CMSC 611: Advanced Computer Architecture

Instruction Level Parallelism

Some material adapted from Mohamed Younis, UMBC CMSC 611 Spr 2003 course slides
Some material adapted from Hennessy & Patterson / © 2003 Elsevier Science
Floating-Point Pipeline

• Impractical for FP ops to complete in one clock
  – (complex logic and/or very long clock cycle)

• More complex hazards
  – Structural
  – Data
Multi-cycle FP Pipeline

7-stage pipelined FP multiply

Non-pipelined DIV operation stalling the whole pipeline for 24 cycles

Pipelined FP addition with latency of 3 cycles

Example: blue indicate where data is needed and red when result is available
Multi-cycle FP: EX Phase

- **Latency**: cycles between instruction that produces result and instruction that uses it
  - Since most operations consume their operands at the beginning of the EX stage, latency is usually number of the stages of the EX an instruction uses
- **Long latency** increases the frequency of RAW hazards
- **Initiation (Repeat) interval**: cycles between issuing two operations of a given type

<table>
<thead>
<tr>
<th>Functional unit</th>
<th>Latency</th>
<th>Initiation interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer ALU</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Data memory (integer and FP loads)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>FP add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>FP multiply (also integer multiply)</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>FP divide (also integer divide)</td>
<td>24</td>
<td>25</td>
</tr>
</tbody>
</table>
FP Pipeline Challenges

• Non-pipelined divide causes structural hazards
• Number of register writes required in a cycle can be larger than 1
• WAW hazards are possible
  – Instructions no longer reach WB in order
• WAR hazards are **NOT** possible
  – Register reads are still taking place during the ID stage
• Instructions can complete out of order
  – Complicates exceptions
• Longer latency makes RAW stalls more frequent

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Clock cycle number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17</td>
</tr>
<tr>
<td>LD F4, 0(R2)</td>
<td>IF</td>
</tr>
<tr>
<td>MULTD F0, F4, F6</td>
<td>IF</td>
</tr>
<tr>
<td>ADDD F2, F0, F8</td>
<td>IF</td>
</tr>
<tr>
<td>SD 0(R2), F2</td>
<td>IF</td>
</tr>
</tbody>
</table>

*Example of RAW hazard caused by the long latency*
## WB Structural Hazard

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Clock cycle number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>MULTD F0, F4, F6</td>
<td>IF</td>
</tr>
<tr>
<td>...</td>
<td>IF</td>
</tr>
<tr>
<td>...</td>
<td>IF</td>
</tr>
<tr>
<td>ADDD F2, F4, F6</td>
<td>IF</td>
</tr>
<tr>
<td>...</td>
<td>IF</td>
</tr>
<tr>
<td>...</td>
<td>IF</td>
</tr>
<tr>
<td>LD F2, 0(R2)</td>
<td>IF</td>
</tr>
</tbody>
</table>

- At cycle 11, the MULTD, ADDD and LD instructions will try to write back
  - structural hazard if there is only one write port
- Additional write ports are not cost effective since they are rarely used
- Instead
  - Detect at ID and stall
  - Detect at MEM or WB and stall
**WAW Data Hazards**

### Instruction Schedule

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Clock cycle number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>MULTD F0, F4, F6</td>
<td>IF</td>
</tr>
<tr>
<td>...</td>
<td>IF</td>
</tr>
<tr>
<td>...</td>
<td>IF</td>
</tr>
<tr>
<td>ADDD F2, F4, F6</td>
<td>IF</td>
</tr>
<tr>
<td>...</td>
<td>IF</td>
</tr>
<tr>
<td>LD F2, 0(R2)</td>
<td>IF</td>
</tr>
<tr>
<td>....</td>
<td>IF</td>
</tr>
</tbody>
</table>

