Technical Report UEC-IS-2004-5, Version 2004-05-24
Graduate School of Information Systems, The University of Electro-Communications

SimCore/Alpha RealScalar Simulator Version 1.0

吉瀬 謙二　片桐 孝洋　本多 弘樹　弓場 敏嗣
電気通信大学 大学院情報システム学研究科

本稿では，クロックレベルのスカラプロセッサシミュレータである SimCore/Alpha RealScalar Simulator Version 1.0 (**RealScalar Version 1.0**) の設計と実装について述べる．RealScalar Version 1.0 が模倣するプロセッサは，6 段の命令パイプラインを持つシンプルなスカラープロセッサである．ダイレク・トマップ方式の命令キャッシュ，ダイレクト・マップ方式のデータキャッシュ，分岐予測機構を持つ．RealScalar Version 1.0 の主な特徴は次の通り．(1) クロックレベルのスカラプロセッサシミュレータとして豊富な機能を提供する．(2)SimCore/Alpha Functional Simulator に追加する形で実装され，追加されるコード量は，C++で 1,215 行と少ない．

キーワード システム開発, プロセッサシミュレータ, スカラプロセッサ, SimCore, RealScalar

SimCore/Alpha RealScalar Simulator Version 1.0

Kenji Kise, Takahiro Katagiri, Hiroki Honda, and Toshitsugu Yuba
Graduate School of Information Systems,
The University of Electro-Communications

This paper describes the design and implementation of a SimCore/Alpha RealScalar Simulator Version 1.0 (**RealScalar Version 1.0**). RealScalar Version 1.0 simulates a 6-stage pipelined scalar processor with a direct-mapped instruction cache, a direct-mapped data cache and a branch predictor. The main features of RealScalar Version 1.0 are as follows: (1) It offers plenty functions as a clock level simulator. (2) It is implemented compactly with 1,215 lines by C++.

Key-words system development, processor simulator, scalar processor, SimCore, RealScalar

1 はじめに

プロセッサアーキテクチャ研究やプロセッサ教育のためのツールとして様々なプロセッサシミュレータが利用されている [7]．近年の PC の高速化や PC クラスタの普及により，プロセッサシミュレータを動作させる環境は劇的に向上している．一方で，シミュレータの構築に費す時間は，実装すべきアイデアの複雑化やハードウェアの大規模化に伴い，増加する傾向にある．既存のツールをベースとして開発を進めたとしても，シミュレータの構築に数カ月を必要としながら，その評価は数週間で終るようなケースも珍しくない．

プロセッサ教育と研究の目的で広く利用されているプロセッサシミュレータとして SPIM[5], SimpleScalar[2] が有名である．SimpleScalar には，高速なシミュレーションを主な目的として実装されているために，可読性が失われている面がある．SPIM も同様に，可読性が高いとは言えない．また，プロセッサシミュレータの高速化に関する研究 [6, 11] は多いが，可読性の向上を優先して構築を進めているシミュレータは少ない．

我々は，可読性が高く扱いやすいコードであることを第一の条件としてプロセッサシミュレータの構築をおこなってきた [10, 3, 8]．本稿では，C++を用いて，パイプライン化された現実的なスカラプロセッサの挙動をクロックレベルで模倣するプロセッサシミュレータ SimCore/Alpha RealScalar Version 1.0 の設計と実装について述べる．本稿では，SimCore/Alpha RealScalar Simulator のことを，省略して RealScalar と記述することがある．RealScalar Version 1.0 が模倣するプロセッサは，6 段の命令パイプラインを持つシンプルなスカラープロセッサである．ダイレク・トマップ方式の命令キャッシュ，ダイレクト・マップ方式のデータキャッシュ，分岐予測機構を持つ．近年，プロセッサの複雑さが問題となりつつある．組み込み用途などのシンプルなプロセッサの評価に加えて，このようなシンプルな構成のプロセッサは，複雑さを議論するためのベースとして，また，チップマルチプロセッサのための要素モジュールなどの利用に適している．

