Improving Processor Efficiency by Statically Pipelining Instructions

Brandon Davis, Ian Finlayson, Peter Gavin, Magnus Själander, Gang-Ryung Uh*, David Whalley, Gary Tyson

Computer Science Department
Florida State University

*Computer Science Department
Boise State University

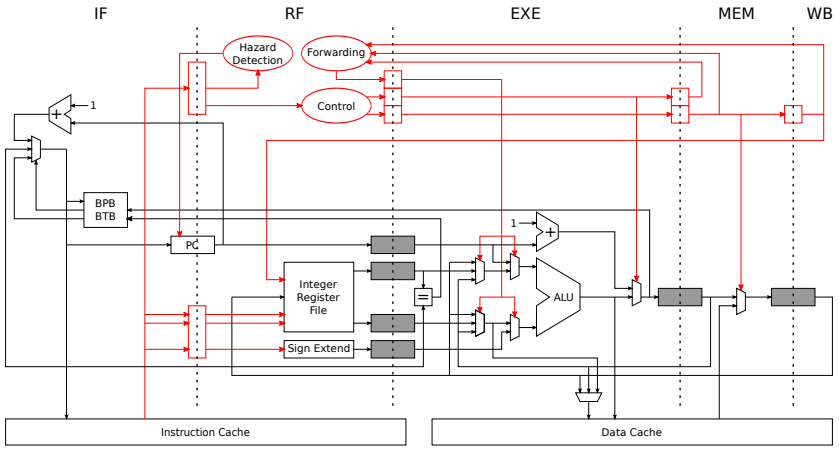
March 8, 2024

Energy Efficient Processor Design

- Need for energy efficient processors.
  - Extend battery life for mobile embedded systems.
  - Reduce generated heat for general-purpose processors.
  - Energy cost for computing is increasing.
- Prefer not to negatively impact performance.

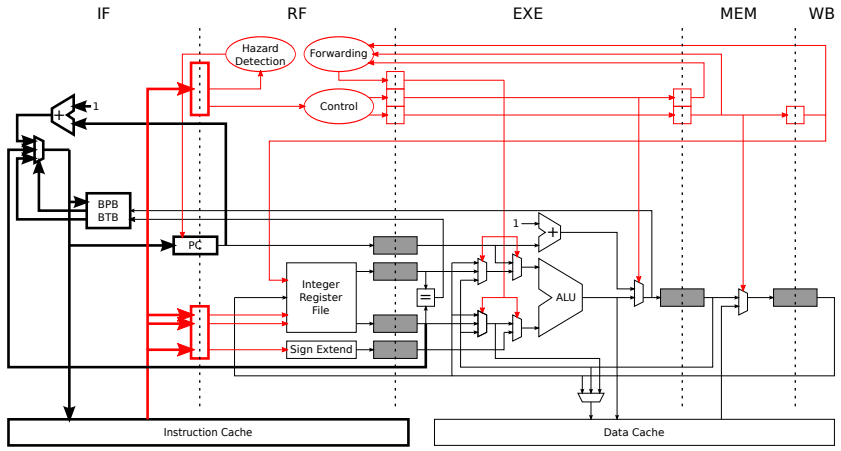
Traditional Instruction Pipelining

- A classical pipeline consists of five stages.



