ALU in ID/RF?
- Return address stack (RAS): push/pop interface
 - To handle *jr* target prediction

Branch prediction

- Deciding the next PC
 - Observe: in 5-stage MIPS with half-phase IF, a conditional branch predictor is of no use (unless prediction and target can both be computed in IF); same is true about a RAS
 - Every cycle, the fetcher has to select from three options: PC+4 (from IF), BTB output (from IF), and actual target (bypassed from EX stage; this is available early for *jal*, *jalr*, and *j*)
 - Assume that the BTB lookup returns a tuple: (hit/miss, BTB contents); on a miss the second entry is arbitrary
 - If last to last instruction was a control flow instruction, compare BTB contents for that instruction with actual target; on a mismatch, select actual target and zero out ID/EX reg.; ID/EX inputs are ANDed with \sim KILL
 - Otherwise select BTB contents if BTB hit; else PC+4

Branch prediction

- Deciding the next PC



`beq $1, $2, label` IF ID/RF EX DMEM WB

- KILL = $\text{IsCntrIns[ID/EX]} \ \&\& \ (\text{NPC[ID/EX]} \neq \text{TGT})$
- NPC is carried till ID/EX register
- IsCntrIns is generated by decoder

Branch prediction

- Understanding the KILL logic
 - Assume that IF operates during the second half of a cycle
- | Cycle → | 0 | 1 | 2 | 3 |
|---------------------|----|-------|-------|-------|
| beq \$1, \$2, label | IF | ID/RF | EX | DMEM |
| InsX | | IF | ID/RF | EX |
| InsY | | | IF | ID/RF |
- We are currently in cycle 2 deciding the PC of InsY
 - The KILL logic may need to remove InsX from the pipeline if the next PC (NPC) generated in cycle 0 was wrong
 - Observe: NPC generated in cycle 0 has been carried forward with the beq instruction and is currently available in ID/EX pipeline register

Branch prediction

- Understanding the KILL logic
 - Assume that IF operates during the second half of a cycle
- | Cycle → | 0 | 1 | 2 | 3 |
|---------------------|----|-------|-------|-------|
| beq \$1, \$2, label | IF | ID/RF | EX | DMEM |
| InsX | | IF | ID/RF | EX |
| InsY | | | IF | ID/RF |
- We are currently in cycle 2 deciding the PC of InsY
 - The KILL logic may need to remove InsX from the pipeline if the next PC (NPC) generated in cycle 0 was wrong
 - So, we need to compare NPC[ID/EX] with the TGT generated in the current cycle in the EX stage
 - This comparison is relevant only if the instruction in EX stage is a control transfer instruction (IsCntrlIns[ID/EX])
 - IsCntrlIns is generated by decoder when decoding an instruction

Branch prediction

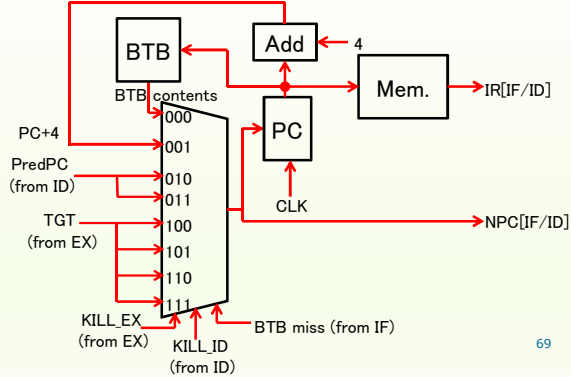
- Understanding the KILL logic
 - Assume that IF operates during the second half of a cycle
- | Cycle → | 0 | 1 | 2 | 3 |
|---------------------|----|-------|-------|-------|
| beq \$1, \$2, label | IF | ID/RF | EX | DMEM |
| InsX | | IF | ID/RF | EX |
| InsY | | | IF | ID/RF |
- We are currently in cycle 2 deciding the PC of InsY
 - The KILL logic may need to remove InsX from the pipeline if the next PC (NPC) generated in cycle 0 was wrong
 - InsX is currently in ID/RF stage and the easiest way to remove it from the pipeline is to zero out the input to the ID/EX pipeline register (that will change InsX to a NOP)
 - This is achieved by ANDing the input to ID/EX register with ~KILL (if KILL is 1, the input to ID/EX will be zero)

Branch prediction

- Deciding the next PC
 - Now assume that either the branch target cannot be made ready in half cycle or instruction fetch cannot be accommodated in half cycle
 - A direction predictor and a RAS is helpful provided instruction decoding followed by prediction can be completed in a cycle
 - We assume that the direction predictor (DP) or RAS can offer a better prediction than BTB and hence, will override the BTB prediction
- | Cycle → | 0 | 1 | 2 | 3 | 4 | 5 |
|---------------------|----|-------|-----|--------|-----|---|
| beq \$1, \$2, label | IF | ID/RF | EX | DMEM | WB | |
| | | | BTB | DP/RAS | TGT | |
- Now the fetcher has another option, namely, the predicted target from ID stage

Branch prediction

- Deciding the next PC



69

Branch prediction

- What is PredPC coming from the decode stage?
 - Assume the existence of an address adder in ID stage
 - We have four options
 - If instruction is conditional branch, look up direction predictor and compute PredPC based on that
 - If instruction is JR, pop RAS and use that as PredPC
 - If instruction is JAL or J, compute PredPC based on target field from IR
 - In all other cases, let PredPC be NPC[IF/ID]
 - After ID/RF stage, NPC is dropped and PredPC is carried forward along the pipeline

MAINAK CS422

70

Branch prediction

- Understanding the KILL_ID and KILL_EX logic

Cycles →	0	1	2	3
beq \$1, \$2, label	IF	ID/RF	EX	DMEM
		BTB	DP/RAS	TGT
InsX		IF	ID/RF	EX
InsY			IF	ID/RF
InsZ				IF

- Assume that we are in cycle 2 and trying to decide the PC of InsZ
- If InsX was mispredicted (i.e., NPC generated in cycle 1 was wrong) or beq was mispredicted, InsY will be removed from the pipeline
- If beq was mispredicted, both InsX and InsY will be removed from the pipeline

71

Branch prediction

- Understanding the KILL_ID and KILL_EX logic

Cycles →	0	1	2	3
beq \$1, \$2, label	IF	ID/RF	EX	DMEM
		BTB	DP/RAS	TGT
InsX		IF	ID/RF	EX
InsY			IF	ID/RF
InsZ				IF

- Assume that we are in cycle 2 and trying to decide the PC of InsZ
- The KILL_ID signal available in cycle 2 is generated based on the correctness of NPC generated in cycle 1 (i.e., when InsX was fetched); this NPC is available in IF/ID pipeline register in cycle 2
- So, NPC[IF/ID] should be compared with PredPC for InsX

72

Branch prediction

- Understanding the KILL_ID and KILL_EX logic

Cycles →	0	1	2	3	
beq \$1, \$2, label	IF	ID/RF	EX	DMEM	WB
		BTB	DP/RAS	TGT	
InsX		IF	ID/RF	EX	DMEM
InsY			IF	ID/RF	EX
InsZ				IF	ID/RF

- Assume that we are in cycle 2 and trying to decide the PC of InsZ
- NPC[IF/ID] should be compared with PredPC for InsX provided InsX is a control transfer instruction
- PredPC and IsCntrlIns are being generated in cycle 2 in ID/RF stage where InsX is located right now
- Therefore, $KILL_ID = IsCntrlIns \ \& \ (PredPC \neq NPC[IF/ID])$