RealScalar Version 1.0 は，SimCore/Alpha Functional Simulator に追加される形で実現され，新たに記述したコード量は C++で 1,215 行と非常に少ない．従来のプロセッサの実現方法とは異なり，クロックサイクル内の動

作を，初期化 (Init)，処理 (Step)，更新 (Update) という 3 つのフェーズに分けて記述する．これにより，複雑なシミュレータ内部の依存関係を軽減できる．また，シミュレータの可読性が向上する．

本稿の構成を示す．2 章では RealScalar Version 1.0 の設計をまとめる．3 章では実装に関して述べる．4 章で本稿をまとめる．

2 RealScalar Version 1.0 の設計

2.1 シミュレータとして豊富な機能

RealScalar Version 1.0 は，クロックレベルのスカラプロセッサシミュレータとして豊富な機能を提供する．本節では，RealScalar の主な機能をまとめる．

2.1.1 クロックレベルのプロセッサシミュレータ

RealScalar Version 1.0 の基本的な機能は，スカラプロセッサの挙動をクロックレベルで模倣することにある．また，指定した命令数あるいはアプリケーションの処理が終了するまでのクロック数を測定し，プロセッサの性能に対応する CPI(cycles per instructions) の値を出力する．

起動時に，t オプションを用いることでクロック単位の動作の様子を標準出力に表示する．RealScalar Version 1.0 の動作例として，アセンブラで記述したサンプルコードを図 1 に，このコードを実行した出力を図 2 に示す．

```
1  .text                    ; comment
2  start:                   ;
3          mov 4, $10       ; 0x120000170
4  loop1:                   ;
5          subq $10,  1,$10 ; 0x120000174
6          bne  $10,loop1   ; 0x120000178
7  callsys_exit:            ;
8          mov 1, $0        ; 0x12000017c
9          call_pal 0x83    ; 0x120000180
```

図 1: アセンブラによるサンプルコード.

図 1 のコードは，一つのシンプルなループの構造を持つ．コードの右側には，コメントとして，各命令が格納されるアドレスを付加した．3 行目の mov 命令はレジスタの 10 番に値 4 を格納する．5 行目の subq 命令はレジスタの 10 番の値をデクリメントする．その後，レジスタの 10 番の値が 0 でなければ，6 行目の分岐命令により，4 行目のラベルに分岐する．識別子として，16 進数で記述する命令アドレスの下 3 文字を利用すると，0170, 0174, 0178, 0174, 0178, 0174, 0178, 0174, 0178, 017c, 0180 と処理が進み，合計で 11 個の命令が処理される．

サンプルコードを RealScalar Version 1.0 で処理した出力を図 2 に示す．見やすいように，出力の一部を削除するなどの整形をおこなっている．図 2 の 4 行目から 37 行目は，分岐予測ミスの出力を除いて，1 行に 1 サイクルの動作結果が表示されている．