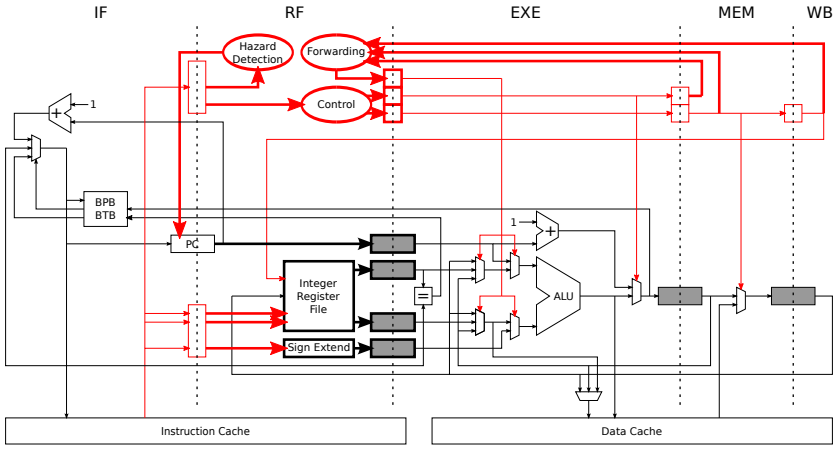
Traditional Instruction Pipelining

- Instruction Fetch (IF) Stage



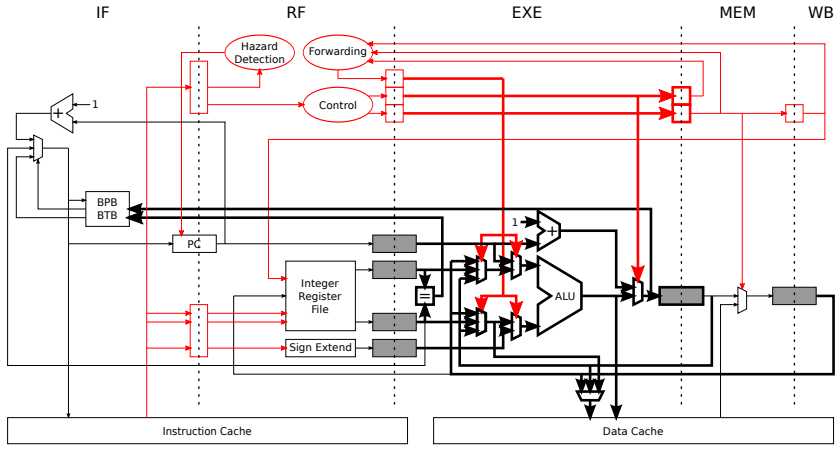
Traditional Instruction Pipelining

- Register Fetch (RF) Stage



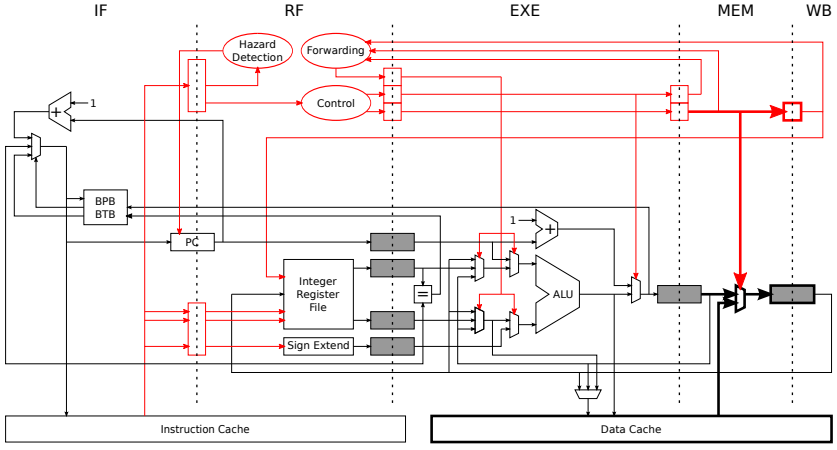
Traditional Instruction Pipelining

- EXecution (EX) Stage



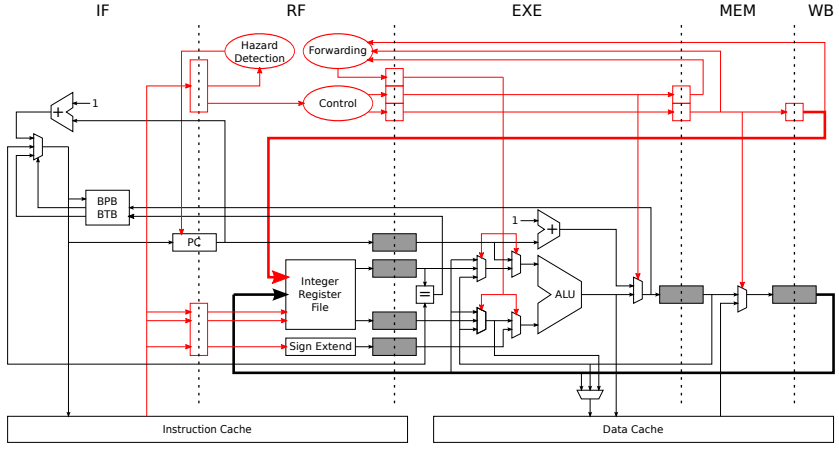
Traditional Instruction Pipelining

- MEMory (MEM) Stage



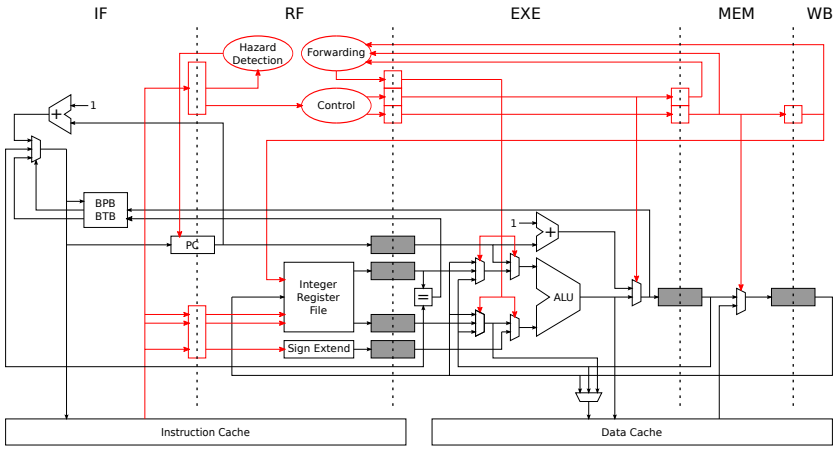
Traditional Instruction Pipelining

- Write Back (WB) Stage



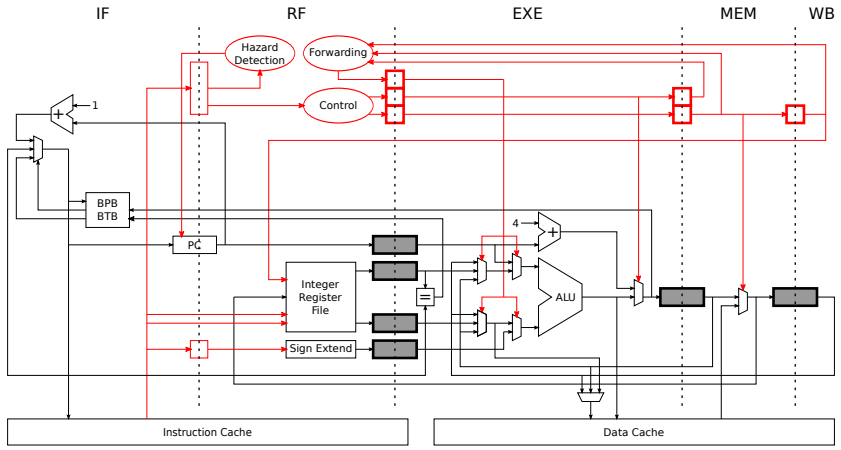
Traditional Instruction Pipelining

- Several inefficiencies with instruction pipelining.



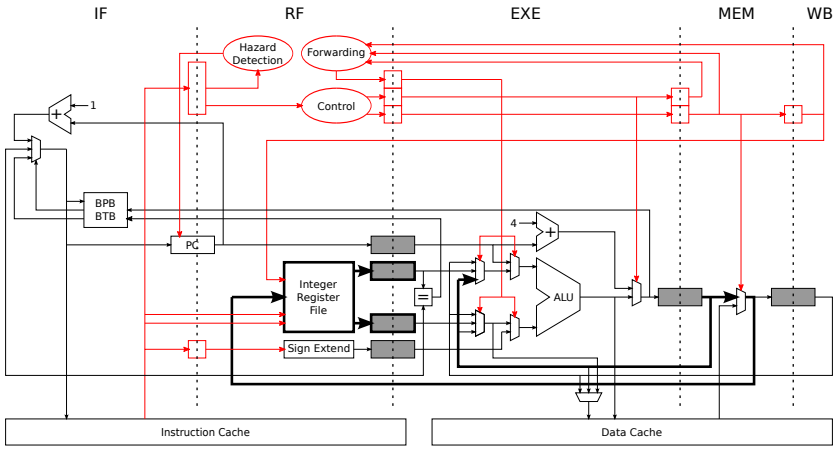
Traditional Instruction Pipelining

- Pipeline registers are getting updated every cycle.



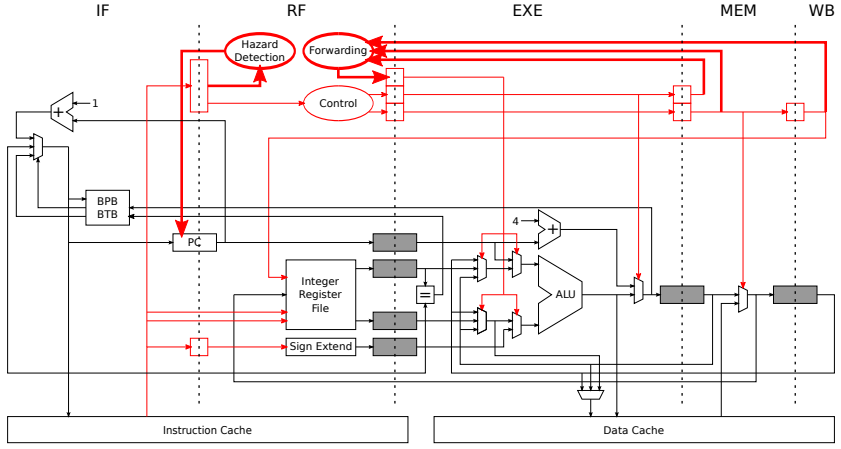
Traditional Instruction Pipelining

- Unnecessary register file accesses due to forwarding.



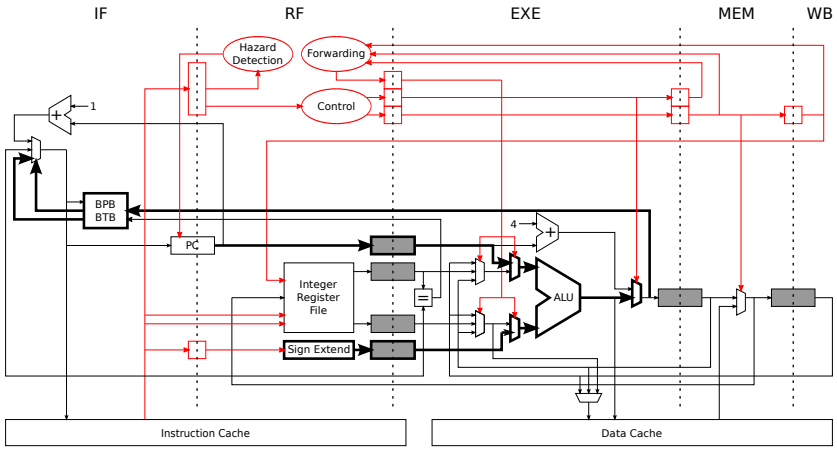
Traditional Instruction Pipelining

- Constantly checking for hazards and forwarding.

