B35APO: Computer Architectures

Lecture 06. Branches and Speculative Execution

Pavel Píša
pisa@fel.cvut.cz

Petr Štěpán
stepan@fel.cvut.cz

CTU
CZECH TECHNICAL UNIVERSITY IN PRAGUE

25. March, 2024

Outline

Today's Lecture Objective

- Familiarize with another possible processor speedup techniques that builds on pipelining – the superscalar architecture
- Branch prediction and speculative execution which are very important concepts of superscalar processors
- All of these techniques are used in RISC-V processors as well as in all current high performance processors

Recap - Quiz

Which of following instruction sequences executed on classic five-stage pipeline processor experience data hazards?

a)

```
addi t0, s1, 4
add  t1, s1, s0
add  s1, s2, x0
```

b)

```
addi t0, s1, 4
add  t1, s2, s3
add  t2, t0, t1
```

A in neither of sequences

B hazard is only in case a)

C hazard is only in case b)

D hazard is in case a) and b)

Recap - Quiz

How can be the following data hazard solved for the processor built during the last lecture?

```
lw   s2, 10(s0)
addi s1, s2, -1
```

A this hazard cannot be solved, it must be solved by the compiler or programmer

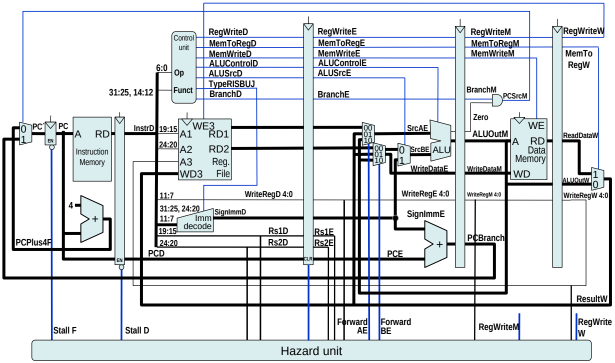
B can be solved by data forwarding

C must only be resolved using stall

D can be solved by a combination of stall and data forwarding

Processor with Pipeline (from Lecture 5)

How does the Hazard Unit make a decision about data hazard and their resolution?



- If RegWriteM==1, MemToRegM==0, WriteRegM!=0 and WriteRegM==RsE1 or RsE2 then set ForwardAE to 2 or ForwardBE to 2
- If RegWriteW==1, WriteRegM!=0 and WriteRegW==RsE1 or RsE2 then set ForwardAE to 1 or ForwardBE to 1

- If MemToRegE==1, RegWriteE==1 WriteRegE!=0 and WriteRegE==Rs1D or Rs2D then do not advance instruction from ID - STALL previous and flush (nop) pass to execute stage.

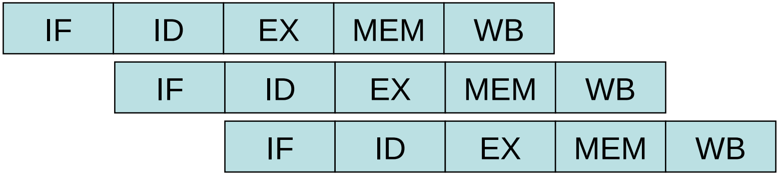
Instruction-level Parallelism

Instruction Level Parallelism (ILP)

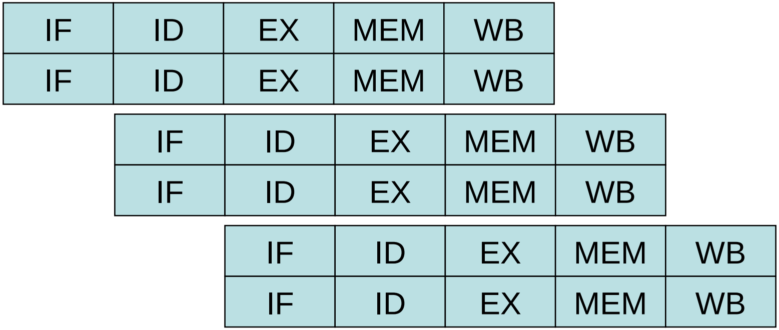
- Pipelining – different phases of different instructions are processed in parallel
- Superpipelining – superpipelining refers to pipelining with more than 10 steps. Slower pipelining phases are split into multiple parts, allowing the processor to increase its frequency and thus its performance.
- Superscalar processor – the same stages of different instructions are processed in parallel
  - multiple ALUs can execute in parallel EX phases of multiple different instructions
  - the fetch phase can fetch multiple sucessive instructions in parallel, i.e. instructions from PC, PC+4, PC+8 and PC+12 addresses
  - the instruction decode buffers allows fetching another instruction group even if previous ones cannot be passed into decode stages