図 2 の 1 行目は起動コマンド，3 行目は各フィールドの説明である．4 行目は 1 サイクル目の処理の結果を示し

```
1  # SimCore_RealScalar -t -C3 aout.txt
2
3   cycle|  IF   ID   RR   EX   MM   WB  |la:D:IC:DC|
4  000001| ---- ---- ---- ---- ---- ---- |--:-: 3:--|
5  000002| ---- ---- ---- ---- ---- ---- |--:-: 2:--|
6  000003| ---- ---- ---- ---- ---- ---- |--:-: 1:--|
7  000004| 0170 ---- ---- ---- ---- ---- |--:-:--:--|
8  000005| 0174 0170 ---- ---- ---- ---- |--:-:--:--|
9  000006| 0178 0174 0170 ---- ---- ---- |--:-:--:--|
10 000007| 017c 0178 0174 0170 ---- ---- |--:-:--:--|
11 000008| ---- 017c 0178 0174 0170 ---- |--:-: 3:--|
12 000009| pipeline stall | 2
13 000010| pipeline stall | 1
14 000011| ---- ---- ---- 0178 0174 0170 |--:-: 2:--|
15 Branch_Miss_Prediction 00002 (017c -> 0174)
16
17 000012| ---- ---- ---- ---- 0178 0174 |--:-: 1:--|
18 000013| 0174 ---- ---- ---- ---- 0178 |--:-:--:--|
19 000014| 0178 0174 ---- ---- ---- ---- |--:-:--:--|
20 000015| 0174 0178 0174 ---- ---- ---- |--:-:--:--|
21 000016| 0178 0174 0178 0174 ---- ---- |--:-:--:--|
22 000017| 0174 0178 0174 0178 0174 ---- |--:-:--:--|
23 000018| 0178 0174 0178 0174 0178 0174 |--:-:--:--|
24 000019| 0174 0178 0174 0178 0174 0178 |--:-:--:--|
25 000020| 0178 0174 0178 0174 0178 0174 |--:-:--:--|
26 000021| pipeline stall | 2
27 000022| pipeline stall | 1
28 000023| ---- ---- ---- 0178 0174 0178 |--:-:--:--|
29 Branch_Miss_Prediction 00003 (0174 -> 017c)
30
31 000024| 017c ---- ---- ---- 0178 0174 |--:-:--:--|
32 000025| 0180 017c ---- ---- ---- 0178 |--:-:--:--|
33 000026| 0184 0180 017c ---- ---- ---- |--:-:--:--|
34 000027| 0188 0184 0180 017c ---- ---- |--:-:--:--|
35 000028| 018c 0188 0184 0180 017c ---- |--:-:--:--|
36 000029| 0190 018c 0188 0184 0180 017c |--:-:--:--|
37 000030| ---- ---- ---- ---- ---- 0180 |--:-:--:--|
38 === 0 million code(11 code)   0.002 MIPS
39 === SimCore takes 0 min  0 second (6497 usec)
40 === SimCore starts at Thu May 20 14:12:27 2004
41 ===
42 === Pipeline depth       :           6
43 === BTB entry            :         512
44 === Branch_Miss_Penalty  :           2
45 === Conditional   Branch :           4
46 === Cond Branch Hit      :           2
47 === Cond Branch Miss     :           2
48 === Cond Branch Hit Rate :       0.500
49 ===
50 === UnConditional Branch :           0
51 ===
52 === Icache entry         :         512
53 === Icache Miss Penalty  :           3
54 === Icache Access        :          20
55 === Icache Hit           :          18
56 === Icache Miss          :           2
57 === Icache Hit Rate      :       0.900
58 ===
59 === Dcache entry         :         512
60 === Dcache Miss Penalty  :           3
61 === Dcache Access        :           0
62 === Dcache Hit           :           0
63 === Dcache Miss          :           0
64 === Dcache Hit Rate      :         nan
65 ===
66 === Clock Cycles         :          30
67 === CPI(cycles per inst.):       2.727
```

図 2: RealScalar の起動コマンドと出力の例.

ている．この時，最初の命令をフェッチしようとするが，命令キャッシュにミスするので，主記憶から当該命令を含むラインを命令キャッシュに格納しようとする．このために，3 サイクルを必要とする．キャッシュにミスした場合のペナルティは C オプションで指定できる．1 行目の起動コマンドで，C3 というオプションを用いていることに注意すること．IC というフィールドには，命令キャッシュにミスし，必要とするラインが利用できるようになるまでのサイクル数を表示する．

7 行目，4 サイクル目で最初の命令がフェッチされる．IF フィールドには，識別子として，フェッチされた命令のアドレスを 16 進数で記述した際の下 3 文字を表示する．7 サイクル目でフェッチした命令は，サイクルが進むに従って，命令デコード (ID)，レジスタリード (RR) と処理が進む．

12 行目では，識別子 0178 の分岐命令が実行ステージ (EX) において，分岐結果を生成し，その結果と分岐予測結果との比較から分岐予測ミスが検出される．このため，分岐予測ミスペナルティの 2 サイクルを費やして (12 行目と 13 行目のストール)，分岐予測ミスから回復し，17 行目の 12 サイクル目から処理を再開する．分岐予測ミスのペナルティは変数 branch_miss_penalty を用いて設定する．分岐予測ミスからの回復の間は，メインメモリを含むプロセッサ外の全ての処理が停止するシステムを想定する．このため，この期間には，命令キャッシュの遅延は変わらない．