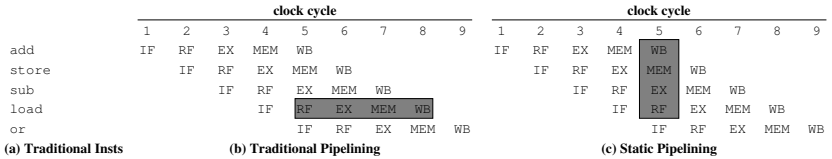


Traditional Instruction Pipelining

- Unnecessary branch target addr calcs and BPB/BTB accesses.

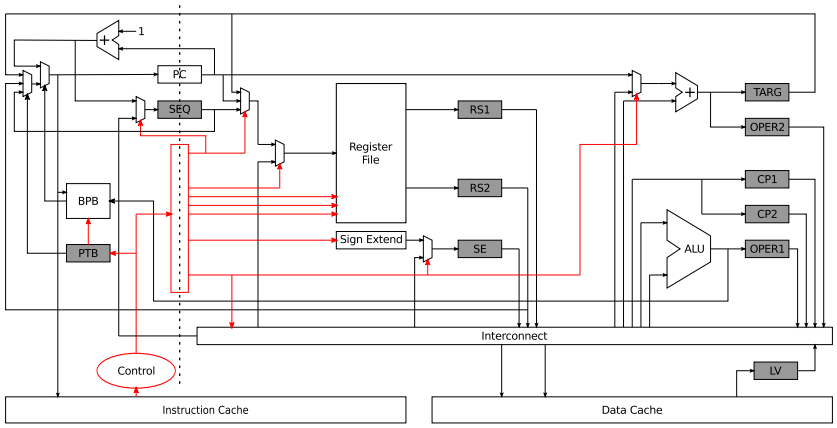


Traditional vs Static Pipelining



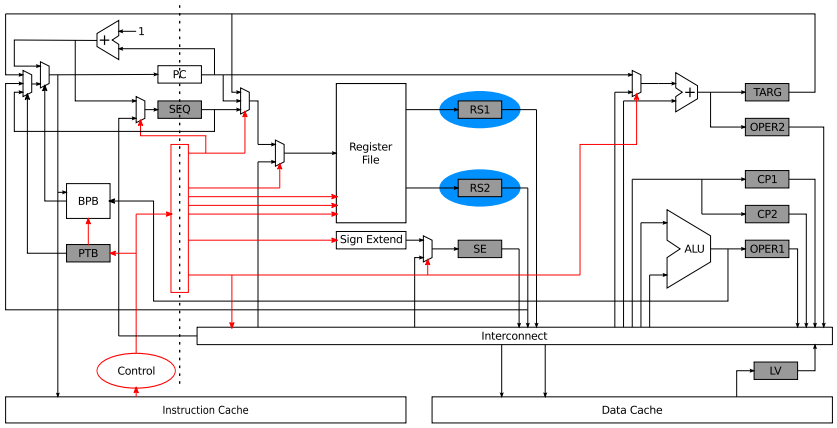
Static Pipeline

- Internal registers can be explicitly accessed in SP instructions.



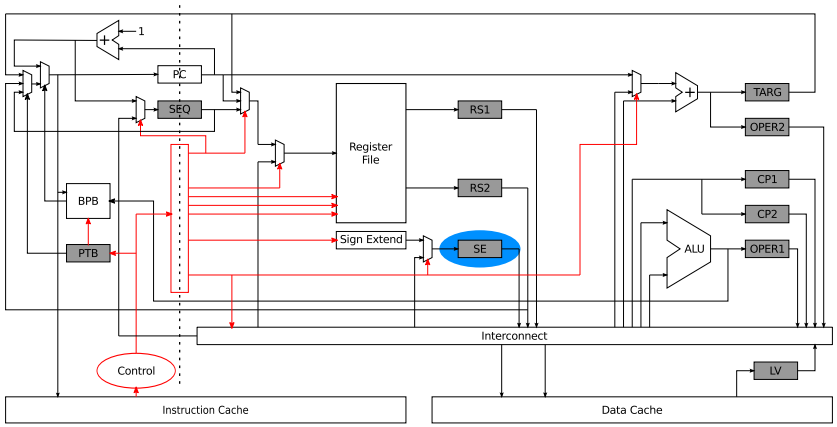
Static Pipeline

- A register file register is loaded into an RS register.



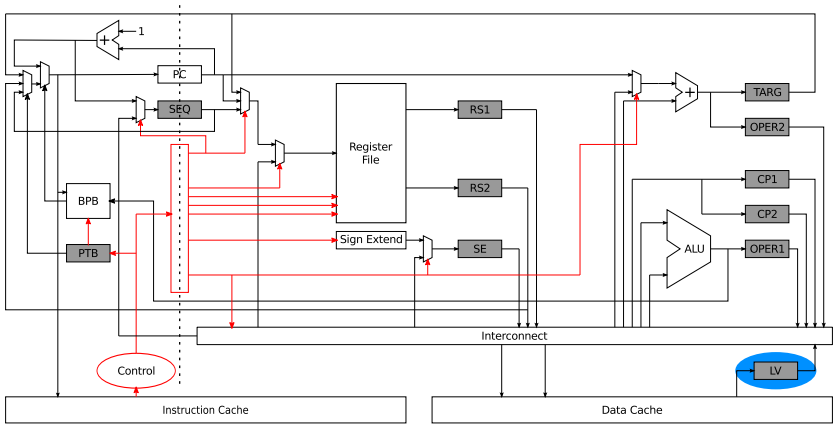
Static Pipeline

- The SE register contains a sign-extended value.



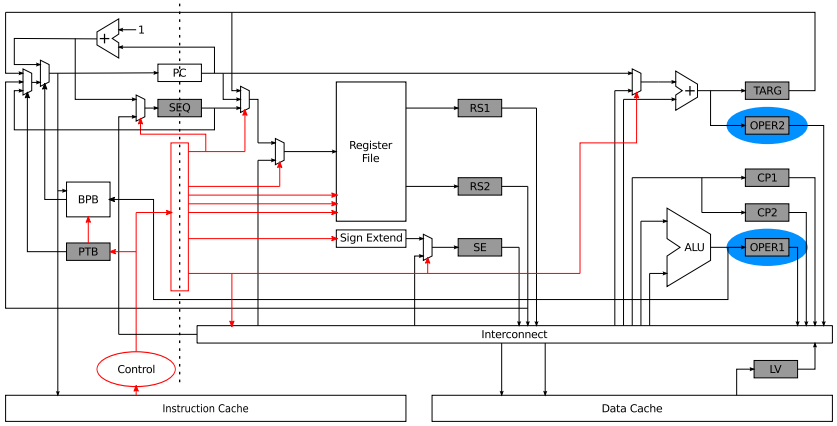
Static Pipeline

- A value from the L1 DC is loaded into the LV register.