Instruction-level Parallelism

Pipelined processor



Superscalar processor



Superscalar Processors

- Superscalar processors have an IPC (Instruction Per Clock) greater than 1.
    - Normal and pipelined processors have upper limit of IPC=1
- The number of instructions which can be processed in parallel and finished in the single clock cycle is called the instruction pipeline width.
- There are two basic variants:
    - **Static** superscalar architecture – only consecutive instructions in a program can run in parallel.
        - If instructions depend on any other in the group, this leads to a processor stall.
    - **Dynamic** superscalar architecture – any instructions that are ready to execute can run in parallel up to number of available functional units and their local pipelines limits.
        - Allows instructions to be executed out-of-order (in other than original program sequence).
        - Leads to better use of the processor's HW resources.

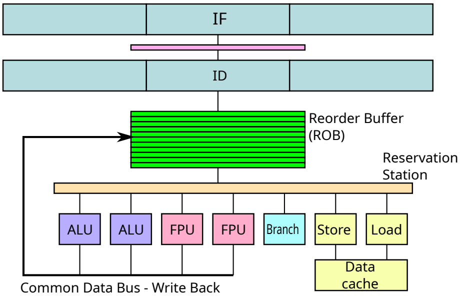
Superscalar Processors

- Individual parallel pipelines can be **unified** – that is, all pipelines are the same and can perform all types of operations
  - In practice, this would be an unnecessarily complex processor – it is not used
- The individual parallel pipelines are **diversified** (specialized) – each pipeline can only process certain subset of instructions:
  - register-only instructions – calculations, comparisons
  - instructions to access the memory – loading/storing data from/to memory
  - branch instructions – PC changing instructions

Outline

Superscalar Architecture

- The basis of the architecture is usually the ReOrder Buffer (ROB), which allows to achieve same final effect as when program is executed sequentially. Register renaming allows to overcome limitations caused by WAW (write-after-write) and WAR (write-after-read) hazards which are results of out of order execution.



- Reservation Stations allows to assign operations to functional units even when some of input operand values are not knows at dispatch time.
- Common Data Bus ensures that calculated values are written to the actual registers even for renamed registers and propagated to instructions waiting for them.

Data Hazards in Superscalar Architecture

For register-only instructions:

- is clear that it is not possible to execute all instructions in parallel
- is possible to increase parallelization by using a larger renamed register file.

```
1: slli t1, s1, 4
2: add  t0, t1, s2
3: addi s2, t0, 8
4: mult t1, s0, s0
5: addi t3, t1, 100
```

This program can be parallelized using renamed registers for intermediate results:

```
1: slli RN0, s1,  4
2: add  RN1, RN0, s2
3: addi RN2, RN1, 8
```

```
4: mult RN3, s0,  s0
5: addi RN4, RN3, 100
```

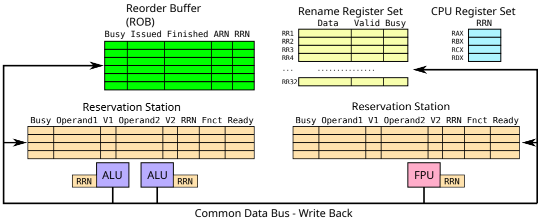
Tomasulo Algorithm

Robert Tomasulo from IBM invented an algorithm for out-of-order data processing on FPUs in 1967. Today, a modification of it is the basis of the architecture of all modern processors. The basic stages used today are:

- Loading an instruction into the ROB and renaming the result register of the operation - getting the register number from the renamed register file.
- Dispatch the instruction into the Reservation Station and wait for the required results of previous instructions
- Performing the calculation and writing the result to the renamed register via the Common Data Bus
- Completing the oldest instruction (circular queue – FIFO) from the ROB and updating the architectural register.

Instructions may be computed out-of-order, but instruction completion is in program order.