Branch prediction

- Understanding the KILL_ID and KILL_EX logic

Cycles →	0	1	2	3	
beq \$1, \$2, label	IF	ID/RF	EX	DMEM	WB
		BTB	DP/RAS	TGT	
InsX		IF	ID/RF	EX	DMEM
InsY			IF	ID/RF	EX
InsZ				IF	ID/RF

- Assume that we are in cycle 2 and trying to decide the PC of InsZ
- If KILL_ID is 1, InsY has been wrongly fetched and should be removed from the pipeline
- This is achieved by ANDing the input to IF/ID pipeline register with $\sim KILL_ID$
 - This will turn InsY into a NOP from ID/RF stage onward

Branch prediction

- Understanding the KILL_ID and KILL_EX logic

Cycles →	0	1	2	3	
beq \$1, \$2, label	IF	ID/RF	EX	DMEM	WB
		BTB	DP/RAS	TGT	
InsX		IF	ID/RF	EX	DMEM
InsY			IF	ID/RF	EX
InsZ				IF	ID/RF

- Assume that we are in cycle 2 and trying to decide the PC of InsZ
- The KILL_EX signal is decided based on the comparison between PredPC generated when the beq instruction passed the ID/RF stage and the beq TGT generated now in EX stage
- Observe: PredPC of beq is currently in ID/EX pipeline reg.

Branch prediction

- Understanding the KILL_ID and KILL_EX logic

Cycles →	0	1	2	3	
beq \$1, \$2, label	IF	ID/RF	EX	DMEM	WB
		BTB	DP/RAS	TGT	
InsX		IF	ID/RF	EX	DMEM
InsY			IF	ID/RF	EX
InsZ				IF	ID/RF

- Assume that we are in cycle 2 and trying to decide the PC of InsZ
- The KILL_EX signal is decided based on the comparison between PredPC generated when the beq instruction passed the ID/RF stage and the beq TGT generated in EX stage
- Hence, $KILL_EX = IsCntrlIns[ID/EX] \ \&\& \ (PredPC[ID/EX] \neq TGT)$

Branch prediction

- Understanding the KILL_ID and KILL_EX logic

Cycles →	0	1	2	3	
beq \$1, \$2, label	IF	ID/RF	EX	DMEM	WB
		BTB	DP/RAS	TGT	
InsX		IF	ID/RF	EX	DMEM
InsY			IF	ID/RF	EX
InsZ				IF	ID/RF

- Assume that we are in cycle 2 and trying to decide the PC of InsZ
- If KILL_EX is 1, both InsX and InsY have been wrongly fetched and should be removed from the pipeline
- This is achieved by ANDing the input to IF/ID as well as ID/EX pipeline registers with \sim KILL_EX
 - Note: input to IF/ID pipeline register is ANDed with \sim (KILL_EX & \sim KILL_ID)

MAINAK CS422

79

Branch prediction

- Summary of KILL_ID and KILL_EX logic

Cycles →	0	1	2	3	
beq \$1, \$2, label	IF	ID/RF	EX	DMEM	WB
		BTB	DP/RAS	TGT	

- Assume that we are in cycle 2 and trying to decide the PC of cycle 3; beq is fetched in cycle 0
- KILL_ID = $\text{IsCntrIns} \ \&\& \ (\text{PredPC} \neq \text{NPC}[\text{IF/ID}])$; kills the instruction being fetched in the current cycle (i.e., cycle 2)
 - Because BTB offered a wrong prediction in cycle 1
 - IF/ID register inputs are ANDed with \sim KILL_ID
- KILL_EX = $\text{IsCntrIns}[\text{ID/EX}] \ \&\& \ (\text{PredPC}[\text{ID/EX}] \neq \text{TGT})$; kills the instructions fetched in the last two cycles
 - Because DP offered a wrong prediction for beq
 - ID/EX and IF/ID inputs are ANDed with \sim KILL_EX
- PredPC gets fed into the ID/EX register in place of NPC

78

Branch prediction

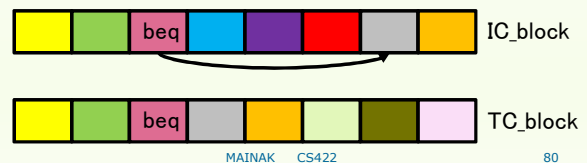
- When to insert an instruction into the BTB?
 - Put taken control instructions after the real target is known (always known after EX stage)
 - Includes unconditional jumps (J), procedure calls (JAL, JALR), and taken conditional branches
 - On a miss, replace an entry (LRU?)
 - On a hit, update the entry
- When to push on RAS?
 - JAL, JALR in ID stage after instruction is decoded
 - May have to repair RAS if KILL_EX is enabled

MAINAK CS422

79

Possible optimizations

- BTB optimizations
 - Store one or more target instructions instead of PC
 - Branch folding for unconditional branches
- Fetcher optimizations
 - Explore both branch paths simultaneously for conditional branches (any problem?)
 - Pre-decoded instructions for predictor access in fetcher
 - Trace cache (more later)



MAINAK CS422

80

Direction predictors

- How about static prediction?
 - Always not-taken (NT) or always taken (T): penalty?
 - Forward not-taken and backward taken (rationale?)
- Deeper pipelines
 - **Big problem: deeper pipelines increase branch penalty**
 - Must have better branch predictors for deeper pipeline

MAINAK CS422

81

Direction predictors

- Deeper pipelines
 - Example: After instruction fetch, MIPS R4000 takes 2 pipe stages to compute the target and one more to evaluate the condition; assume 4% unconditional jump, 6% NT conditional branch, 10% T conditional branch; evaluate CPI increase for three schemes: unconditional stall, predicted always T, predicted always NT
 - Let us calculate the branch penalty (CPI delta) for each case

	j	NT Cond.	T Cond.	Total
– Stall	2 (+0.08)	3 (+0.18)	3 (+0.3)	+0.56
– Predicted T	2 (+0.08)	3 (+0.18)	2 (+0.2)	+0.46
– Predicted NT	2 (+0.08)	0 (0)	3 (+0.3)	+0.38

MAINAK CS422

82

Branch likely

- Filling the delay slot may be problematic
 - Compiler must predict (statically) where the branch is going
 - If cannot prove correctness of prediction, it has to be conservative
 - Some ISAs provide a nullifying branch or branch likely instruction
 - Compiler encodes the predicted direction in the instruction and fills the delay slot accordingly
 - If at run-time the branch turns out to behave otherwise, the delay slot is flushed
 - Example: MIPS offers cancel-if-not-taken branch instruction so that if compiler thinks the branch will be taken it can fill the delay slot from the target; is a cancel-if-taken instruction that useful? (decision favoring loops)

MAINAK CS422

83

Control dependence

- Roughly every fifth instruction is a branch
 - Need to be on the right *control flow path*
 - This is the source of input to the pipeline
 - Static techniques are not enough
 - Need highly accurate dynamic predictors
 - Speculate past branches: Alpha 21264 allows 20 outstanding branches, MIPS R10000 allows only 4
 - Need to speculate past predicted branches in deeper pipelines (not a big issue in five-stage pipe)
 - Prediction accuracy?
 - Probability of a correct prediction is p
 - Probability of staying on correct path after n predictions p^n
 - What is minimum p if n is 4 (MIPS R10k)? If n is 20?