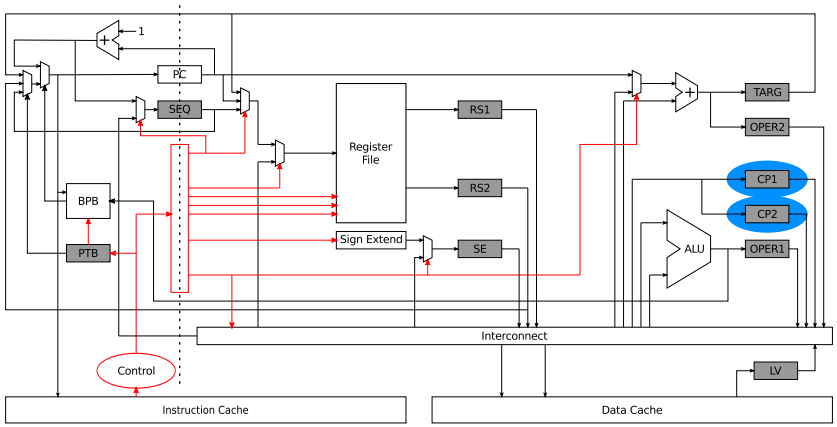
Static Pipeline

- OPER1 contains the result of an ALU operation and OPER2 contains the result of an integer addition.



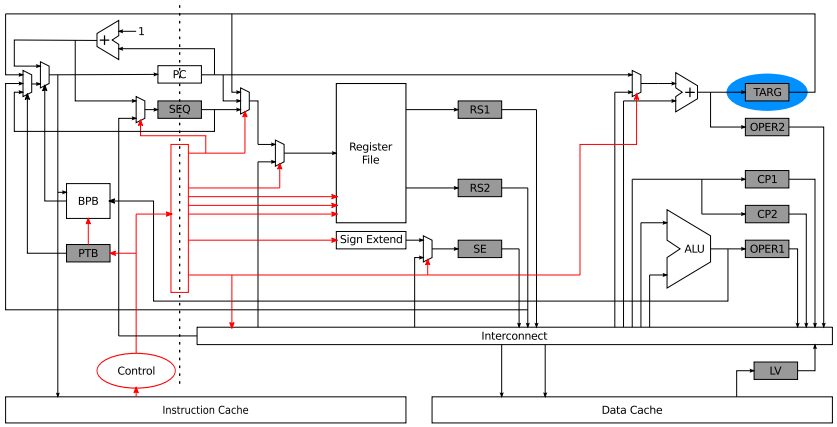
Static Pipeline

- Internal SP registers can be copied to a CP register.



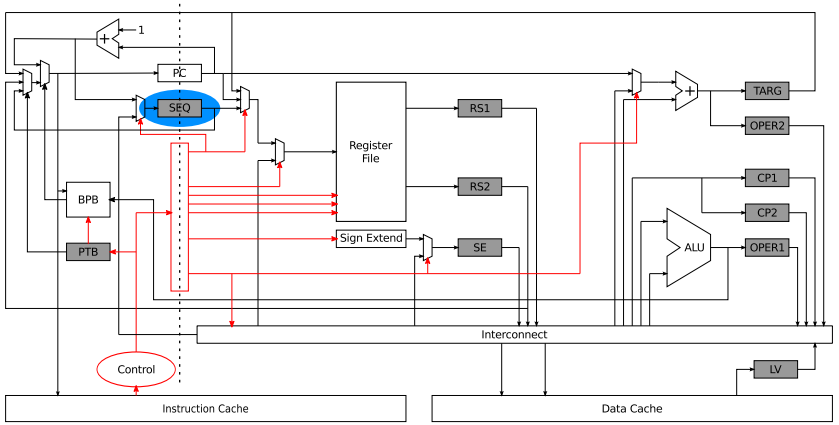
Static Pipeline

- The TARG register holds a branch target address.



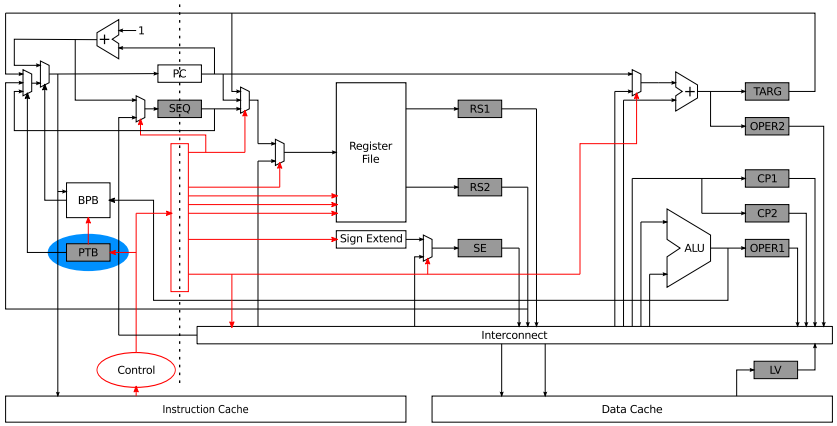
Static Pipeline

- The SEQ register is used to hold the PC incremented address.

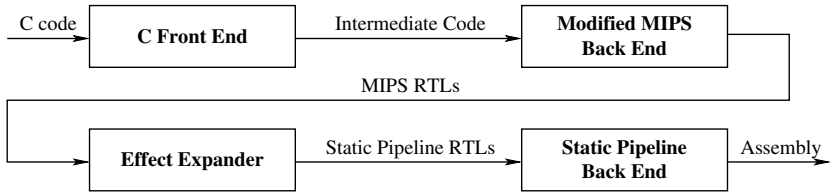


Static Pipeline

- The PTB indicates when the next instruction is a branch.



Overview of Compilation Process



Example

Source

```
for (i = 0; i < 100; i++)
   a[i] += m;
```

MIPS

```
   r6=<value m>;
   r9=<address a>;
L2:
   r3 = M[r9];
   r2 = r3 + r6;
   M[r9] = r2;
   r9 = r9 + 4;
   PC = r9 != r5, L2;
```


Initial Static Pipelined Code

```
   r6 = LV;       # LV == value m
   r9 = OPER2;    # OPER2 == address a
L2:
   # r3 = M[r9];
   # r2 = r3 + r6;
   # M[r9] = r2;
   # r9 = r9 + 4;
   # PC = r9 != r5, L2;
```


Initial Static Pipelined Code

```
   r6 = LV;
   r9 = OPER2;
L2:
   # r3 = M[r9];
   RS1 = r9;
   LV = M[RS1];
   r3 = LV;
   # r2 = r3 + r6;
   # M[r9] = r2;
   # r9 = r9 + 4;
   # PC = r9 != r5, L2;
```


Initial Static Pipelined Code

```
   r6 = LV;
   r9 = OPER2;
L2:
   # r3 = M[r9];
   RS1 = r9;
   LV = M[RS1];
   r3 = LV;
   # r2 = r3 + r6;
   RS1 = r3;
   RS2 = r6;
   OPER2 = RS1 + RS2;
   r2 = OPER2;
   # M[r9] = r2;
   # r9 = r9 + 4;
   # PC = r9 != r5, L2;
```