Superscalar Architecture



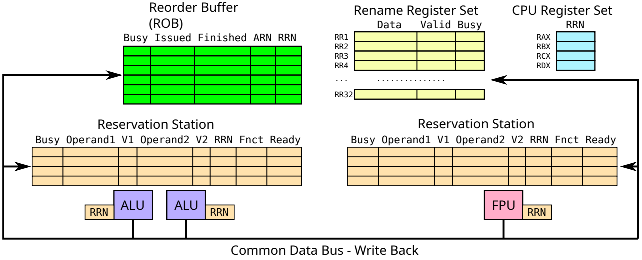
Rename Register File:

- A set of hardware registers, often many times more than architectural register file size.
- The Busy flag indicates that the register is being used, the Valid flag indicates that the value is calculated and is valid. When the instruction completes, the architectural register is set to Rename Register Number (RRN) value.

ReOrder Buffer:

- The cyclic queue contains instructions inserted in program order
- The queue ensures that instructions are completed in program order
- Binding between Reservation Station via Rename Register Number (RRN), which will contain the result of the operation
- Via the Common Data Bus is informed that the RRN register has been calculated and the instruction gets the Finished flag
- Complete instruction - instruction is removed from the ROB only when it and all preceding instructions are finished.

Superscalar Architecture



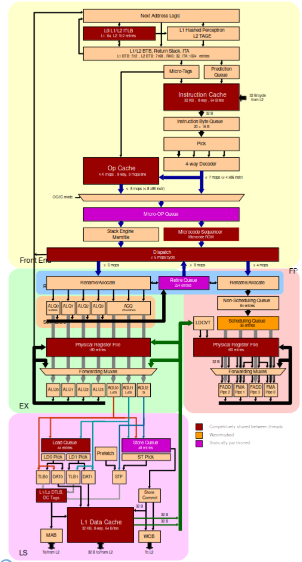
Reservation Station:

- Contains two operands, if the V1 or V2 (valid) flag is set then the operand represent value. If the V1 or V2 flag is not set, then the operand contains an RRN whose value is being waited for to be received from the CDB
- The RRN number indicates which register to write the result to.
- If both operands are valid and the functional unit is free, the operation is entered and the entry is removed from the Reservation Station. The RRN is passed with operation and data through the functional unit which can send it with result to the Common Data Bus and propagate results to waiting reservation stations.

Memory Read Data Hazards

- If the lw and sw instructions use different addresses, they can be reordered.
- If lw follows sw to the same address, data forwarding can be implemented.
- In practice, however, one instruction may overtake the other, so that it is not yet calculated where the data will be stored, i.e. whether a match will occur.
    - Solution - speculative execution of the lw instruction, i.e. execution even though it is not clear whether the data will be correct
    - When the sw instruction completes, all speculative executions of lw are checked
    - When a conflict is found, speculative execution of the lw instruction is cancelled
- Execution is treated as if the order is preserved.
- A big problem in multiprocessor systems – memory consistency when parallel computations are performed on different processor cores.

AMD Zen2 - Microarchitecture



- 7 nm process (from 12 nm), I/O die utilizes 12 nm
- Core (8 cores on CPU chiplet), 6/8/4 µOPs in parallel
  - Frontend, µOP cache (4096 entries)
  - FPU, 256-bit Eus (256-bit FMAs) and LSU (2x256-bit L/S), 3 cycles DP vector mult latency
  - Integer, 180 registers, 3x AGU, scheduler (4x16 ALU + 1x28 AGU)
  - Reorder Buffer 224 entries
- Memory subsystem
  - L1 i-cache and d-cache, 32 KiB each, 8-way associative
  - L2 512 KiB per core, 8-way,
  - L2 DTLB 2048-entry
  - 48 entry store queue

Autor: QuietRub
Source: `https://en.wikichip.org/wiki/amd/microarchitectures/zen_2`

Recap - Quiz

What is a control hazard?