18 行目で，分岐先のアドレスから命令をフェッチして処理を再開する．37 行目，30 サイクル目で 11 番目の最後の命令の処理が完了し，シミュレーションが終了する．

38 行目から 67 行目は設定パラメータと統計データの出力である．67 行目に示す様に，11 命令の実行に 30 サイクルを費やすことから，CPI として 2.727 という値を得ることができる．

2.1.2 命令パイプラインとキャッシュレイテンシの変更

起動時のオプション指定により，キャッシュレイテンシ (C オプション) と命令パイプラインの構成 (P オプション) を変更することができる．標準設定のパイプライン段数は 6 段だが，P オプションを用いて，命令パイプラインの構成を変更した場合には，命令デコード (ID) とレジスタリード (RR) の間に，何もおこなわないステージが挿入される．

パイプラインの段数を変更した場合の例として，パイプラインの長さを 7 段に変更して，図 1 のサンプルプログラムを実行した出力を図 3 に示す．1 行目に実行コマンドを示す．P オプションを用いてパイプライン長を 7 に設定している．また，C オプションを用いてキャッシュのレイテンシを 2 サイクルに設定している．3 行目に示した様に，命令デコード (ID) とレジスタリード (RR) の間に何も処理をおこなわないステージが挿入される．

近年のプロセッサは，命令パイプラインの段数が深くなる傾向にある．RealScalar Version 1.0 の持つ，命令パイプラインの段数を変更できる機能は，これらのプロセッサを想定した場合の評価に利用できる．

```
 1  # SimCore_RealScalar -t -P7 -C2 aout.txt
 2
 3   cycle|  IF   ID        RR   EX   MM   WB  |la:D:IC:
 4  000001| ---- ---- ---- ---- ---- ---- ---- |--:-: 2:
 5  000002| ---- ---- ---- ---- ---- ---- ---- |--:-: 1:
 6  000003| 0170 ---- ---- ---- ---- ---- ---- |--:-:--:
 7  000004| 0174 0170 ---- ---- ---- ---- ---- |--:-:--:
 8  000005| 0178 0174 0170 ---- ---- ---- ---- |--:-:--:
 9  000006| 017c 0178 0174 0170 ---- ---- ---- |--:-:--:
10  000007| ---- 017c 0178 0174 0170 ---- ---- |--:-: 2:
11  000008| ---- ---- 017c 0178 0174 0170 ---- |--:-: 1:
12  000009| pipeline stall | 2
13  000010| pipeline stall | 1
14  000011| ---- ---- ---- ---- 0178 0174 0170 |--:-:--:
15  Branch_Miss_Prediction 00002 (017c -> 0174)
16
17  000012| 0174 ---- ---- ---- ---- 0178 0174 |--:-:--:
18  000013| 0178 0174 ---- ---- ---- ---- 0178 |--:-:--:
19  000014| 017c 0178 0174 ---- ---- ---- ---- |--:-:--:
20  000015| 0180 017c 0178 0174 ---- ---- ---- |--:-:--:
21  000016| 0184 0180 017c 0178 0174 ---- ---- |--:-:--:
22  000017| pipeline stall | 2
23  000018| pipeline stall | 1
24  000019| ---- ---- ---- ---- 0178 0174 ---- |--:-:--:
25  Branch_Miss_Prediction 00003 (017c -> 0174)
26
27  000020| 0174 ---- ---- ---- ---- 0178 0174 |--:-:--:
28  000021| 0178 0174 ---- ---- ---- ---- 0178 |--:-:--:
29  000022| 0174 0178 0174 ---- ---- ---- ---- |--:-:--:
30  000023| 0178 0174 0178 0174 ---- ---- ---- |--:-:--:
31  000024| 0174 0178 0174 0178 0174 ---- ---- |--:-:--:
32  000025| 0178 0174 0178 0174 0178 0174 ---- |--:-:--:
33  000026| 0174 0178 0174 0178 0174 0178 0174 |--:-:--:
34  000027| pipeline stall | 2
35  000028| pipeline stall | 1
36  000029| ---- ---- ---- ---- 0178 0174 0178 |--:-:--:
37  Branch_Miss_Prediction 00004 (0174 -> 017c)
38
39  000030| 017c ---- ---- ---- ---- 0178 0174 |--:-:--:
40  000031| 0180 017c ---- ---- ---- ---- 0178 |--:-:--:
41  000032| 0184 0180 017c ---- ---- ---- ---- |--:-:--:
42  000033| 0188 0184 0180 017c ---- ---- ---- |--:-:--:
43  000034| 018c 0188 0184 0180 017c ---- ---- |--:-:--:
44  000035| 0190 018c 0188 0184 0180 017c ---- |--:-:--:
45  000036| 0194 0190 018c 0188 0184 0180 017c |--:-:--:
46  000037| ---- ---- ---- ---- ---- ---- 0180 |--:-:--:
```