Initial Static Pipelined Code

```
  r6 = LV;
  r9 = OPER2;
L2:
  # r3 = M[r9];                  # r9 = r9 + 4;
  RS1 = r9;                      SE = 4;
  LV = M[RS1];                   RS1 = r9;
  r3 = LV;                       OPER2 = RS1 + SE;
  # r2 = r3 + r6;                r9 = OPER2;
  RS1 = r3;                      # PC = r9 != r5, L2;
  RS2 = r6;                      SE = offset(L2);
  OPER2 = RS1 + RS2;             TARG = PC + SE;
  r2 = OPER2;                    RS1 = r9;
  # M[r9] = r2;                  RS2 = r5;
  RS1 = r9;                      PC = RS1 != RS2, TARG;
  RS2 = r2;
  M[RS1] = RS2;
```


Copy Propagation

```
  r6 = LV;
  r9 = OPER2;
L2:
  RS1 = r9;                      SE = 4;
  LV = M[RS1];                   RS1 = r9;
  r3 = LV;                       OPER2 = RS1 + SE;
  RS1 = r3;                      r9 = OPER2;
  RS2 = r6;                      SE = offset(L2);
  OPER2 = RS1 + RS2;             TARG = PC + SE;
  r2 = OPER2;                    RS1 = r9;
  RS1 = r9;                      RS2 = r5;
  RS2 = r2;                      PC = RS1 != RS2, TARG;
  M[RS1] = RS2;
```


Copy Propagation

```
  r6 = LV;
  r9 = OPER2;
L2:
  RS1 = r9;                      SE = 4;
  LV = M[RS1];                   RS1 = r9;
  r3 = LV;                       OPER2 = RS1 + SE;
  RS1 = r3;                      r9 = OPER2;
  RS2 = r6;                      SE = offset(L2);
  OPER2 = LV + RS2;              TARG = PC + SE;
  r2 = OPER2;                    RS1 = r9;
  RS1 = r9;                      RS2 = r5;
  RS2 = r2;                      PC = OPER2 != RS2, TARG;
  M[RS1] = OPER2;
```


Dead Assignment Elimination

```
  r6 = LV;
  r9 = OPER2;
L2:
  RS1 = r9;                      SE = 4;
  LV = M[RS1];                   RS1 = r9;
  r3 = LV;                       OPER2 = RS1 + SE;
  RS1 = r3;                      r9 = OPER2;
  RS2 = r6;                      SE = offset(L2);
  OPER2 = LV + RS2;              TARG = PC + SE;
  r2 = OPER2;                    RS1 = r9;
  RS1 = r9;                      RS2 = r5;
  RS2 = r2;                      PC = OPER2 != RS2, TARG;
  M[RS1] = OPER2;
```

Dead Assignment Elimination

```
    r6 = LV;
    r9 = OPER2;
L2:
    RS1 = r9;                    SE = 4;
    LV = M[RS1];                 RS1 = r9;
    r3 = LV;                     OPER2 = RS1 + SE;
    RS1 = r3;                    r9 = OPER2;
    RS2 = r6;                    SE = offset(L2);
    OPER2 = LV + RS2;            TARG = PC + SE;
    r2 = OPER2;                  RS1 = r9;
    RS1 = r9;                    RS2 = r5;
    RS2 = r2;                    PC = OPER2 != RS2, TARG;
    M[RS1] = OPER2;
```

Motivation Compilation Results Summary

Dead Assignment Elimination

```
    r6 = LV;
    r9 = OPER2;
L2:
    RS1 = r9;                    SE = 4;
    LV = M[RS1];                 RS1 = r9;
    r3 = LV;                     OPER2 = RS1 + SE;
    RS1 = r3;                    r9 = OPER2;
    RS2 = r6;                    SE = offset(L2);
    OPER2 = LV + RS2;            TARG = PC + SE;
    r2 = OPER2;                  RS1 = r9;
    RS1 = r9;                    RS2 = r5;
    RS2 = r2;                    PC = OPER2 != RS2, TARG;
    M[RS1] = OPER2;
```

Motivation Compilation Results Summary

Redundant Assignment Elimination

```
    r6 = LV;
    r9 = OPER2;
L2:
    RS1 = r9;
    LV = M[RS1];
    RS2 = r6;
    OPER2 = LV + RS2;
    RS1 = r9;
    M[RS1] = OPER2;
    SE = 4;
    RS1 = r9;
    OPER2 = RS1 + SE;
    r9 = OPER2;
    SE = offset(L2);
    TARG = PC + SE;
    RS2 = r5;
    PC = OPER2 != RS2, TARG;
```

Motivation Compilation Results Summary

Redundant Assignment Elimination

```
    r6 = LV;
    r9 = OPER2;
L2:
    RS1 = r9;
    LV = M[RS1];
    RS2 = r6;
    OPER2 = LV + RS2;
    RS1 = r9;
    M[RS1] = OPER2;
    SE = 4;
    RS1 = r9;
    OPER2 = RS1 + SE;
    r9 = OPER2;
    SE = offset(L2);
    TARG = PC + SE;
    RS2 = r5;
    PC = OPER2 != RS2, TARG;
```

Redundant Assignment Elimination

```
    r6 = LV;
    r9 = OPER2;
  L2:
    RS1 = r9;
    LV = M[RS1];
    RS2 = r6;
    OPER2 = LV + RS2;
    M[RS1] = OPER2;
    SE = 4;
    OPER2 = RS1 + SE;
    r9 = OPER2;
    SE = offset(L2);
    TARG = PC + SE;
    RS2 = r5;
    PC = OPER2 != RS2, TARG;
```


Loop Invariant Code Motion

```
    r6 = LV;
    r9 = OPER2;
  L2:
    RS1 = r9;
    LV = M[RS1];
    RS2 = r6;
    OPER2 = LV + RS2;
    M[RS1] = OPER2;
    SE = 4;
    OPER2 = RS1 + SE;
    r9 = OPER2;
    SE = offset(L2);
    TARG = PC + SE;
    RS2 = r5;
    PC = OPER2 != RS2, TARG;
```


Loop Invariant Code Motion

```
    r6 = LV;                      r6 = LV;
    r9 = OPER2;                   r9 = OPER2;
  L2:                             SE = offset(L2);
    RS1 = r9;                     TARG = PC + SE;
    LV = M[RS1];                L2:
    RS2 = r6;                     RS1 = r9;
    OPER2 = LV + RS2;             LV = M[RS1];
    M[RS1] = OPER2;               RS2 = r6;
    SE = 4;                       OPER2 = LV + RS2;
    OPER2 = RS1 + SE;             M[RS1] = OPER2;
    r9 = OPER2;                   SE = 4;
    SE = offset(L2);              OPER2 = RS1 + SE;
    TARG = PC + SE;               r9 = OPER2;
    RS2 = r5;                     RS2 = r5;
    PC = OPER2 != RS2, TARG;      PC = OPER2 != RS2, TARG;
```


Loop Invariant Code Motion

```
    r6 = LV;
    r9 = OPER2;
    SE = offset(L2);
    TARG = PC + SE;
  L2:
    RS1 = r9;
    LV = M[RS1];
    RS2 = r6;
    OPER2 = LV + CP2;
    M[RS1] = OPER2;
    SE = 4;
    OPER2 = RS1 + SE;
    r9 = OPER2;
    RS2 = r5;
    PC = OPER2 != RS2, TARG;
```