Direction predictors

- Let us encode each taken branch as 1 and not taken branch as 0
 - The behavior of a conditional branch can be represented as a binary string
 - Loop branch: 11111...110
 - Alternating if-else: 1010101010 or 0101010101
 - If-else branches can exhibit a wide variety of patterns

Direction predictors

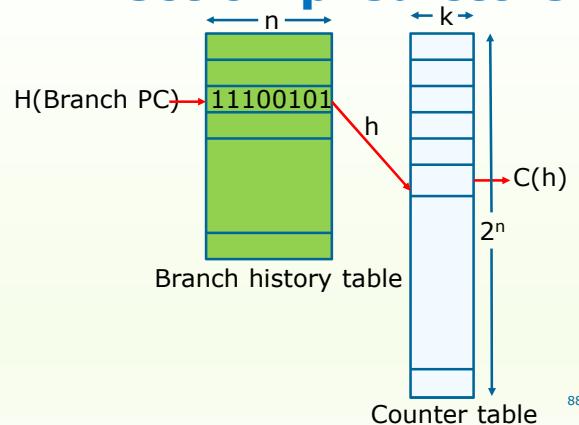
- The problem of direction prediction is essentially design of an estimator that, given an n -bit history, tells us the next most likely outcome for a particular static branch instruction
 - All branch predictors compute the probability of seeing a zero or one, given the recent pattern history h of some limited length n
 - Suppose the number of time 0 appears after h is $C0$ and similarly define $C1$; the prediction is 1 if $C1 \geq C0$ and 0 otherwise
 - Instead of having two counters, $C1 - C0$ is maintained

Direction predictors

- Let the difference counter be $C(h)$ for a certain history h
 - Let $C(h)$ be of k bits length (this is independent of the history length)
 - Can count from 0 to $2^k - 1$
 - On seeing 0 after h , decrement $C(h)$; on seeing 1 after h , increment $C(h)$
 - Saturates at boundaries (a saturating counter)
 - Does not decrement below 0 or increment above $2^k - 1$
 - By examining $C(h)$ at any point in time, we can say which outcome had higher likelihood in the last 2^k occurrences of history h
 - Need to shift the origin to mid-point 2^{k-1}
 - If $C(h)$ is below mid-point, the prediction is zero; otherwise 1

87

Direction predictors

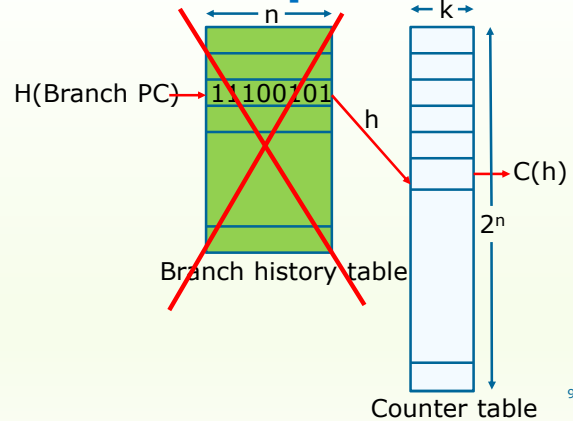


Direction predictors

- How many different histories?
 - No history? [No BHT] Two possible designs:
 - Just a global counter that counts occurrences of 0's and 1's
 - Not very useful
 - One counter per branch (known as bimodal predictor)
 - Somewhat useful: helps identify largely bimodal distributions
 - One global history? [Size of BHT = 1 entry]
 - Captures cross-correlation between branches
 - Since history is a sliding pattern, h will keep on changing
 - One counter for each history pattern? Use a hashmap
 - One local history per branch? Size of BHT?
 - Hash history patterns to counters
 - Loses global correlation

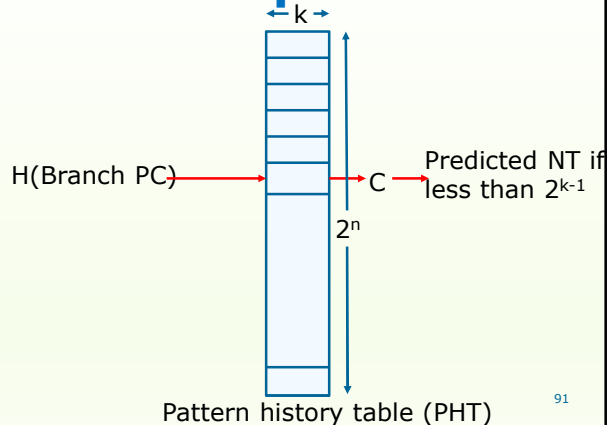
89

Bimodal predictors



90

Bimodal predictors



91

Bimodal predictors

- Simplest of the lot (used in MIPS R10000)
 - Maintains a pattern history table (PHT)
 - MIPS called it a branch history table (BHT)
 - Each entry is a saturating counter
 - Basic idea is to go with the most frequent pattern seen in the past
 - How do you index it?
 - Aliasing?
 - How wide is each counter? Do you really implement a counter?
 - Performance of loops (simplest kind of control flow)
 - Alternating branches?
 - Correlating branches?

MAINAK CS422

92

Correlating predictors

- Need some global view
 - if $(f == x) \{ f = y; \}$ if $(g == x) \{ g = y; \}$ if $(f != g) \{ \dots \}$
 - Global history? [One BHT entry for all branches]
 - Two levels of tables [BHT and PHT, already introduced]
- Taxonomy of branch predictors
 - First level table: global (G), per set (S), per branch (P)
 - Second level table: global (g), per set (s), per branch (p)
 - Update method: static (S), adaptive (A)
 - Let's look at PAp, SAg, GAg (what does each one buy?)
 - The BHT in GAg is referred to as the global history register
 - One special case GAg: gshare (index PHT with PC xor GHR)
 - PA* and SA* are often referred to as local predictors and GAg is referred to as global predictors
 - Hybrids: combine multiple of these (why?)

MAINAK

CS422

93

Direction predictors

- Direction predictor DP: $\{\text{BranchPC}, \text{GHR}\} \rightarrow \{0, 1\}$
 - $\text{DP} \equiv \text{SEL}(\text{DP}_0, \text{DP}_1, \dots, \text{DP}_{k-1}, \text{BranchPC}, \text{GHR})$
 - $\text{DP}_n : \{\text{BranchPC}, \text{GHR}\} \rightarrow \{0, 1\}$
 - $\text{SEL} : \{0, 1\}^k, \text{BranchPC}, \text{GHR} \rightarrow \{0, 1\}$
 - $\text{DP}_n = \text{PHT}_n(\text{BHT}_n)$
 - $\text{BHT} : \{\text{BranchPC}, \text{GHR}\} \rightarrow \{\text{BranchPC}, \text{History (h)}\}$
 - $\text{PHT} : \{\text{BranchPC}, \text{History (h)}\} \rightarrow \{0, 1\}$
 - $\text{BHT} = \{\text{History (\#m)}\}^p$
 - $\text{PHT} = \{\text{Saturating counters (\#q)}\}^{2^m}$
 - Bimodal predictor (BIM) has a null BHT and a PHT that takes branch PC as input
 - PAp, SAp, GAp have p different PHTs, each of size shown above