図 3: RealScalar の起動コマンドと出力の例．命令パイプラインの構成を 7 段に，キャッシュレイテンシを 2 サイクルとした場合．

2.1.3 動作検証機能の提供

RealScalar Version 1.0 は，プロセッサシミュレータの開発時に，バグの早期発見を支援する 2 つの機能を提供する．

動作検証の機能の一つとして，一定の期間に処理が進んでいないこと (完了した命令の数がゼロであること) を検出するウォッチドッグタイマの機能を提供する．標準の設定では，256 サイクルの間に処理された命令数がゼロの場合を異常として，エラー出力とともにシミュレー

```
 1  #define WD_INT 0xff /** watch dog interval **/
 2  void scalar_chip::house_keeping(){
 3    static int wd_cnt = 0; /* watch dog count */
 4
 5    e2->clock++;
 6
 7    /*** watch dog timer check by 0xff cycles ***/
 8    if((e2->clock & WD_INT)==WD_INT){
 9      if(wd_cnt==0){
10        printf("\nWatch Dog Timer: retired %lld\n",
11               e->retired_inst);
12        exit(1);
13      }
14      wd_cnt=0;
15    }
```

図 4: ウォッチドッグタイマの実装.

```
 1  ** Verify Error: 446 code retire
 2  ** simple_chip / real_scalar
 3  Cpc:       1200089a0        1200089a0
 4  Npc:       1200089a4        1200089a4
 5  Rav:               0                0
 6  Rbv:       11ff96fd0        11ff96fd0
 7  Rcv:       120007e1c        120007e1c
 8  Adr:       11ff96fd0        11ff96fd0
 9  v0   00:0000000000000051 0000000000000051
10  t0   01:0000000000000000 000000000000777*
11  t1   02:0000000000000009 0000000000000009
12  t2   03:fffffffffffffd0 fffffffffffffd0
13  t3   04:0000000000000200 0000000000000200
14  t4   05:0000000000000000 0000000000000000
15  t5   06:000000007fffffff 000000007fffffff
16  t6   07:0000000000000000 0000000000000000
17  t7   08:0000000140015a84 0000000140015a84
18  s0   09:000000011ff97008 000000011ff97008
19  s1   10:0000000000000000 0000000000000000
20  s2   11:0000000000000000 0000000000000000
21  s3   12:0000000000000000 0000000000000000
22  s4   13:0000000000000000 0000000000000000
23  s5   14:0000000000000000 0000000000000000
24  fp   15:0000000000000000 0000000000000000
25  a0   16:0000000000000009 0000000000000009
26  a1   17:0000000000000009 0000000000000009
27  a2   18:0000000120006128 0000000120006128
28  a3   19:000000011ff97188 000000011ff97188
29  a4   20:0000000000000040 0000000000000040
30  a5   21:0000000000000000 0000000000000000
31  t8   22:0000000000000000 0000000000000000
32  t9   23:0000000000000010 0000000000000010
33  t10  24:0000000000000000 0000000000000000
34  t11  25:0000000000000000 0000000000000000
35  ra   26:0000000120007ed8 0000000120007ed8
36  t12  27:0000000000000000 0000000000000000
37  at   28:0000000000000000 0000000000000000
38  gp   29:0000000140023e90 0000000140023e90
39  sp   30:000000011ff96fd0 000000011ff96fd0
40  zero 31:0000000000000000 0000000000000000
```

図 5: 意図的に間違った値でレジスタ 01 番を書き換えた場合の出力例.