Loop Invariant Code Motion

```
    r6 = LV;                      r6 = LV;
    r9 = OPER2;                   r9 = OPER2;
    SE = offset(L2);              SE = offset(L2);
    TARG = PC + SE;               TARG = PC + SE;
L2:                               SE = 4;
    RS1 = r9;                 L2:
    LV = M[RS1];                  RS1 = r9;
    RS2 = r6;                     LV = M[RS1];
    OPER2 = LV + CP2;             RS2 = r6;
    M[RS1] = OPER2;               OPER2 = LV + CP2;
    SE = 4;                       M[RS1] = OPER2;
    OPER2 = RS1 + SE;             OPER2 = RS1 + SE;
    r9 = OPER2;                   r9 = OPER2;
    RS2 = r5;                     RS2 = r5;
    PC = OPER2 != RS2, TARG;      PC = OPER2 != RS2, TARG;
```


CP Register Allocation

```
    r6 = LV;
    r9 = OPER2;
    SE = offset(L2);
    TARG = PC + SE;
    SE = 4;
L2:
    RS1 = r9;
    LV = M[RS1];
    RS2 = r6;
    OPER2 = LV + RS2;
    M[RS1] = OPER2;
    OPER2 = RS1 + SE;
    r9 = OPER2;
    RS2 = r5;
    PC = OPER2 != RS2, TARG;
```


CP Register Allocation

```
    r6 = LV;                      CP1 = LV;
    r9 = OPER2;                   CP2 = OPER2;
    SE = offset(L2);              SE = offset(L2);
    TARG = PC + SE;               TARG = PC + SE;
    SE = 4;                       SE = 4;
L2:                           L2:
    RS1 = r9;                     RS1 = r9;
    LV = M[RS1];                  LV = M[CP2];
    RS2 = r6;                     RS2 = r6;
    OPER2 = LV + RS2;             OPER2 = LV + CP1;
    M[RS1] = OPER2;               M[CP2] = OPER2;
    OPER2 = RS1 + SE;             OPER2 = CP2 + SE;
    r9 = OPER2;                   CP2 = OPER2;
    RS2 = r5;                     RS2 = r5;
    PC = OPER2 != RS2, TARG;      PC = OPER2 != RS2, TARG;
```


CP Register Allocation

```
    r6 = LV;                      CP1 = LV;
    r9 = OPER2;                   CP2 = OPER2;
    SE = offset(L2);              SE = offset(L2);
    TARG = PC + SE;               TARG = PC + SE;
    SE = 4;                       SE = 4;
L2:                           L2:
    RS1 = r9;                     LV = M[CP2];
    LV = M[RS1];                  OPER2 = LV + CP1;
    RS2 = r6;                     M[CP2] = OPER2;
    OPER2 = LV + RS2;             OPER2 = CP2 + SE;
    M[RS1] = OPER2;               CP2 = OPER2;
    OPER2 = RS1 + SE;             RS2 = r5;
    r9 = OPER2;                   PC = OPER2 != RS2, TARG;
    RS2 = r5;
    PC = OPER2 != RS2, TARG;
```

Loop Invariant Code Motion

```
    CP1 = LV;
    CP2 = OPER2;
    SE = offset(L2);
    TARG = PC + SE;
    SE = 4;
L2:
    LV = M[CP2];
    OPER2 = LV + CP1;
    M[CP2] = OPER2;
    OPER2 = CP2 + SE;
    CP2 = OPER2;
    RS2 = r5;
    PC = OPER2 != RS2, TARG;
```


Loop Invariant Code Motion

```
    CP1 = LV;
    CP2 = OPER2;
    SE = offset(L2);
    TARG = PC + SE;
    SE = 4;
L2:
    LV = M[CP2];
    OPER2 = LV + CP1;
    M[CP2] = OPER2;
    OPER2 = CP2 + SE;
    CP2 = OPER2;
    RS2 = r5;
    PC = OPER2 != RS2, TARG;
```

```
    CP1 = LV;
    CP2 = OPER2;
    SE = offset(L2);
    TARG = PC + SE;
    SE = 4;
    RS2 = r5;
L2:
    LV = M[CP2];
    OPER2 = LV + CP1;
    M[CP2] = OPER2;
    OPER2 = CP2 + SE;
    CP2 = OPER2;
    PC = OPER2 != RS2, TARG;
```


Using Sequential Address Register

```
    CP1 = LV;
    CP2 = OPER2;
    SE = offset(L2);
    TARG = PC + SE;
    SE = 4;
    RS2 = r5;
L2:
    LV = M[CP2];
    OPER2 = LV + CP1;
    M[CP2] = OPER2;
    OPER2 = CP2 + SE;
    CP2 = OPER2;
    PC = OPER2 != RS2, TARG;
```


Using Sequential Address Register

```
    CP1 = LV;
    CP2 = OPER2;
    SE = offset(L2);
    TARG = PC + SE;
    SE = 4;
    RS2 = r5;
L2:
    LV = M[CP2];
    OPER2 = LV + CP1;
    M[CP2] = OPER2;
    OPER2 = CP2 + SE;
    CP2 = OPER2;
    PC = OPER2 != RS2, TARG;
```

```
    CP1 = LV;
    CP2 = OPER2;
    SE = offset(L2);
    TARG = PC + SE;
    SE = 4;
    RS2 = r5;
    SEQ = PC + 1;
L2:
    LV = M[CP2];
    OPER2 = LV + CP1;
    M[CP2] = OPER2;
    OPER2 = CP2 + SE;
    CP2 = OPER2;
    PC = OPER2 != RS2, SEQ;
```

Using Sequential Address Register

```
    CP1 = LV;
    CP2 = OPER2;
    SE = offset(L2);
    TARG = PC + SE;
    SE = 4;
    RS2 = r5;
  L2:
    LV = M[CP2];
    OPER2 = LV + CP1;
    M[CP2] = OPER2;
    OPER2 = CP2 + SE;
    CP2 = OPER2;
    PC = OPER2 != RS2, TARG;
```

```
    CP1 = LV;
    CP2 = OPER2;
    SE = 4;
    RS2 = r5;
    SEQ = PC + 1;
  L2:
    LV = M[CP2];
    OPER2 = LV + CP1;
    M[CP2] = OPER2;
    OPER2 = CP2 + SE;
    CP2 = OPER2;
    PC = OPER2 != RS2, SEQ;
```


Static Pipeline Instruction Formats

- A template ID is used to specify how the remaining fields are interpreted.

5-bit ID	10-bit Effect	10-bit Effect	7-bit Effect
5-bit ID	3-bit PTB	7-bit Effect	10-bit Effect	7-bit Effect
5-bit ID	10-bit Effect	Long Immediate	
5-bit ID	3-bit PTB	7-bit Effect	Long Immediate

10-bit Effects

ALU Operation
FPU Operation
Load or Store Operation
Dual Register Reads (RS1 and RS2)
Register Write

7-bit Effects

Integer Addition
Load Signed Word
Single Register Read (RS1 or RS2)
Short Immediate
Copy Operation
Prepare to Branch (PTB)

Code after Renaming and Scheduling Effects

```
  CP1=LV;          SE=4;            RS2=r5;
  CP2=OPER2;
L2:
  LV=M[CP2];       OPER2=CP2+SE;
  OPER1=LV+CP1;
  M[CP2]=OPER1;    CP2=OPER2;       PC=OPER2!=RS2,SEQ;
```


Handling Transfers of Control

- three components of a transfer of control
  - calculate the target address
  - decide whether or not to branch
  - indicate point of transfer
- We have already decoupled the target address calculation.
- We now specify information about the transfer of control in the instruction preceding the point of transfer.