MAINAK

CS422

94

Direction predictors

- PAp
 - p = no. of static branch instructions N, p different PHTs, each indexed with the respective BHT entry contents
 - Total size = $mp + pq2^m \text{ bits} = mN + Nq2^m \text{ bits}$
- PA_s
 - p = no. of static branch instructions N, one PHT for a set of branches; p/s number of PHTs
 - Total size = $mp + (p/s)q2^m \text{ bits} = mN + (N/s)q2^m \text{ bits}$
- PA_g
 - p = no. of static branch instructions N, one global PHT indexed by BHT entry contents
 - Total size = $mp + q2^m \text{ bits} = mN + q2^m \text{ bits}$

MAINAK

CS422

95

Direction predictors

- SAp
 - No. of PHTs = no. of static branch instructions N, each indexed with the respective BHT entry contents
 - No. of BHT entries = K
 - Total size = $mK + Nq2^m \text{ bits}$
- SA_s
 - One PHT for a set of branches; N/s number of PHTs where N is the number of static branch instructions
 - No. of BHT entries = K
 - Total size = $mK + (N/s)q2^m \text{ bits}$
- SA_g
 - One global PHT indexed by BHT entry contents
 - Total size = $mK + q2^m \text{ bits}$

MAINAK

CS422

96

Direction predictors

- GAp
 - BHT has just one entry, the global history register
 - No. of PHTs = no. of static branch instructions N, each indexed with the BHT entry contents
 - Total size = $m + Nq2^m$ bits = $m + Nq2^m$ bits
- GAs
 - BHT has just one entry, the global history register
 - One PHT for a set of branches; N/s number of PHTs where N is the number of static branch instructions
 - Total size = $m + (N/s)q2^m$ bits
- GAg
 - BHT has just one entry, the global history register
 - One global PHT indexed by BHT entry contents
 - Total size = $m + q2^m$ bits

97

Direction predictors

- Gshare
 - Similar to GAg
 - PHT is indexed with BranchPC XOR GHR
- Updating the direction predictor
 - This is done after EX stage when the branch outcome is known
 - First PHT is updated by indexing it with the old BHT entry
 - Need to carry the BHT entry along the pipeline
 - Next BHT is accessed and the BHT entry is updated by shifting in the outcome
 - Should the predictor be updated speculatively with the predicted outcome?
 - Pros and cons

MAINAK CS422

98

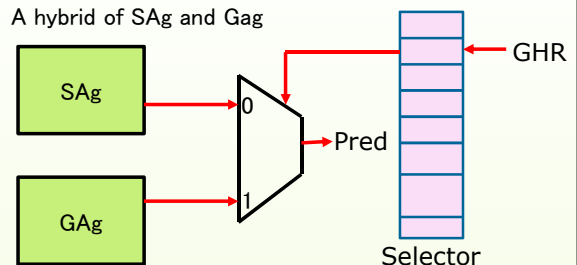
21264 Tournament

- A hybrid of SAg and GAg
 - Combines a local and a global predictor
 - SAg has 1024 entries (p) with 10 bits each (m) in BHT and the second level has 1024-entry PHT each being 3 bits (q)
 - GAg has 12-bit GHR (m) and 4096-entry PHT each being 2 bits (q)
 - Meta predictor or selector is BIM with 4096 entries (indexed by GHR)
- Need to carry two BHT entries along the pipeline
 - One from SAg and one from GAg
- Update protocol for the components?

MAINAK CS422

99

21264 Tournament

- A hybrid of SAg and GAg
 
- Need to carry two BHT entries along the pipeline
 - One from SAg and one from GAg
- Update protocol for the components?

MAINAK CS422

100

Handling indirect calls

- BTB indexing with hash of path history
 - Efficient handling of indirect calls (already discussed)
 - Two possibilities: index with $h(PC, GHR)$ or index with a running hash of all branch PCs within a sliding window
 - Former is easier to implement as GHR is already maintained; for latter, a sliding window of branch PCs has to be maintained

MAINAK CS422

101

Fused BTB and DP

- Timeline for BTB and direction predictor lookup
 - BTB in IF and DP after instruction is decoded
 - May be low-performance if BTB has low accuracy e.g., for alternating branches BTB is always wrong
- Extend each BTB entry with a single bit to specify if the entry corresponds to a conditional branch
 - Look up BTB and direction predictor in parallel with instruction fetch in IF stage
 - If BTB lookup hits and indicates a conditional branch, use direction predictor's outcome to direct fetch in the next cycle; use target provided by BTB entry
 - A BTB entry always holds the target for a conditional branch and is never invalidated (unless replaced)
 - On a BTB miss, the DP outcome is carried forward and used or discarded after the instruction is decoded

Fused BTB and DP

- Deciding the next PC
 - Now there are four signals: KILL_EX from EX, KILL_ID from ID, BTBmiss from IF, isCondBr from IF (only if BTB hit)
 - There are five selections: TGT from EX, PredPC from ID, BTB outcome for non-conditional branches, DP outcome for conditional branches that hit in BTB, and PC+4
 - If KILL_EX is 1, select TGT from EX
 - If KILL_EX is 0 and KILL_ID is 1, select PredPC from ID
 - If KILL_EX is 0, KILL_ID is 0, BTBmiss is 1, select PC+4
 - If KILL_EX is 0, KILL_ID is 0, BTBmiss is 0, isCondBr is 1, select DP outcome
 - If KILL_EX is 0, KILL_ID is 0, BTBmiss is 0, isCondBr is 0, select BTB outcome

MAINAK CS422

103

Criticality of branches

- Not all branches are equally important
 - Predicting some correctly is critical to performance, while others have very little impact on performance
- Branch criticality factors
 - Misprediction penalty for last DP
 - Minimum is number of pipe stages between last DP and branch execution
 - Certain branches may get delayed due to data dependence
 - Predicting these correctly is important for performance
 - Cache pollution due to wrong path execution
 - Conflicting instruction and data working sets along the two branch paths
 - Criticality of the correct path e.g., if always starts with an instruction or data cache miss

Criticality of branches

- Critical branches need high prediction accuracy
 - Identifying branches that mispredict frequently is easy
 - Have a small cache of recently mispredicted branch PCs
 - All of these may not be critical
 - Discovering good features that correlate well with the behavior of the critical branches is difficult
 - Most of these branches are data-dependent
 - Prediction accuracy depends on the entropy of the data they depend on
 - Today's best DP has very high prediction accuracy
 - Small number of branches cause most of the mispredictions and these are highly critical branches
 - The performance gap between such a predictor and an oracle is large

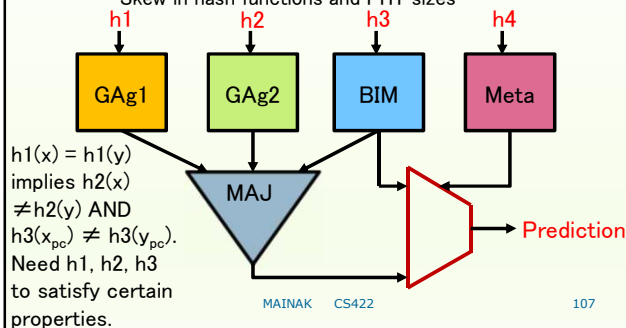
Advanced predictors

- Active area of research
 - Deeper pipelines need much better predictors
 - MIPS R10000 has a 5-stage integer pipe, Alpha 21264 has a 7-stage integer pipeline, Intel Pentium 4 EE has a 31+ – stage integer pipeline
 - Branch misprediction penalty: number of cycles between prediction and verification
 - Deeper the pipeline is, more work is lost due to misprediction
 - Challenges
 - Destructive aliasing in PHT
 - Need for large history (exponential storage overhead)
 - Better prediction for data-dependent branches
 - Efficient handling of indirect calls