A hazard that must be addressed by data forwarding

B hazard that must be handled by stall

C a situation in which the currently processed instruction must be discarded

D the problem of unstable results of logical operations

Outline

Control Hazards in Superscalar Architecture

- With conditional branches, it is not clear which instructions will be executed.
- Calculating the condition for a branch can take a long time, there are many instructions in progress.
  - Solution - speculative fetching of additional instructions
  - After all the calculations necessary for the branch decision are completed, it is checked whether branch should be really taken or not.
  - If the prediction is wrong, speculatively executed instructions must be discarded/flushed.
- Even unconditional branches/jumps have trouble to know branch target. The branch targed may depend on the computation of previous instructions for jump to register, and therefore cannot be easily determined when fetching instructions.
  - Solution - speculatively guess branch target address, based on the history of past branches.
  - After all calculations are done, check if the correct address has been predicted.
  - Especially important for returning from a function.

Superscalar Architecture Control hazards

- A branch is statistically every 4 to 7 instructions in the program
- 20% of branches are unconditional – they always taken, no need to decide
- 80% of branches are conditional
    - about 66% are branches to a higher address, or forward
        - these branches correspond to branching type `if`
        - of these, statistically about 60% are – we will denote **NT** (Not Taken)
    - the rest about 34% are branches to a lower address, or backwards
        - these branches correspond to branches of the type `for`, `while` and `do ... while`
        - of these, statistically 99% (almost all) will be taken – we will denote **T** (Taken)

Static Predictors

Static predictors always have the same result for a given branch instruction:

- A predictor that would always estimate that it always taken
  - According to the statistics on the previous page, it would have a success rate of $p_{taken} = (0.66 * 0.4 + 0.34 * 0.99) = 0.60$
  - Statistically, it turns out that p_{taken} is in the range of $60 - 70\%$ for most programs.
- BTFNT predictor – Backwards Taken / Forwards Not Taken
  - According to the sign of the relative branches - backward branch is predicted taken, forward branches is predicted
  - According to the stats on the previous page, it would have a success rate of $p_{taken} = (0.66 * 0.6 + 0.34 * 0.99) = 0.73$

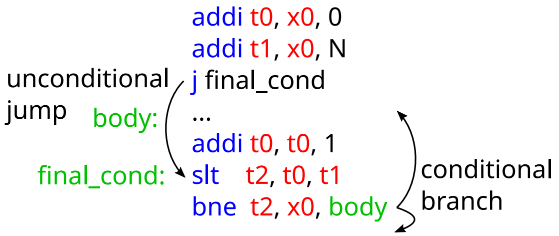
Static Predictor BTFNT

Example of translating a `for` cycle:

```
for (i=0; i<N; i++) {

  ...

}
```



```
            addi t0, x0, 0
            addi t1, x0, N
            j final_cond
body:       ...
            addi t0, t0, 1
final_cond: slt  t2, t0, t1
            bne  t2, x0, body
```

For a conditional branch, $N-1$ times taken and 1 is not taken.
It branch backwards and the probability of a correct prediction is $\frac{N-1}{N}$

Static Predictor BTFNT

Example of translating the `if else` construct:

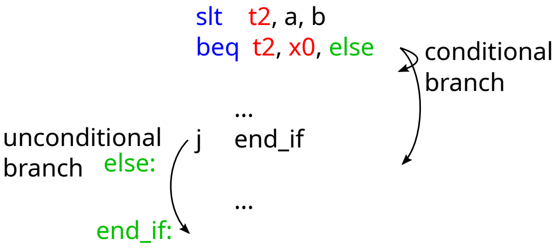
```
if (a<b) {

   ...

} else {

   ...

}
```



```
        slt   t2, a, b
        beq   t2, x0, else

              ...
        j     end_if
else:

              ...
end_if:
```

The unconditional branch depends on the values of a, b, so nothing can be said about it in general, except that it always jumps forward.
The statistical behavior depends on the type of program, but for a mixture of different programs it turns out that the probability of a taken branch is 0.4.

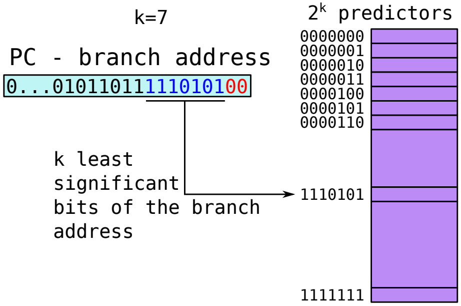
Dynamic Branch Predictors

Dynamic predictors try to estimate whether a branch will occur based on the past behavior of a given branch instruction:

- It would be ideal if each branch instruction had its own predictor
  - But this is not possible, a branch instruction can be at any location in 4GiB memory

Solution: we will have 2^k predictors and select a predictor according to the k lowest bits of the branch instruction address

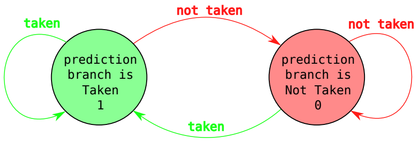
- Some branch instructions at different addresses but with the same lowest k bits of the address will interfere – **interfere**.
- Interference can very adversely affect prediction success.