Handling Transfers of Control

- three components of a transfer of control
  - calculate the target address
  - decide whether or not to branch
  - indicate point of transfer
- We have already decoupled the target address calculation.
- We now specify information about the transfer of control in the instruction preceding the point of transfer.

```
SE = offset(L6);
TARG = PC + SE;
...
PC = OPER2 != RS2, TARG;
```

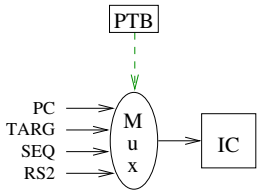

Handling Transfers of Control

- three components of a transfer of control
  - calculate the target address
  - decide whether or not to branch
  - indicate point of transfer
- We have already decoupled the target address calculation.
- We now specify information about the transfer of control in the instruction preceding the point of transfer.

```
SE = offset(L6);
TARG = PC + SE;
...
PC = OPER2 != RS2, TARG;
```

```
SE = offset(L6);
TARG = PC + SE;
...
PTB = b:&TARG;
PC = OPER2 != RS2, @PTB;
```


Specifying Transfers of Control

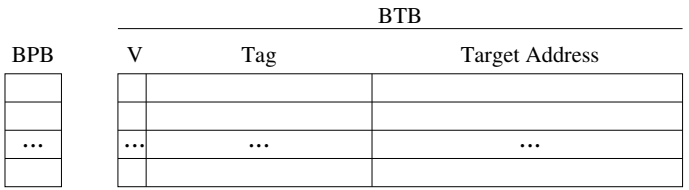


PTB = &TARG;
PC = ALUR != RS2, @PTB;
target instruction

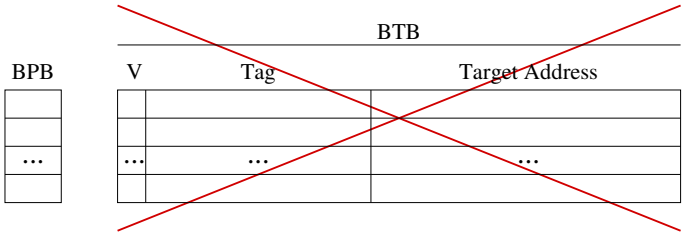
1	2	3	4
IF	EX		
	IF	EX	
		IF	EX

- PTB indicates which address should be used when accessing the instruction cache (IC).
- In cycle 2 the signal assigned to PTB indicates the TARG register.
- In cycle 3 the address in TARG is used to access the IC.
- Branch predictor is accessed in cycle 2 and can override the PTB assignment when the branch is predicted to be not taken.

Efficient Transfers of Control



Efficient Transfers of Control



- By calculating the target address earlier and specifying the point of transfer in the preceding instruction:
  - No longer need a BTB.
  - Only need to access the BPB on conditional branches.

PTB and SEQ Placement

```
   CP1=LV;          SE=4;            RS2=r5;
   CP2=OPER2;
L2:
   LV=M[CP2];       OPER2=CP2+SE;
   OPER1=LV+CP1;
   M[CP2]=OPER1;    CP2=OPER2;       PC=OPER2!=RS2,SEQ;
```


PTB and SEQ Placement

```
   CP1=LV;          SE=4;            RS2=r5;
   CP2=OPER2;
L2:
   LV=M[CP2];       OPER2=CP2+SE;
   OPER1=LV+CP1;
   M[CP2]=OPER1;    CP2=OPER2;       PC=OPER2!=RS2,SEQ;
=>
   CP1=LV;          SE=4;            RS2=r5;
   CP2=OPER2;       SEQ=PC+1;
L2:
   LV=M[CP2];       OPER2=CP2+SE;
   OPER1=LV+CP1;    PTB=b:&SEQ;
   M[CP2]=OPER1;    CP2=OPER2;       PC=OPER2!=RS2,@PTB;
```


Other Optimizations

- Loop invariant code motion of register file accesses using CP registers.
- Transforming large constants to small constants.
- Scheduling effects across basic blocks.

Avoiding Use of Large Constants

- Large constants require 17 bits in our encoding.
- More effects can be performed in parallel if a 7-bit constant can instead be used.
- Assume *const2* cannot be represented in 7 bits, but *const2-const1* can be represented in 7 bits.

```
SE=const1;
OPER2=RS1+SE;
...
SE=const2;
OPER2=RS1+SE;
```


Avoiding Use of Large Constants

- Large constants require 17 bits in our encoding.
- More effects can be performed in parallel if a 7-bit constant can instead be used.
- Assume *const2* cannot be represented in 7 bits, but *const2-const1* can be represented in 7 bits.

```
SE=const1;                  SE=const1;
OPER2=RS1+SE;               OPER2=RS1+SE;
...                         ...
SE=const2;                  SE=const2-const1;
OPER2=RS1+SE;               OPER2=OPER2+SE;
```


Avoiding Use of Large Constants Example

```
    SE=188;     RS1=r29;          RS2=r23;
    SE=184;     OPER2=RS1+SE;
                OPER2=RS1+SE;     M[OPER2]=SEQ;
    SE=180;                       M[OPER2]=CP1;
    SE=176;     OPER2=RS1+SE;
                OPER2=RS1+SE;     M[OPER2]=CP2;
    SE=172;                       M[OPER2]=RS2;
    SE=168;     OPER2=RS1+SE;
    RS2=r24;
    RS2=r25;    OPER2=RS1+SE;     M[OPER2]=RS2;
                                  M[OPER2]=RS2;
=>
```


Avoiding Use of Large Constants Example

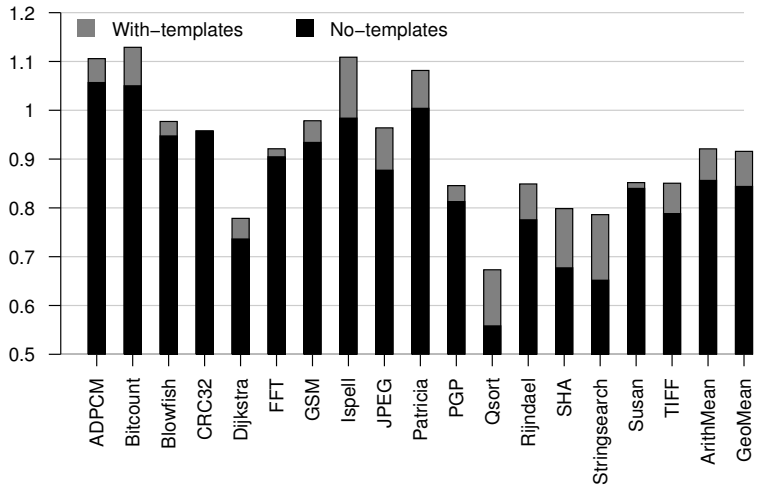
```
    SE=188;     RS1=r29;          RS2=r23;
    SE=184;     OPER2=RS1+SE;
                OPER2=RS1+SE;     M[OPER2]=SEQ;
    SE=180;                       M[OPER2]=CP1;
    SE=176;     OPER2=RS1+SE;
                OPER2=RS1+SE;     M[OPER2]=CP2;
    SE=172;                       M[OPER2]=RS2;
    SE=168;     OPER2=RS1+SE;
    RS2=r24;
    RS2=r25;    OPER2=RS1+SE;     M[OPER2]=RS2;
                                  M[OPER2]=RS2;
=>
    SE=188;     RS1=r29;          RS2=r23;
    SE=-4;      OPER2=RS1+SE;
                OPER2=OPER2+SE;   M[OPER2]=SEQ;
                OPER2=OPER2+SE;   M[OPER2]=CP1;
                OPER2=OPER2+SE;   M[OPER2]=CP2;
    RS2=r24;    OPER2=OPER2+SE;   M[OPER2]=RS2;
    RS2=r25;    OPER2=OPER2+SE;   M[OPER2]=RS2;
                                  M[OPER2]=RS2;
```

Experimental Evaluation

- Extended the GNU assembler to assemble SP instructions.
- Implemented a simulator based on SimpleScalar in-order MIPS.
- MIPS baseline is a VPO MIPS port with all optimizations enabled.
- Used 256 entry BPB, 256 entry BTB, 8KB L1 IC, 8K L1 DC.
- Compiled and simulated 17 MiBench benchmarks.