- WAW hazards can be corrected by either:
  - Stalling the latter instruction at MEM until it is safe
  - Preventing the first instruction from overwriting the register
- Correcting at cycle 11 OK unless intervening RAW/use of F2
- WAW hazards can be detected at the ID stage
  - Convert 1st instruction to no-op
- WAW hazards are generally very rare, designers usually go with the simplest solution
Detecting Hazards

- Hazards among FP instructions & and combined FP and integer instructions
- Separate int & fp register files limits latter to FP load and store instructions
- Assuming all checks are to be performed in the ID phase:
  - Check for structural hazards:
    - Wait if the functional unit is busy (Divides in our case)
    - Make sure the register write port is available when needed
  - Check for a RAW data hazard
    - Requires knowledge of latency and initiation interval to decide when to forward and when to stall
  - Check for a WAW data hazard
    - Write completion has to be estimated at the ID stage to check with other instructions in the pipeline
- Data hazard detection and forwarding logic from values stored between the stages
Maintaining Precise Exceptions

• Pipelining FP instructions can cause out-of-order completion

• Exceptions also a problem:
  
  DIVF    F0, F2, F4
  ADDF    F10, F10, F8
  SUBF    F12, F12, F14

  – No data hazards
  – What if DIVF exception occurs after ADDF writes F10?
Four FP Exception Solutions

1. Settle for imprecise exceptions
   - Some supercomputers still use this approach
   - IEEE floating point standard requires precise exceptions
   - Some machines offer slow precise and fast imprecise exceptions

2. Buffer the results of all operations until previous instructions complete
   - Complex and expensive design (many comparators and large MUX)
   - History or future register file
Four FP Exception Solutions

3. Allow imprecise exceptions and get the handler to clean up any miss
   - Save PC + state about the interrupting instruction and all out-of-order completed instructions
   - The trap handler will consider the state modification caused by finished instructions and prepare machine to resume correctly
   - Issues: consider the following example
     Instruction1: Long running, eventual exception
     Instructions 2 … (n-1): Instructions that do not complete
     Instruction n: An instruction that is finished
   - The compiler can simplify the problem by grouping FP instructions so that the trap does not have to worry about unrelated instructions
Four FP Exception Solutions

4. Allow instruction issue to continue only if previous instruction are guaranteed to cause no exceptions:
   - Mainly applied in the execution phase
   - Used on MIPS R4000 and Intel Pentium
Stalls/Instruction, FP Pipeline

- doduc: Add/subtract/convert: 1.7, Compares: 1.7, Multiply: 3.7, Divide: 15.4
- ear: Add/subtract/convert: 1.6, Compares: 2.0, Multiply: 2.5, Divide: 12.4
- hydro2d: Add/subtract/convert: 2.3, Compares: 2.5, Multiply: 3.2, Divide: 0.0
- mdilfp: Add/subtract/convert: 2.1, Compares: 1.2, Multiply: 2.9, Divide: 0.0
- su2cor: Add/subtract/convert: 0.7, Compares: 1.5, Multiply: 1.6, Divide: 0.6

Number of stalls range from 0.0 to 25.0.
This figure (A.36 in the book) contains several errors in either graph or data. Only take-home: result stalls are most common by far.
Instruction Level Parallelism (ILP)

- Overlap the execution of unrelated instructions
- Both instruction pipelining and ILP enhance instruction throughput not the execution time of the individual instruction
- Potential of IPL within a basic block is very limited
  - In "gcc" 17% of instructions are control transfer meaning on average 5 instructions per branch
Loops: Simple & Common

for (i=1; i<=1000; i=i+1)
  x[i] = x[i] + y[i];

- Techniques like loop unrolling convert loop-level parallelism into instruction-level parallelism
  - statically by the compiler
  - dynamically by hardware
- Loop-level parallelism can also be exploited using vector processing
- IPL feasibility is mainly hindered by data and control dependence among the basic blocks
- Level of parallelism is limited by instruction latencies
Major Assumptions

- Basic MIPS integer pipeline
- Branches with one delay cycle
- Functional units are fully pipelined or replicated (as many times as the pipeline depth)
  - An operation of any type can be issued on every clock cycle and there are no structural hazard