1-bit Smith Predictor

The simplest predictor is the 1-bit Smith predictor

- Has only two states, switches according to past behavior
- Predicts that it will turn out the same way it did last time.
    - very simple to implement, evaluate and adjust to reality


taken
not taken
not taken
prediction branch is Taken 1
prediction branch is Not Taken 0
taken

1-bit Smith Predictor

Prediction of the for loop:

```
for (i=0; i<5; i++) {
  ...
}
```

```
      addi t0, x0, 0
      addi t1, x0, 5
      j cond
body: ...
      addi t0, t0, 1
cond: slt  t2, t0, t1
      bne  t2, x0, body
```

If the predictor does not interfere with other branches, then it starts in state 0 – NT (Not Taken).

Real behaviour	T	T	T	T	T	NT
Prediction	NT	T	T	T	T	T

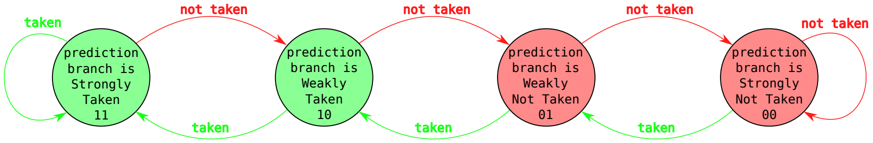
You can see that the predictor is not successful at the beginning of the for loop and at the end.

The success rate of the 1-bit Smith predictor for a loop with r iterations is $\frac{r-2}{r}$.

2-bit Smith Predictor

The 2-bit Smith predictor is still one of the simplest used ones

- It already has 4 states, represented by 2 bits.
- 2 states predict a taken, 2 states predict a non-taken
- Prediction depends on past behaviour, but tolerates one deviation from regularity
- Very simple to implement, evaluate and adjust to reality


taken
not taken
not taken
not taken
not taken
prediction branch is Strongly Taken 11
prediction branch is Weakly Taken 10
prediction branch is Weakly Not Taken 01
prediction branch is Strongly Not Taken 00
taken
taken
taken

2-bit Smith predictor

Prediction for for loop:

```
for (i=0; i<5; i++) {
  ...
}
```

```
      addi t0, x0, 0
      addi t1, x0, 5
      j cond
body: ...
      addi t0, t0, 1
cond: slt  t2, t0, t1
      bne  t2, x0, body
```

If the predictor does not interfere with other branches, then it starts in state 10 – WT (Weakly taken).

Real behaviour	T	T	T	T	T	NT	
State	WT	ST	ST	ST	ST	ST	WT
Prediction	T	T	T	T	T	T	

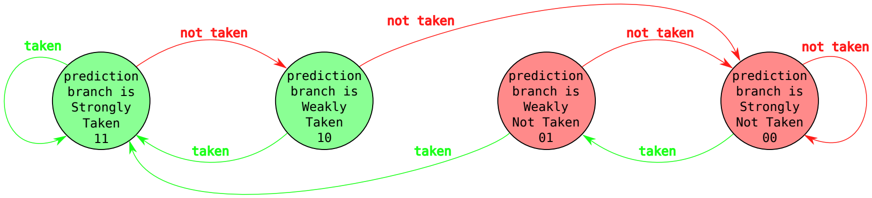
You can see that the predictor is not successful only at the end of the for loop.

The success rate of the 2-bit Smith predictor for a loop with r iterations is $\frac{r-1}{r}$.