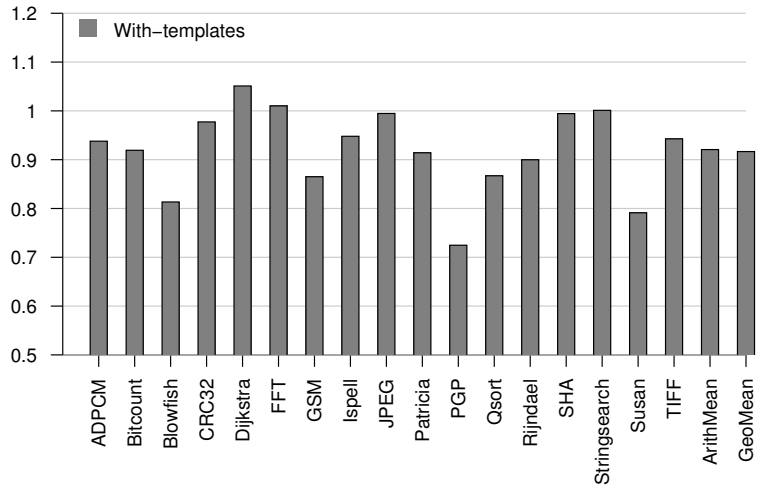
Performance

- On average an 8% reduction in cycles with 32-bit instructions and a 14% reduction using a long instruction format.



Code Size

- On average an 8% reduction in code size.

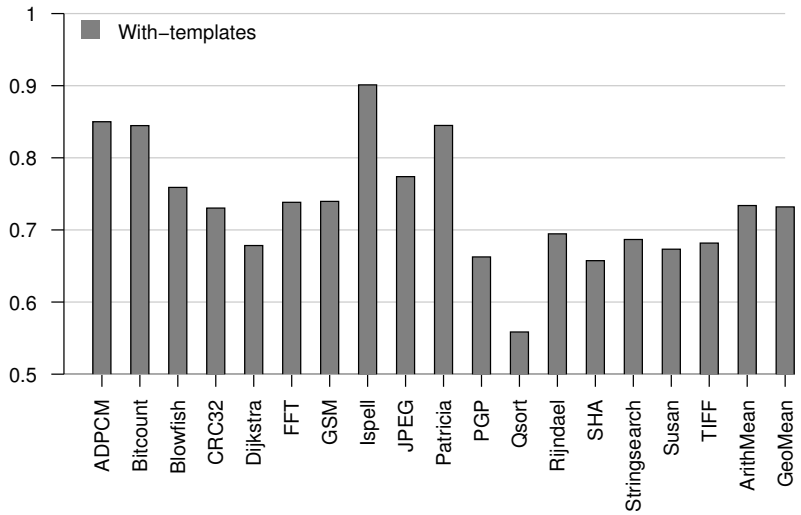


Processor Energy Estimation

- Used simulated counts of events along with estimates of power for each event.
- Component power normalized to a register file access.
  - 5.10 L1 Caches (8kB)
  - 0.65 branch prediction buffer
  - 2.86 branch target buffer
  - 1.00 register file access
  - 4.11 arithmetic logic unit
  - 12.60 floating point unit
  - 0.10 internal register writes

Energy

- On average a 27% reduction in energy.



Reasons for Energy Reductions

- Energy is reduced by decreasing the operations in a conventional pipeline.

Reasons for Energy Reductions

- Energy is reduced by decreasing the operations in a conventional pipeline.
  - 0.26 register file reads

Reasons for Energy Reductions

- Energy is reduced by decreasing the operations in a conventional pipeline.
  - 0.26 register file reads
  - 0.33 register file writes

Reasons for Energy Reductions

- Energy is reduced by decreasing the operations in a conventional pipeline.
  - 0.26 register file reads
  - 0.33 register file writes
  - 0.39 internal register writes

Motivation Compilation Results Summary

Reasons for Energy Reductions

- Energy is reduced by decreasing the operations in a conventional pipeline.
  - 0.26 register file reads
  - 0.33 register file writes
  - 0.39 internal register writes
  - 0.59 branch target address calculations

Motivation Compilation Results Summary

Reasons for Energy Reductions

- Energy is reduced by decreasing the operations in a conventional pipeline.
  - 0.26 register file reads
  - 0.33 register file writes
  - 0.39 internal register writes
  - 0.59 branch target address calculations
  - 0.13 BPB accesses

Motivation Compilation Results Summary

Reasons for Energy Reductions

- Energy is reduced by decreasing the operations in a conventional pipeline.
  - 0.26 register file reads
  - 0.33 register file writes
  - 0.39 internal register writes
  - 0.59 branch target address calculations
  - 0.13 BPB accesses
  - 0.00 BTB accesses

Related Work

- SP ISA is similar to horizontal microcode, but SP ISA is exposed to the compiler at the microarchitectural level.

Motivation Compilation Results Summary

Related Work

- SP ISA is similar to horizontal microcode, but SP ISA is exposed to the compiler at the microarchitectural level.
- several techniques to reduce register file accesses in conventional pipeline
  - bypass skip, bypass R0, static strands
  - requires more hardware logic
  - SP approach avoids more register file accesses

Motivation Compilation Results Summary

Related Work

- SP ISA is similar to horizontal microcode, but SP ISA is exposed to the compiler at the microarchitectural level.
- several techniques to reduce register file accesses in conventional pipeline
  - bypass skip, bypass R0, static strands
  - requires more hardware logic
  - SP approach avoids more register file accesses
- Other more fully exposed datapaths include the TTA, NISC, and FlexCore.
  - SP has fewer internal registers.
  - SP relies on several additional compiler optimizations.
  - SP focus is on reducing energy usage.
  - SP evaluation is on much larger benchmarks.

Motivation Compilation Results Summary

Future Work

- Obtain additional performance improvements.
  - Enhance the scheduling of effects.
  - Experiment with more ALUs, internal registers, intermixing 32 bit and 64 bit instructions, etc.
  - Perform more aggressive compiler optimizations to provide more ELP opportunities.
    - loop unrolling
    - software pipelining
- Evaluate energy savings using a VHDL implementation.

Conclusions

- SP allows for a new level of compiler optimizations to reduce energy usage by avoiding redundant or unnecessary operations in conventional pipelines.
- A low-level SP representation can be used and still achieve performance and code size improvements.
- SP reduces energy usage by signficantly reducing register file accesses, internal register writes, branch predictions, branch target address calculations, and completely eliminating the need for a BTB, as well as decreasing execution time.

Questions???