106

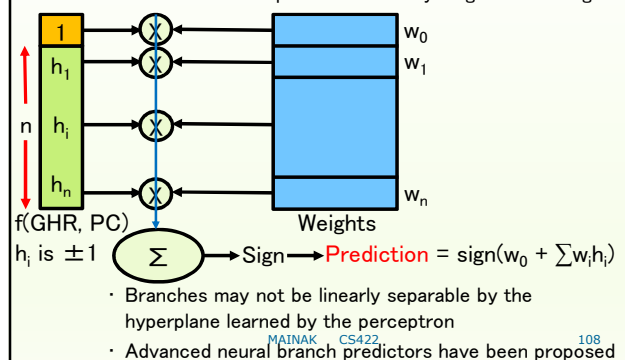
Advanced predictors

- Gskew family of predictors
 - Aims at reducing PHT aliasing by smart hashing
 - Skew in hash functions and PHT sizes



Advanced predictors

- Perceptron predictors
 - Linear relationship between history length and storage

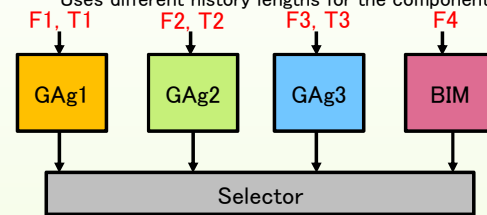


Advanced predictors

- Prediction by partial match (PPM)
 - Another way to have long history is by preparing a hash of the history and using the hash to index into PHT
 - One popular hash is folded XOR e.g., could hash a 40-bit history to a ten-bit index by computing $h[39:30] \text{ XOR } h[29:20] \text{ XOR } h[19:10] \text{ XOR } h[9:0]$
 - Bits of PC can further be combined with the folded XOR to have an overall hash function $F(PC, h)$
 - Such a hash, though allows use of long history, may suffer from aliasing
 - PPM disambiguates the aliases by maintaining a tag with each PHT entry; the tag is also a hash of PC and h, say, $T(PC, h)$
 - There is now a possibility of PHT hit/miss

Advanced predictors

- Global PPM (GPPM) predictors
 - Uses tag with each PHT entry to disambiguate aliases
 - Uses different history lengths for the components



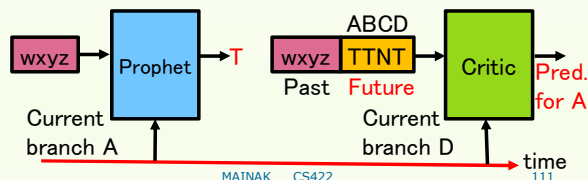
TAGE (Tagged Geometric length) predictor uses history lengths in GP

MAINAK CS422

110

Advanced predictors

- Prophet-critic predictor
 - Override by taking into account "future"
 - A two-component predictor; prophet's prediction is overridden by critic's prediction and critic's prediction is the final prediction
 - An example of a specific kind of overriding branch predictors



MAINAK CS422

111

Advanced predictors

- This is only a tiny sampling of relatively recent ideas in the domain of advanced branch predictors
- A very vast body of literature exists
 - Still an important area of research

MAINAK CS422

112

Summary

- Redirect fetch from various stages of the pipeline with increasingly better prediction
 - Branch target buffer, return address stack, direction predictors
- The fetcher selects the most appropriate next PC every cycle from among different indications coming from different stages
- Research problem
 - Focus effort on critical data-dependent branches
 - What features correlate well with the behavior of these branches?
 - Can compiler offer help?
 - Helper threads to pre-execute branches?

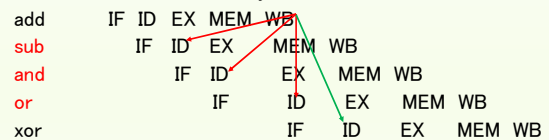
MAINAK

CS422

113

Data hazards

- Pipelining disturbs the sequential thought-process
 - Data dependencies among instructions start to show up
- ```
add r1, r2, r3
sub r4, r1, r5
and r6, r1, r7
or r8, r1, r9
xor r10, r1, r11
```
- Result of add is needed by all instructions (RAW hazard)



MAINAK

CS422

114

## Data hazards

- How to avoid increasing CPI?
    - Stalling is clearly not acceptable
    - Phased register file solves three-cycle apart RAW
    - Can we forward the correct value just in time?
- ```
add    IF ID EX MEM WB
sub     IF ID EX MEM WB
and     IF ID EX MEM WB
```
- Read wrong value in ID/RF, but bypassed value overrides it (how to implement it?)
 - Always feed bypassed value to the ALU input
 - How many sources in bypass network?
 - Do we need to bypass to MEM stage also?

MAINAK

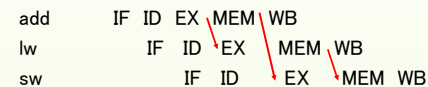
CS422

115

Data hazards

- MEM/WB to MEM bypass

```
add r1, r2, r3
lw  r4, 0(r1)
sw  r4, 20(r1)
```



- How many destinations in bypass network?
- Overall complexity (i.e. number of MUXes and wires)?
- Yet another drawback of deep pipelining!

MAINAK

CS422

116

Data hazards

- Another MEM/WB to MEM bypass example

```
add r1, r2, r3  IF ID EX MEM WB
sw   r1, 0(r2)      IF ID EX MEM WB
```

- Another MEM/WB to EX bypass path

- Need for stores

```
add r1, r2, r3  IF ID EX MEM WB
sub r4, r1, r1  IF ID EX MEM WB
sw   r1, 0(r2)      IF ID EX MEM WB
```

- Why not have a WB to MEM bypass path instead?

- Always take value from the “most recent” bypass path

```
add r1, r2, r3  IF ID EX MEM WB
sub r1, r1, r5  IF ID EX MEM WB
and r1, r1, r7  IF ID EX MEM WB
```

117

Data hazards

- Can we always avoid stalling?

```
lw   r1, 0(r2)
sub  r4, r1, r5
and  r6, r1, r7
or   r8, r1, r9
```

```
lw   IF ID EX MEM WB
sub  IF ID EX MEM WB
and  IF ID EX MEM WB
or   IF ID EX MEM WB
```

- Need some time travel (backwards)! Not yet feasible!!
- Hardware *pipeline interlock* to stall the *sub* by a cycle
- Early generations of MIPS (Microprocessor without Interlocked Pipeline Stages) had the compiler to fill the *load delay slot* with something independent or a NOP

118

Bypass and stall logic

- What does the bypass logic look like?
 - How many forwarding paths?
 - How large are the MUXes?
- What about load interlocks?
 - Detecting possible hazards early simplifies things
 - Fixed positions of rs, rt, rd are important for RF access and hazard detection
 - In MIPS all interlocks can be implemented in ID/RF
 - Need to control IF and EX → *send NOP*
 - MIPS R3000 does not have any hardware interlock: compiler fills the load delay slot *and branch delay slot*

do not fetch anything

MAINAK CS422

119

Multi-cycle EX stage

- Why do we need multi-cycle EX stage?
 - Primarily to support floating-point operations: these are much complex to be finished in a cycle
 - Also, multiple functional units may be needed to avoid structural hazards
 - Assume four functional units: integer ALU, fp and integer multiplier, fp adder/subtractor, fp and integer divider
 - Latency of an instruction is defined by the number of cycles needed to produce the result from the time it issues (textbook takes a slightly different view)
 - Assume integer ALU instructions have latency of 1 cycle, loads have latency of 2 cycles (why?), fp add: 4 cycles, fp and integer multiply: 7 cycles, fp and integer divide: 25 cycles

MAINAK CS422

120

MEM → assume all cache hit

Multi-cycle EX stage

- Repeat interval of an instruction
 - Number of cycles between two instructions in the same category that can execute without a structural hazard
 - Depends on how the functional units are pipelined
 - Assume that all units other than the divider are pipelined
 - Division has a repeat interval of 25 cycles while other instructions can issue back-to-back (repeat interval 1 cycle)
 - What does the pipeline look like?
 - More pipeline registers
 - Any other complications?