2-bit Smith Predictor with Hysteresis

Analogue of the 2-bit Smith predictor

- When two changes occur in succession, it goes straight to the Strongly state and two more opposite behaviors must occur in succession for the predictor to flip to the new state.


taken
not taken
not taken
not taken
not taken
prediction branch is Strongly Taken 11
prediction branch is Weakly Taken 10
prediction branch is Weakly Not Taken 01
prediction branch is Strongly Not Taken 00
taken
taken
taken

Predictor Evaluation

- It's impossible to decide in general which of predictors is better. It is always possible to find counterexamples where each of the predictors behaves counterproductive
- The only possibility is a statistical analysis of different programs:

Static predictor - always taken	59.25
1-bit Smith predictor	68.75
2-bit Smith predictor with hysteresis	81.75

Source: `https://ieeexplore.ieee.org/document/6918861`
H. Arora, S. Kotecha and R. Samyal, "Dynamic Branch Prediction Modeller for RISC Architecture," 2013 International Conference on Machine Intelligence and Research Advancement, Katra, 2013, pp. 397-401.

Prediction Interdependence/Correlation

In practice, it turns out that the prediction depends on the previous behavior of the program.

```
if (x==2) { // jump s1
}
if (y==2) { // jump s2
}
if (x!=y) { // jump s3
}
```

If the variables x, y do not change in the bodies of the conditions s1 and s2, then we have a strong dependency of the branches s3:

s1	s2	$\Longrightarrow$ s3	explanation
not taken	taken	not taken	x==2 and y!=2 therefore x!=y
taken	not taken	not taken	x!=2 and y==2 therefore x!=y
not taken	not taken	taken	x==2 and y==2, $\rightarrow$ x!=y is wrong
taken	taken	don't know	x!=2 and y!=2, don't know if x!=y

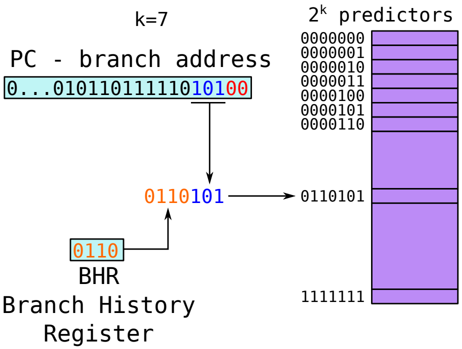
Branch History – Correlated Predictors

The Branch Histrory Register (BHR) contains information about whether the last *m* braches has been taken or not:

- If taken then contains 1, if not taken then contains 0
- The new information is inserted at the lowest bit, the oldest information is discarded from the highest position, the other information is shifted.

To find the predictor index, the $k - m$ lowest bits of the branch instruction address are used and this information is concatenated with the bits from the BHR

- Advantage – a different predictor is selected based on the previous branch outcome.
- Disadvantage – some combinations do not occur in the BHR and therefore these predictors are not



GShare Predictors

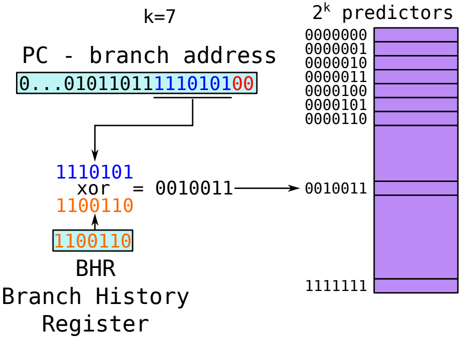
The GShare approach is similar to the previous one:

- It also uses the BHR register, which in this implementation has directly *k* bits

To find the predictor index, the *k* lowest bits of the branch instruction address are taken and xor is performed with the bits from the BHR.

Benefits:

- better statistically distributes to all predictors.
- allows to use a longer BHR register.



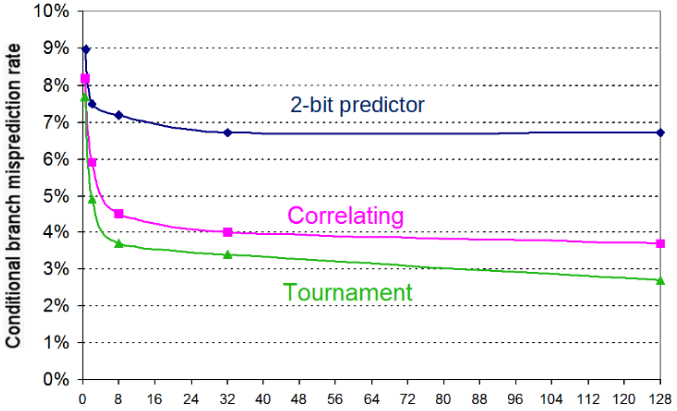
Tournament Predictors

The basis of a tournament predictor is a competition between two other, simpler predictors. To choose which predictor is better, and therefore which predictor will be output, a 1-bit or 2-bit predictor can be used.