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using results</th>
<th>Latency in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store Double</td>
<td>2</td>
</tr>
<tr>
<td>Load Double</td>
<td>FP ALU op</td>
<td>1</td>
</tr>
<tr>
<td>Load Double</td>
<td>Store Double</td>
<td>0</td>
</tr>
</tbody>
</table>
Compilation Overview

Source -> Scanner -> Parser -> Static Semantics Analyzer -> Code generator -> Optimizer

Control flow graph
Call graph
Data dependence representation
M/C code

Control flow graph

Start
Basic Block
Condition
Basic Block
Call subroutine
Basic Block
End

Call graph

A -> B
D
C

Data dependence representation

Z = A
A use X1

M/C code

X1: A = B
new def. for A

X2: A = C
new def. for A

Call subroutine

Y = A + 3
A use X1 or X2
Motivating Example

for (i=1000; i>0; i=i-1)
    \( x[i] = x[i] + s \);

Standard Pipeline execution

Loop:
- LD F0,x(R1)
- ADDD F4,F0,F2
- SD x(R1),F4
- SUBI R1,R1,8
- BNEZ R1,Loop

Smart compiler

Loop:
- LD F0,x(R1)
- ADDD F4,F0,F2
- SUBI R1,R1,8
- BNEZ R1,Loop
- SD x+8(R1),F4

Sophisticated compiler optimization reduced execution time from 10 cycles to only 6 cycles
Loop Unrolling

Replicate loop body 4 times, will need cleanup phase if loop iteration is not a multiple of 4

Loop:
LD F0,x(R1)
ADDD F4,F0,F2
SD x(R1),F4
SUBI R1,R1,8
BNEZ R1,Loop

• 6 cycles, but only 3 are loop body
• Loop unrolling limits overhead at the expense of a larger code
  – Eliminates branch delays
  – Enable effective scheduling
• Use of different registers needed to limit data hazard

Loop:
LD F0,x(R1)
ADDD F4,F0,F2
SD x(R1),F4 ;drop SUBI & BNEZ
LD F6,x-8(R1)
ADDD F8,F6,F2
SD x-8(R1),F8 ;drop again
LD F10,x-16(R1)
ADDD F12,F10,F2
SD x-16(R1),F12 ;drop again
LD F14,x-24(R1)
ADDD F16,F14,F2
SD x-24(R1),F16
SUBI R1,R1,#32 ;alter to 4*8
BNEZ R1,LOOP
Scheduling Unrolled Loops

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Instruction</th>
<th>Cycle</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Loop: LD F0,x(R1)</td>
<td>1</td>
<td>Loop: LD F0,x(R1)</td>
</tr>
<tr>
<td>3</td>
<td>ADDD F4,F0,F2</td>
<td>2</td>
<td>LD F6,x-8(R1)</td>
</tr>
<tr>
<td>6</td>
<td>SD x(R1),F4</td>
<td>3</td>
<td>LD F10,x-16(R1)</td>
</tr>
<tr>
<td>7</td>
<td>LD F6,x-8(R1)</td>
<td>4</td>
<td>LD F14,x-24(R1)</td>
</tr>
<tr>
<td>9</td>
<td>ADDD F8,F6,F2</td>
<td>5</td>
<td>ADDD F4,F0,F2</td>
</tr>
<tr>
<td>12</td>
<td>SD x-8(R1),F8</td>
<td>6</td>
<td>ADDD F8,F6,F2</td>
</tr>
<tr>
<td>13</td>
<td>LD F10,x-16(R1)</td>
<td>7</td>
<td>ADDD F12,F10,F2</td>
</tr>
<tr>
<td>15</td>
<td>ADDD F12,F10,F2</td>
<td>8</td>
<td>ADDD F16,F14,F</td>
</tr>
<tr>
<td>18</td>
<td>SD x-16(R1),F12</td>
<td>9</td>
<td>SD x(R1),F4</td>
</tr>
<tr>
<td>19</td>
<td>LD F14,x-24(R1)</td>
<td>10</td>
<td>SD x-8(R1),F8</td>
</tr>
<tr>
<td>21</td>
<td>ADDD F16,F14,F2</td>
<td>11</td>
<td>SUBI R1,R1,#32</td>
</tr>
<tr>
<td>24</td>
<td>SD x-24(R1),F16</td>
<td>12</td>
<td>SD x-16(R1),F12</td>
</tr>
<tr>
<td>25</td>
<td>SUBI R1,R1,#32</td>
<td>13</td>
<td>BNEZ R1,LOOP</td>
</tr>
<tr>
<td>27</td>
<td>BNEZ R1,LOOP</td>
<td>14</td>
<td>SD x+8(R1),F1</td>
</tr>
<tr>
<td>28</td>
<td>stall</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Loop unrolling exposes more computation that can be scheduled to minimize the pipeline stalls.