ションを停止する．この機能により，何らかの理由で，シミュレーションが終了しないという状態を回避できる．図 4 に，ウォッチドッグタイマの実装を示す．1 行目の define 文で，検出のためのインターバルを設定する．関数 house_keeping は，毎サイクルに 1 回呼び出される．3 行目で宣言されている wd_cnt という変数は，ゼロで初期化され，命令の処理が完了する度にインクリメントされる．9 行目の判定において，この変数の値が 0 の場合には，256 サイクルの間に処理が進んでいないことを意味するので，異常と判断される．

二つ目の動作検証の機能として，一つの命令の処理が完了する度に，機能レベルシミュレータによるアーキテクチャステートと比較し，アーキテクチャステートのレベルで正しい動作を検証する機能を提供する．RealScalar Version 1.0 を起動する際に，z オプションを用いることで，クロックレベルのシミュレータと機能レベルのシミュレータの 2 つのオブジェクトを生成し，これらのアーキテクチャステートを比較しながら処理を進める．例として，1000 サイクルの処理が終わった時点で意図的に値 0x777 でレジスタ 01 番を上書きするように変更を加えた RealScalar Version 1.0 の z オプションを用いた出力を図 5 に示す．出力結果の 9 行目から 40 行目には，32 個の汎用レジスタの値が表示されている．左側が機能レベルシミュレータの値，右側がクロックレベルシミュレータの値である．この例では，10 行目のレジスタ 01 番の値が，機能レベルシミュレータでは 0 となっているが，クロックレベルシミュレータでは 0x777 になっており，これらの値が一致していないことがわかる．10 行目の様に，値が異なる所にはアスタリスク (*) を出力する．

RealScalar Version 1.0 は，これら動作検証の機能を提供し，バグの早期発見を支援する．

2.2 コンパクトな記述

RealScalar Version 1.0 では，シミュレーション速度に関する十分な最適化が施されていない．コンパクトな記述と可読性の向上を優先して記述した．

```
 1    scalar.cc ------------- 678
 2    fetch.cc --------------  80
 3    cache.cc -------------- 217
 4    scalar.h -------------- 240
 5        SimCore/
 6         sim.cc -----------  19
 7         chip.cc ----------  75
 8         instruction.cc --- 231
 9         memory.cc -------- 297
10         arithmetic.cc ---- 468
11         debug.cc --------- 168
12         syscall.cc ------- 578
13         etc.cc ----------- 378
14         define.h --------- 550
```

図 6: RealScalar のファイル構成．ファイル名と行数を列挙した.

RealScalar Version 1.0 のファイル構成 (ファイル名と行

数) を図 6 にまとめる．全てのコードは C++を用いて記述されている．RealScalar は，SimCore/Alpha Functional Simulator [8] に追加される形で実現されており，6 行目から 14 行目に示したファイルは，SimCore/Alpha Functional Simulator のものと同一である．RealScalar Version 1.0 として新たに記述したコードは scalar.cc, fetch.cc, cache.cc, scalar.h という 4 つのファイルにまとめられており，これらのコード量の合計は 1,215 行と非常に少ない．

コンパクトな記術は RealScalar Version 1.0 の特徴の一つである．可読性の向上のために，SimCore/Alpha Functional Simulator と同様に，グローバル変数，条件付きコンパイル，定数以外のマクロ，goto 文を利用していない．

2.3 明快な動作記述方式の採用

クロックレベルで動作を模倣するプロセッサシミュレータを C++で記述する場合には，Verilog-HDL などのハードウェア記述言語と異なり，プロセッサのレジスタの更新のタイミングに注意する必要がある．