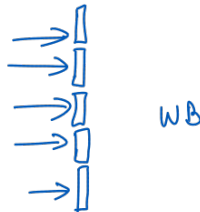
MAINAK CS422

121

Multi-cycle EX stage

- Integer ALU pipeline
IF ID EX WB
- Load/Store pipeline
IF ID EX MEM WB
- FP add pipeline
IF ID A1 A2 A3 A4 WB
- FP and integer multiply pipeline
IF ID M1 M2 M3 M4 M5 M6 M7 WB
- FP and integer divide pipeline
IF ID D1-D25 WB
- An instruction after getting decoded is sent along one of these pipelines depending on the opcode
 - All pipelines share IF, ID, and WB; share EX stage?

122

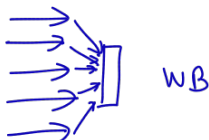


Multi-cycle EX stage

- Design options for WB stage
 - Have multiple pipeline registers in front it, one for each different pipeline (five registers)
 - Need a priority encoder to pick up the contents of the correct register
 - Can pick multiple registers if there are multiple register write ports
 - Different types of data hazards may arise (more discussion coming soon)
 - Have one pipeline register in front it and have a register write port scheduler in the ID stage to make sure that in a cycle two pipes do not write to this register
 - More discussion coming soon

MAINAK CS422

what if two types of write go for WB in same stage



New data hazards

- Structural hazards
 - Divider: stall instruction issue in ID/RF
 - Any suggestion for CPI improvement? (other than pipelined divider)
 - Floating-point register write ports

mult.d	IF	ID	M1	M2	M3	M4	M5	M6	M7	WB
add.d			IF	ID	A1	A2	A3	A4		WB
load.d					IF	ID	EX	MEM		WB
 - More write ports or hardware interlock?
 - Interlock options: detect in ID (shift register write port scheduler), detect just before WB (stall which instruction?)
 - Shift register's bit n is 1 if and only if some instruction will write to register file n cycles later [e.g., 0010001000]
 - Shift to right by one bit position every cycle

MAINAK CS422

124

read from LS side

$$\begin{array}{r} 1111011010 \\ 10000000 \\ \hline 1111011 \end{array}$$

New data hazards

Structural hazards

- Floating-point register write ports

mult.d, ..., ..., add.d, ..., load.d

mult.d IF ID M1 M2 M3 M4 M5 M6 M7 WB

add.d IF ID A1 A2 A3 A4 WB

load.d IF ID EX MEM WB

- Algorithm to maintain shift register port scheduler

- Let X be the shift register initialized to 0

- Do the following every cycle

- $X \leftarrow X \gg 1$

- Suppose the instruction currently in ID stage writes to register file n cycles later; prepare mask $\leftarrow 1000 \dots 0$ where there are n-1 zeros e.g., for the mult.d instruction mask $\leftarrow 10000000$

- while $((X \& \text{mask}) \neq 0)$ mask $\leftarrow \text{mask} \ll 1$ i.e., keep delaying register write of this instruction; the number of iterations is the number of interlock cycles required to avoid write port hazard

$X \leftarrow X | \text{mask}$

MAINAK

CS422

125

New data hazards

- More stalls due to RAW data hazard

- Even bypassing cannot nullify all stalls

load.d \$f4, 0(\$2)

mult.d \$f0, \$f7, \$f6

add.d \$f2, \$f0, \$f4

store.d \$f2, 0(\$2)

load.d IF ID EX MEM WB

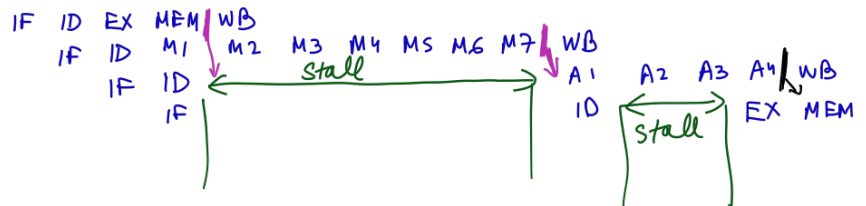
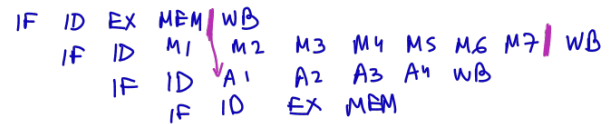
mult.d IF ID M1 M2 M3 M4 M5 M6 M7 WB

add.d IF ID A1 A2 A3 A4 WB

store.d IF ID EX MEM

MAINAK CS422

126



WAW hazards

- Write-after-write

add.d \$f2, \$f4, \$f6 IF ID A1 A2 A3 A4 WB

load.d \$f2, 0(\$2) IF ID EX MEM WB

- Is this realistic? Can WAW hazard ever happen if the compiler is sane?

bnez \$1, label

div.d \$f0, \$f2, \$f4 IF ID D1 D2 D3 D4 D5 D6

...

...

label: load.d \$f0, 20(\$4) IF ID EX MEM WB

- Handling WAW: delay issue of the latter instruction or prevent the earlier one from writing (can do it in ID/RF?)

- Which one is better performance-wise? *2nd one*

- What is the hardware?

- One way: Need one shift register for each fp register

- Another possibility: Maintain a bank of comparators

MAINAK

CS422

127

Hazard detection

- Need to look for integer and fp hazards

- Integer and floating-point instructions use separate register files

- But floating-point load/store uses integer registers as base: there could be a hazard between integer and floating-point instructions

- Also there are move instructions (mtc1 and mfc1) that move to/from floating-point register file to integer register file

- So in these last two cases we need to detect hazards between integer and floating-point instructions

- Otherwise hazards can happen between integer instructions only or floating-point instructions only: simplification made possible due to separate register files

- Any problems of having separate files? Why not unified?

MAINAK CS422

128

I D X MEX WB
 I D EX MEM WB
 I D X ME

$2N^2 \rightarrow N$ func units, each's output to 2 sources of all N func units ($N \times 2N$)
 $2NS \Rightarrow S$ pipeline regs, each to 2 srcs of all N func units ($S \times 2N$)

Hazard detection

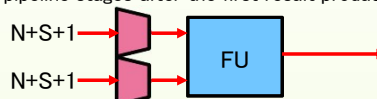
- Better to club structural, RAW, WAW hazard detection in ID/RF stage
 - For our example pipeline, structural hazard involves availability of divider and availability of register write port
 - For RAW detection, need to compare sources of current instruction with destinations of all outstanding instructions e.g. all fp adds issued during the last three cycles, all fp multiplications issued during the last six cycles, any division issued during the last 24 cycles, the load issued in the last cycle, etc. (load delay slot solves the last one)
 - For WAW detection, need to compare destination of current instruction with destination of all outstanding instructions

MAINAK CS422

129

New bypass control

- More wiring (more sources and destinations) $N=5$
 - $2 \times N^2 + 2 \times N \times S$ wires $S=1$
 - N = number of functional units (two inputs), S = number of pipeline stages after the first result producing stage



- This is an overestimate: why?
- For MIPS
 - Inter-file move instructions (mtc1 and mfc1) execute on adder/subtractor
 - Integer multiply/divide produces results in Hi/Lo
 - Implications on bypass network?