Understanding dependence among instructions is the key for detecting and performing the transformation.
Inter-instruction Dependence

• Determining how one instruction depends on another is critical not only to the scheduling process but also to determining how much parallelism exists.

• If two instructions are parallel they can execute simultaneously in the pipeline without causing stalls (assuming there is not structural hazard).

• Two instructions that are dependent are not parallel and their execution cannot be reordered.
Dependence Classifications

• Data dependence (RAW)
  – Transitive: \( i \rightarrow j \rightarrow k = i \rightarrow k \)
  – Easy to determine for registers, hard for memory
    • Does \( 100(R4) = 20(R6) \)?
    • From different loop iterations, does \( 20(R6) = 20(R6) \)?

• Name dependence (register/memory reuse)
  – Anti-dependence (WAR): Instruction \( j \) writes a register or memory location that instruction \( i \) reads from and instruction \( i \) is executed first
  – Output dependence (WAW): Instructions \( i \) and \( j \) write the same register or memory location; instruction ordering must be preserved

• Control dependence, caused by conditional branching
Example: Name Dependence

Loop:

LD    F0,x(R1)
ADDD  F4,F0,F2
SD    x(R1),F4
LD    F0,x-8(R1)
ADDD  F4,F0,F2
SD    x-8(R1),F4
LD    F0,x-16(R1)
ADDD  F4,F0,F2
SD    x-16(R1),F4
LD    F0,x-24(R1)
ADDD  F4,F0,F2
SD    x-24(R1),F4
SUBI  R1,R1,#32
BNEZ  R1,Loop

Register renaming

Loop:

LD    F0,x(R1)
ADDD  F4,F0,F2
SD    x(R1),F4
LD    F6,x-8(R1)
ADDD  F8,F6,F2
SD    x-8(R1),F8
LD    F10,x-16(R1)
ADDD  F12,F10,F2
SD    x-16(R1),F12
LD    F14,x-24(R1)
ADDD  F16,F14,F2
SD    x-24(R1),F16
SUBI  R1,R1,#32
BNEZ  R1,LOOP

• Again Name Dependencies are Hard for Memory Accesses
  – Does 100(R4) = 20(R6)?
  – From different loop iterations, does 20(R6) = 20(R6)?
• Compiler needs to know that R1 does not change → 0(R1)≠ -8(R1)≠ -16(R1)≠ -24(R1)
  and thus no dependencies between some loads and stores so they could be moved
Control Dependence

- Control flow of a program is enforced by conditional branching.
- Two constraints on control dependences:
  - An instruction control dependent on a branch cannot be moved before the branch.
  - An instruction not control dependent on a branch cannot be moved after the branch.
- Observing control dependence is not a must.
  - But preserve program correctness in exception behavior and data flow.

```
X1: A = B
   new def. for A

Condition

Z = A
     A use X1

X2: A = C
    new def. for A

Call subroutine

Y = A + 3
    A use X1 or X2
```
Preserving Correctness