How the 1-bit tournament predictor works

- Calculate the prediction with predictors P1 and P2.
- If the results are the same, the prediction is used.
- If the predictions is incorrect:
  - The resulting prediction is whichever predictor has been successful in the past. This information is stored in a 1-bit state machine.
  - Select predictor P1 or P2 for the next prediction, depending on the outcome.

Tournament Predictors



Size of predictors in KiB

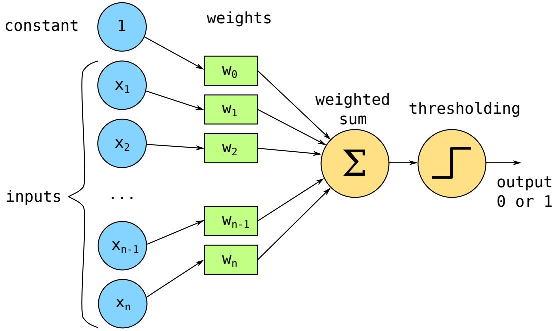
Recap - Quiz

Which predictor can best predict the following fact of jumps if it starts in Taken or Weakly Taken state:

T	T	T	NT	NT	NT	T	T	T

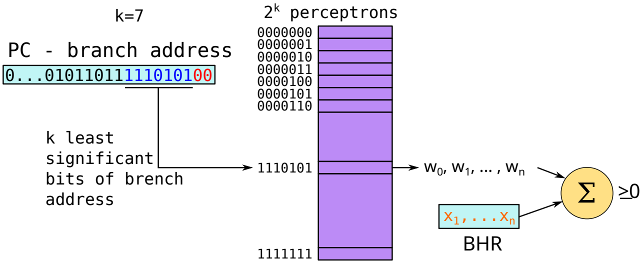
- A 1-bit Smith predictor
- B 2-bit Smith predictor
- C 2-bit Smith predictor with hysteresis
- D all make the same number of wrong predictions

Perceptron



The formula $\xi = \sum_{i=0}^{n} x_i \cdot w_i$ is used to calculate the value. The final evaluation is by means of a transfer function, in our case threshold function, which has the form: $f(\xi) = \begin{cases} 1 & \text{for } \xi \geq 0 \\ 0 & \text{for } \xi < 0 \end{cases}$

AMD Zen2 Branch Predictor

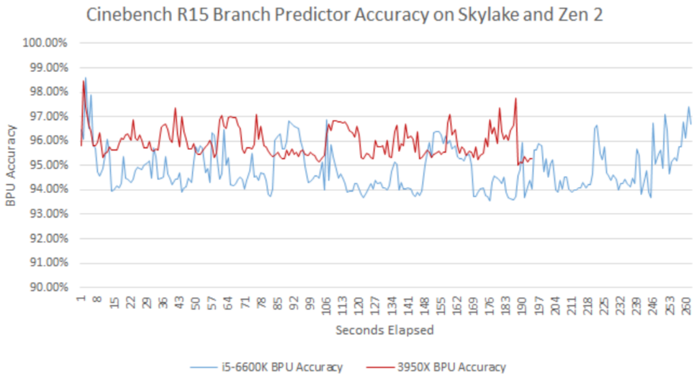


According to the branch address, one perceptron is selected from the table. The perceptron is defined by the weights w_i.
The prediction result is the sign of the weighted sum $\xi = \sum_{i=0}^{n} x_i \cdot w_i$, where x_i are bits from the BHR branch history register.
Advantage – better results than gshare, can be used for long BHR history registers, disadvantage – complex calculation, cannot get the result in one CPU clock.

AMD Zen2 Branch Predictor

- Calculating the perceptron output is relatively slow, normal perceptrons use real floating point numbers. It is possible to speed up with 16-bit real numbers, or by using a fixed decimal point.
- The actual implementation of branch predictors has three levels
  - level 1 – 16 very fast predictors, deciding in the same clock cycle which next instructions to fetch
  - level 2 – 512 predictors, which will refine the prediction in the next clock cycle, either discarding the loaded instructions or confirming them
  - level 3 – 7168 predictors, in 4 clock cycles refine the prediction. Again, the existing fetched instructions are either kept or discarded.
- The average time delay for a bad final prediction is approximately 18 clock cycles.
- At 4GHz and if every tenth instruction is a jump, a 1% misprediction results in a performance degradation of almost 2%.