MAINAK CS422

130

int ALU

addu \$10, \$11, \$12

RF/EX - value it read in ID/RF (val in \$11)
 M7/WB - if \$11 ← mult
 D1-D25/WB - if \$11 ← div
 A4/WB - if prev instr is mfc1 then \$11 ← \$f1
 MEM/WB - output of cache (if prev instr is load to \$11)
 WB - prev-prev instr wrote to \$11
 EX/WB - prev instr same instr

e.g. M7 S1 S2 WB $S=3$
 → Ex/WB bypass in FP adder
 mtc1 \$f1, \$f1 \$f1 ← \$f1 addu \$f1, \$f1, \$f0
 mfc1 \$f1, \$f1 \$f1 ← \$f1 addu \$f1, \$f0, \$f1
 → ALU/WB bypass in INT ALU
 add.s \$f1, \$f2, \$f3
 mfc1 \$f1, \$f1

New bypass control

- Wider MUXes
 - How many inputs? $N+S+1$ in worst case
 - What about WAR hazard?
 - Write after read
 - Ins1 reads from register \$X
 - A later instruction Ins2 writes to \$X
 - Ins2 completes before Ins1
 - Is there a problem?
- Ins1 IF ID ...
 Ins2 IF ID ...
- Is RAR a hazard?
 - Read after read

MAINAK CS422

131

Exceptions

- Synonymous to interrupts or faults
 - Raised by I/O device request, system calls, integer arithmetic overflow, floating-point arithmetic anomaly, page faults, misaligned memory access, memory protection violation, decoding illegal opcode, etc.
 - Usual model is to transfer control to some kernel handler
 - The kernel handler decodes the situation and takes appropriate action
- Types of exceptions
 - Synchronous vs. asynchronous: asynchronous easy to handle
 - User requested vs. coerced or hardware
 - User maskable vs. user non-maskable
 - Within vs. between instructions: latter is easy
 - Resuming vs. terminating

underflow
overflow

FP mult - needs int ALU for mtc instr dependence
 FD add - also needs int ALU bypass

mtc1 \$f1, \$f1
 add.s \$f2, \$f3, \$f1

Data bypass - also needs all

MAINAK CS422

Data Cache Bypass:

the bypass gives the stored value to the D cache. Not the address

sw (\$2, 0(\$10))

swc1 (\$f2, 0(f10))

bypass from multiplier

33

Synchronous - come from within an instr, **async** - come from external events (key punch)
 user req - user prog has raised this excepⁿ **hardware** - hardware raised
 user maskable - masked by user prog, can be ignored
 non maskable - related to health of machine (user cannot ask to ignore)
 within - raised by some operation of an instr (overflow)
 between instr - external signal (easy, can be deferred)
 terminating - FP arithmetic anomaly (div by 0)
 resuming - page fault

Exceptions

- I/O device request
 - Async, coerced, non-maskable, between, resuming
- System call
 - Sync, user requested, non-maskable, between, resuming
- Trap (breakpoint, etc.)
 - Sync, user requested, maskable, between, resuming
- Arithmetic overflow/underflow, math exceptions
 - Sync, coerced, maskable, within, resuming/terminating
- Page fault
 - Sync, coerced, non-maskable, within, resuming
- Misaligned memory access
 - Sync, coerced, maskable, within, resuming/terminating
- Memory protection violation
 - Sync, coerced, non-maskable, within, resuming/terminating

133

Exceptions

- Illegal opcode
 - Sync, coerced, non-maskable, within, terminating
- Hardware malfunction
 - Async, coerced, non-maskable, within, terminating
- Power failure
 - Async, coerced, non-maskable, within, terminating

134

Precise exceptions

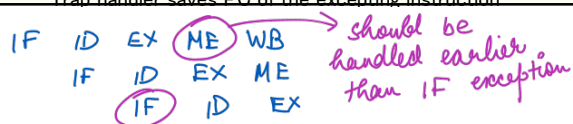
- Within instruction and restartable/resuming
 - Exception occurring in some pipeline stage
 - The exception must be taken transparently (save state, transfer control to OS, restore state, resume execution)
 - In a pipelined processor an instruction may take an exception deep into the pipeline (e.g. MEM stage); by this time quite a few subsequent instructions are already moving in the pipe
 - Each instruction carries an exception vector with it which tells if this instruction took an exception and if yes in which stage
 - The vector is examined at the end of MEM or beginning of WB stage; in case of a marked exception all pipe stages are fed with zeros (NOPs) to turn off any state change (e.g. memory write and register write)
 - A trap instruction is fetched and it transfers control to OS
 - Trap handler saves PC of the excepting instruction

Precise exceptions

- What is precise exception?
 - A processor is said to support precise exception if all instructions before the excepting instruction execute normally, all instructions after the excepting instruction do not change any programmer visible state of the processor, and after the exception is handled if it is restartable, execution must begin at the excepting instruction
 - Integer pipeline must implement restartable exceptions to be able to implement page faults and TLB misses
 - What about fp pipeline? Different latency of instructions makes it very hard; why?
 - Normally two floating-point modes are supported: imprecise and precise exception; in precise mode overlapping between fp instruction is limited (at least 10 times slower)

MAINAK CS422

136



Precise exceptions

- Five-stage MIPS integer pipeline
 - Which exceptions are possible in each pipe stage?
 - IF: page fault, memory protection; misaligned access?
 - ID/RF: illegal opcode
 - EX: arithmetic exception (signed overflow)
 - MEM: page fault, memory protection, misaligned access
 - WB: none
 - In the same cycle multiple instructions can take exceptions
 - Worse: exceptions can occur out of order (MEM and IF)
 - Exception vector associated with each instruction provides a way to handle these in order

137

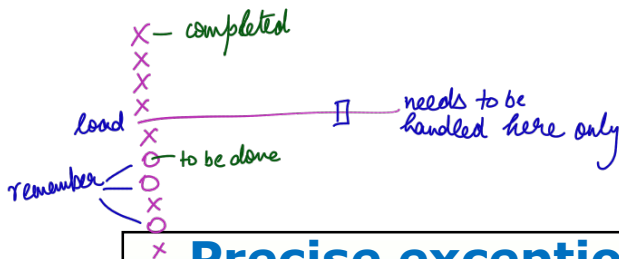
Precise exceptions

- What about branch delay slot?
 - Load in BD slot taking exception
 - How do you handle this? *bne lw* → need lw pc along with target pc
 - Two solutions
 - Let branch PC be the EPC
 - Remember multiple PCs and some more states