• Preserving exception behavior means that any changes in the ordering of instruction execution must not change how exceptions are raised

  BEQZ R2, L1  LW  R1, 0(R2)
  LW  R1, 0(R2)  BEQZ R2, L1

  – LW may cause memory protection exception

• Data flow reflects changes to variables (registers and memory) throughout the program
Loop-level Parallelism

- Loop-level parallelism is normally analyzed at the source code level or close to it while most analysis of IPL after code generation.
- Loop level analysis involves determining what dependences exist among the operands in the loop across iterations of the loop (bulk is data dependence).

```plaintext
for (i=1000; i>0; i=i-1) 
x[i] = x[i] + s;
```

All dependences are within the same iteration and thus loop is parallel.
Loop-level Parallelism

• Loop-carried dependence is caused by data dependence between operands modified in consecutive iterations.

Example: Assuming A, B, C are distinct and non-overlapping for (i=1; i<=100; i=i+1) {
    A[i+1] = A[i] + C[i]; /* S1 */
    B[i+1] = B[i] + A[i+1]; /* S2 */

- S2 uses the value, A[i+1], computed by S1 in the same iteration
- S1 uses a value computed by S1 in an earlier iteration

• Loop-carried dependence can be efficiently handled by loop unrolling.
Branching Dilema

• With the increased pipeline throughput, control dependence rapidly becomes the limiting factor to the amount of ILP

• For pipelines that issue n-instructions per clock cycle, the negative impact of stalls caused by control hazards magnifies

• Compiler-based techniques rely on static program properties to handle control hazards

• Hardware-based techniques refer to the dynamic behavior of the program to predict the outcome of a branch
Branch Target Cache

• Predict not-taken: still stalls to wait for branch target computation
• If address could be guessed, the branch penalty becomes zero
• Cache predicted address based on branch address
• Complications for complex predictors: do we know in time?
Branch Target Cache

- PC of instruction to fetch
- Look up
- Predicted PC
- Number of entries in branch-target buffer

Flow:
- No: instruction is not predicted to be branch. Proceed normally
- Yes: then instruction is branch and predicted PC should be used as the next PC
- Branch predicted taken or not taken
Handling Branch Target Cache

- No branch delay if the branch prediction entry is found and is correct
- A penalty of two cycles is imposed for a wrong prediction or a cache miss
- Cache update on misprediction and misses can extend the time penalty
- Dealing with misses or misprediction is expensive and should be optimized
Branch target caching can be applied to expedite unconditional jumps (branch folding) and returns for procedure calls.

For calls from multiple sites, not clustered in time, a stack implementation of the branch target cache can be useful.
Basic Branch Prediction

- Simplest dynamic branch-prediction scheme
  - use a branch history table to track when the branch was taken and not taken
  - Branch history table is a small 1-bit buffer indexed by lower bits of PC address with the bit is set to reflect the whether or not branch taken last time
- Performance = $f(\text{accuracy, cost of misprediction})$
- Problem: in a loop, 1-bit branch history table will cause two mispredictions:
  - End of loop case, when it exits instead of looping
  - First time through loop on next time through code, when it predicts exit instead of looping
2-bit Branch History Table

- A two-bit buffer better captures the history of the branch instruction
- A prediction must miss twice to change
**N-bit Predictors**

- 2-bit is a special case of n-bit counter
  - For every entry in the prediction buffer
  - Increment/decrement if branch taken/not
  - If the counter value is one half of the maximum value \((2^n - 1)\), predict taken

- Slow to change prediction, but can change
- Prediction accuracy of a 4096-entry prediction buffer ranges from 82% to 99% for the SPEC89 benchmarks.