AMD Zen2 Predictor



Source: Analyzing Zen 2's Cinebench R15 Lead By clamchowder from `https://chipsandcheese.com`

Outline

Branch Target Prediction

Branches have different branch target address formats in both RISC-V and other processors:

- branch/jump to a fixed address – the branch target is the address directly specified in the branch instruction (not in RISC-V because it has a fixed instruction length and a 32-bit address does not fit in the instruction code)
- branch relative to the address of the jump instruction – the branch target is calculated as the sum of the PC at the time the branch instruction is fetched and the specific value specified in the instruction.
- branch to the position specified in the register or in memory – RISC-V has the `jalr` instruction, i.e. jump to the address specified in the register, x86 contains the `ret` instruction – return from the subroutine according to the address specified in memory on the stack and the `jump indirect` instruction, where the address points to the memory where the jump target address is specified. The indirect jump instruction can be used when branching according to a table, which can be used when compiling a `switch` construct, or when calling dynamic library functions or virtual methods of the object.

Branch Target Prediction

- The branch target needs to be determined at the instruction fetch stage.
- It is possible to have special dedicated adders for jump target addresses, but even there the summation takes some time.
- Therefore it is important to estimate the branch target:
  - BTB (Branch Target Buffer) is either a fully associative memory or a partially associative with a given degree of associativity.
  - Lines are pairs:
    - key – BIA (Branch Instruction Address) – i.e. the PC value at the time of the branch
    - BTA (Branch Target Address) – the target address of the branch
- If the PC matches the value of the BIA in the BTB and the predictor simultaneously predicts a branch, the PC is speculatively changed to the value of the corresponding BTA

Function Return Prediction

The most common branch accoring address in register or memory is a return from function:

- For fast prediction of the return address of a function, modern CPUs contain a – Return Address Stack (RAS)
  - This is a fast stack memory – remembering a limited number (up to 32) of return addresses
  - The value is stored in the RAS when the function is called, then when the ret instruction is fetched, the top of the RAS serves as a predictor of the branch target
- Works reliably for function nesting levels depending on RAS size

Outline

Example of how to remove a jump in a program

You can find many tricks on the Internet to remove condition and therefore conditional jumps in your programs.
Here we will demonstrate removing the if from the calculation of the absolute value of an integer:

program in C

```
if (x<0) {
  x = -x;
}
```

program in RISC-V

```
 slt t0, s0, x0
 beq t0, x0, skip
 sub s0, x0, s0
skip:
```

comment compares whether x<0 will jump or not according to the result. calculate -x

To remove the conditional jump, which would be very hard to estimate, the following construction can be used, using the highest sign bit:

program in C

```
int tmp = x>>31;
x^= tmp;
x -= tmp;
```

program in RISC-V

```
srai t0, s0, 31
xor  s0, s0, t0
sub  s0, s0, t0
```

comment
tmp will be either 0 or 0xFFFFFFFFF
does nothing, or bitwise inversion for tmp==0 it does nothing, otherwise it subtracts -1, so it adds 1

Example of How to Remove a Branch in a Program

If the value of b is 0 or 1, then the following C program can be executed:

```
a = ( (b!=0) ? c : d);
```

change to program:

```
static const int lookup_table[] = {d,c};
a = lookup_table[b];
```

Multiple branches can also be eliminated at once, as long as again b1, b2, b3 only take values 0 or 1:

```
a = ( b1 ? c : ( b2 ? d : (b3 ? e : f)));
```

can be changed to a program:

```
static const int lookup_table[] = { f, e, d, d, c, c, c, c };
a = lookup_table[b1 * 4 + b2 * 2 + b3];
```

Example of how to remove a branch in a program

Similarly, converting a number from 0 to 15 to a hex character can either

```
if (a<10) {
  ch = '0'+a;
} else {
  ch = 'A'+(a-10);
}
```

can be changed to a program:

```
static const int hex_c[] = {'0','1','2','3','4','5','6','7',
                            '8','9','A','B','C','D','E','F'};
ch = hex_c[a];
```