branch taken/not taken and branch target

MAINAK CS422

138

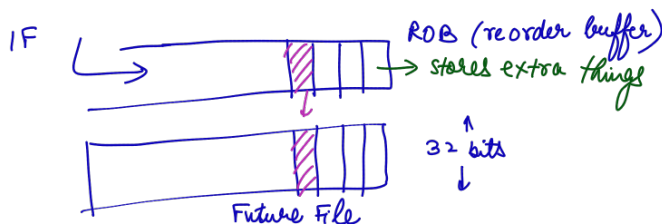


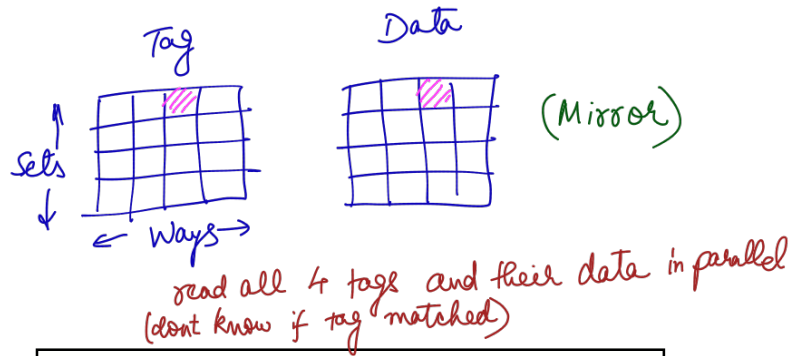
Precise exceptions

- What about the fp pipeline?
 - Out-of-order completion
 - Four possible solutions
 - Imprecise mode
 - History file (CYBER 180/990, VAX) and future file (P6 enhances it to retirement register file; used in Pentium Pro, Pentium II, III)
 - Ins1 comes before Ins2; Ins1 raises an exception *after* Ins2 has completed; we need to nullify the effect of Ins2
 - Let software handle preciseness i.e. finish incomplete instructions and ignore the completed ones; resume after the last completed instruction
 - Issue only if all instructions are guaranteed to complete without taking exceptions i.e. detect exception as early as possible (MIPS R2000, R3000, R4000, Intel Pentium)

Precise exceptions

- What about the fp pipeline?
 - A clever implementation of future file with in-order completion and out-of-order execution is used in all modern processors today
 - Precise exception is easy with in-order completion
 - Implementing in-order completion requires maintaining a FIFO queue of all fetched instructions (this queue has several names, re-order buffer or ROB for short being the most popular)
 - When an instruction comes to the head of the ROB and it has already completed execution, its result (if any) is copied from the future file to the main register file
 - Copy can be avoided with a clever future file implementation that employs register renaming (more later)
 - An ROB entry has a one-to-one correspondence with the instruction's future file entry





Pipelining a CISC ISA

- Widely varying latency of instructions
 - Magnifies the problems of fp pipeline by a large amount
 - Worse: data hazard *within* instruction (same register may be read and written to multiple times)
 - VAX 8800 invented microinstructions: translate CISC instruction to a sequence of RISC-like simple instructions; since 1995 IA-32 and later x86-64 use this technique
 - What about precise exceptions?
 - Looks extremely hard to support: instructions modify CPU states at different times and possibly multiple times
 - Think of a string copy instruction
 - Can use history or future file, but CISC makes that hard too
 - VAX decided to save and restore partially completed instructions: maintain state to decide where to start

MIPS R4000 family

- Implements 64-bit MIPS ISA
- One member of the family: R4400
- 8-stage pipeline (for faster clock decompose memory access)
 - IF: select PC, start instruction access
 - IS: instruction fetch → Tag Matching
 - RF: instruction cache hit detection, decode, hazard check and activate interlock if needed, register operand fetch
 - EX: branch (both condition and target), ALU, effective address of load/store
 - DF: data access
 - DS: data access
 - TC: data cache hit detection, store completion
 - WB: register write

Direct mapped caches

all hazard stall in RF stage

Pipeline stalls

- Load delay
 - 2 cycles (how?)

load \$2, 0(\$29)	IF	IS	RF	EX	DF	DS	TC	WB
add \$3, \$2, \$2		IF	IS	RF				
 - Widely used in all microprocessors today: load hit/miss speculation (R4000 uses blind speculation)
 - Worst case: 3 cycles; also hardware to back up by one cycle (miss may take longer; the "back up" hardware turns the dependent issued in *last* cycle to NOP, and then stalls the pipe until miss returns)
 - Pipeline interlock is implemented to stall dependent for 2 cycles

Best Case - 2 cycle
Worst Case - 3 cycle

143

Pipeline stalls

- Branch delay
 - 3 cycles

beq \$1, \$2, label	IF	IS	RF	EX	DF	DS	TC	WB
Ins?			IF					
Ins?				IF				
Ins?					IF			
Ins						IF		
 - One is filled by compiler (just after the branch): usual branch delay slot (support for backward compatibility)
 - During the next two cycles fetching continues from fall-through (predicted NT)
 - No direction predictor or BTB

144

Bypass network

- More wiring
 - How many sources and destinations?
 - IF IS RF EX DF DS TC WB
 - $N = ?$
 - $S = 4$
 - Bigger MUXes

MAINAK CS422

145

Floating-point pipe

- Three major units
 - Divider, multiplier, adder (*integer side has separate FU*)
 - Each instruction goes through eight phases visiting each phase zero or more times
 - Mantissa add (A): done in adder
 - Divide (D): done in divider
 - Exception test (E): done in multiplier
 - First stage of multiplication (M): done in multiplier
 - Second stage of multiplication (N): done in multiplier
 - Rounding (R): done in adder
 - Operand shift (S): done in adder
 - Unpack (U): unpack hardware (*IEEE 754: copy of mantissa etc*)

MAINAK CS422

146

Floating-point pipe

- Pipe stages (latency, repeat interval) *→ to match exp. normalization*
 - Add/subtract: U, S+A, A+R, R+S (4, 3)
 - Add1 U S+A A+R R+S
 - Add2 U
 - Multiply: U, E+M, M², N, N+A, R (8, 4)
 - Mult1 U E+M M M M N N+A R
 - Mult2 U E+M M M M N N+A R
 - Divide: U, A, R, D²⁷, D+A, D+R, D+A, D+R, A, R (36, 35)
 - Square root: U, E, (A+R)¹⁰⁸, A, R (112, 111)
 - Negate: U, S (2, 1)
 - Absolute: U, S (2, 1)
 - Compare: U, A, R (3, 2)
- Observe how structural hazard dictates the repeat interval

MAINAK CS422

147

Overall performance

- Branch stalls are more important than load stalls in most applications (SPEC92)
 - Need good branch predictors
- Floating-point RAW stalls are more important than structural stalls
 - Better to reduce latency of floating-point instructions (i.e. optimized algorithms) as opposed to more functional units or subunits
- Average CPI for SPECint92 on R4400: 1.54
 - +0.16 due to load stalls, +0.38 due to branch penalty
- Average CPI for SPECfp92 on R4400: 2.48
 - +0.01 due to load stalls, +0.33 due to branch penalty, +0.95 due to RAW stalls, +0.18 due to other stalls

MAINAK CS422

148

MIPS R4300

- Was popular in embedded market
 - Implements MIPS64 ISA
 - Five-stage integer pipe
 - Used in Nintendo-64 game engines, color laser printers, network processors
 - A very popular embedded processor NEC VR4122 is derived from it: borrows the integer pipe and uses software for floating-point
 - MIPS R4300 extends the integer pipe to execute floating-point instructions (multiple EX stages)
 - All instructions take equal number of cycles to finish
 - All integer instructions go through the floating-point execution stages doing nothing
 - Larger bypass network

MAINAK

CS422

149