- The performance impact depends on frequency of branching instructions and the penalty of misprediction.
SPEC89 benchmarks

- 4096 entries (2 bits/entry)
- Unlimited entries (2 bits/entry)

- Buffer size has little impact beyond a certain size
- Misprediction is because either:
  - Wrong guess for that branch
  - Got branch history of wrong branch when index the table

Optimal Size for 2-bit Branch Buffers
Hypothesis: recent branches are correlated; that is, behavior of recently executed branches affects prediction of current branch
(2,2) Correlating Predictors

• Record m most recently executed branches as taken or not taken, and use that pattern to select the proper branch history table

• (m,n) predictor means record last m branches to select between 2m history tables each with n-bit counters
  – Old 2-bit branch history table is a (0,2) predictor

• In a (2,2) predictor, the behavior of recent branches selects between, four predictions of next branch, updating just that prediction

Total size = $2^m \times n \times \#$ prediction entries selected by branch address
Accuracy of Different Schemes

- 4096 entries (2 bits/entry)
- Unlimited entries (2 bits/entry)
- 1024 entries (2,2)
Example

• Assume that d has values 0, 1, or 2 (alternating between 0, 2)
• Assume that the sequence will be executed repeatedly
• Ignore all other branches including those causing the sequence to repeat
• All branches are initially predicted to untaken state

if (d==0)  
d=1;
if (d==1)  
L1:
BNEZ R1, L1 ; branch b1 (d!=0)
DADDI R1, R0, #1 ; d==0, sp d=1
DSUBUI R3, R1, #1
BNEZ R3, L2 ; branch b2 (d!=1)
....
L2:
**Example**

**With a single bit predictor**

<table>
<thead>
<tr>
<th>d=?</th>
<th>b1 prediction</th>
<th>b1 action</th>
<th>New b1 prediction</th>
<th>b2 prediction</th>
<th>b2 action</th>
<th>New b2 prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>0</td>
<td>T</td>
<td>NT</td>
<td>NT</td>
<td>T</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>2</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>0</td>
<td>T</td>
<td>NT</td>
<td>NT</td>
<td>T</td>
<td>NT</td>
<td>NT</td>
</tr>
</tbody>
</table>

- All branches are mispredicted

```c
if (d==0)
    d=1;
else if (d==1)
```
Example

With one bit predictor with one bit of correlation

<table>
<thead>
<tr>
<th>d=?</th>
<th>b1 prediction</th>
<th>b1 action</th>
<th>New b1 prediction</th>
<th>b2 prediction</th>
<th>b2 action</th>
<th>New b2 prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>NT/NT</td>
<td>T</td>
<td>T/NT</td>
<td>NT/NT</td>
<td>T</td>
<td>NT/T</td>
</tr>
<tr>
<td>0</td>
<td>T/NT</td>
<td>NT</td>
<td>T/NT</td>
<td>NT/T</td>
<td>NT</td>
<td>NT/T</td>
</tr>
<tr>
<td>2</td>
<td>T/NT</td>
<td>T</td>
<td>T/NT</td>
<td>NT/T</td>
<td>T</td>
<td>NT/T</td>
</tr>
<tr>
<td>0</td>
<td>T/NT</td>
<td>NT</td>
<td>T/NT</td>
<td>NT/T</td>
<td>NT</td>
<td>NT/T</td>
</tr>
</tbody>
</table>

- Except for first iteration, all branches are correctly predicted

if (d==0) d=1;
if (d==1)

if (d==0) BNEZ R1, L1 ; branch b1 (d!=0)
  DADDI R1, R0, #1 ; d==0, sp d=1
L1: DSUBUI R3, R1, #1 ; branch b2 (d!=1)
  BNEZ R3, L2

....
L2:
Tournament Predictors

- Multilevel branch predictors use several levels of branch prediction tables together with an algorithm to choose among them.
- Tournament selectors are the most popular form of multilevel branch predictors (e.g. DEC Alpha 21264).
- Tournament predictors combines both local and global predictor.
- Selection between the two predictors are based on a selector (2-bit counter).
- Make a transition with two wrong prediction using the current table for which the correct prediction would have been possible using the other predictor.
Performance of Tournament Predictors

Based on SPEC 89 benchmark

Tournament predictors slightly outperform correlating predictors