SimpleScalar などのクロックレベルシミュレータでは，プロセッサの動作を命令パイプラインに対して逆順 (reverse order) に進めていくことで，1 クロックの動作を実現する．この場合には，レジスタの値が利用されるタイミングと更新されるタイミングを厳密に管理しなければ，正しい動作のシミュレータを構築することができない．このため，逆順の記述方式を採用するシミュレータには，記述が困難になるという欠点がある．

この欠点を解決するために，RealScalar では，クロックサイクル内の動作を Init フェーズ, Step フェーズ, Update フェーズという 3 つに分割する記述方式を採用する．この様子を図 7 に示す．また，シミュレータの記述において配線という概念を導入する．Init フェーズでは幾つかの配線の初期値を設定する．Step フェーズにおいて幾つかの配線の値を確定する．Update フェーズでは，先行するフェーズで確定した配線の値を利用して，パイプラインレジスタなどのレジスタの値を更新する．

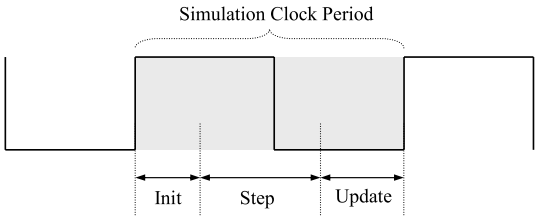

図 7: クロック内の動作を 3 つのフェーズに分割して記述する方式を採用.

3 つのフェーズの関係は RealScalar のメインループのコードに明示的に現れる．図 8 に示すメインループの 6 行目が Init フェーズである．9 行目から 17 行目までが Step フェーズである．20 行目から 24 行目までが Update フェーズである．このように，フェーズを明確に区別することで，フェーズに跨る関数の移動は許されないが，フェーズ内の関数の順番を意識する必要がなくなる．例えば，9 行目から 17 行目までの 9 個の関数呼び出しの順番を変更したとしても，シミュレーションの結果は変わらない．

```
 1  void scalar_chip::main_loop(){
 2    while(ev->sys->running){
 3      if(stall) stall--;
 4      else{
 5        /******** (1) Init    ********/
 6        Init();
 7
 8        /******** (2) Step    ********/
 9        Step();
10        pc_gen->Step(w); /* IF */
11        b_pred->Step(w); /* IF */
12        icache->Step(w); /* IF */
13        dcache->Step(w); /* MM */
14        IDecode();       /* ID */
15        RegRead();       /* RR */
16        Execute();       /* EX */
17        WriteBack();     /* WB */
18
19        /******** (3) Update  ********/
20        Update();
21        pc_gen->Update(w);
22        b_pred->Update(w);
23        icache->Update(w);
24        dcache->Update(w);
25      }
26      /******** etc.        ********/
27      ev->e2->clock++;
28      house_keeping();
29    }
30  }
```

図 8: RealScalar のメインループ.

28 行目の house_keeping という関数は，データの収集や動作検証の機能を提供するために呼び出される．

3 RealScalar Version 1.0 の実装

本章では，RealScalar Version 1.0 が模倣するプロセッサアーキテクチャを定義する．また，RealScalar の主な関数のコードを示し，それぞれの関数の動作を解説する．

3.1 プロセッサアーキテクチャ

RealScalar Version 1.0 は，図 9 に示すシンプルなスカラプロセッサを模倣する．プロセッサの主な特徴と標準のパラメータを列挙する．ここに示した幾つかのパラメータは変更可能である．

- 6 段の命令パイプライン構成．命令の演算レイテンシは，整数の乗算が 4 サイクル，浮動小数点演算が 5 サイクル.
- ラインサイズ 32B, ダイレクト・マップ方式，512 エントリ，容量 4KB の命令キャッシュを備える．
- ラインサイズ 32B, ダイレクト・マップ方式，512 エントリ，ノー・ライト・アロケートの容量 4KB のデータキャッシュを備える．

- キャッシュは 1 階層のみの構成で，L2 キャッシュは存在しない．命令キャッシュとデータキャッシュの参照は 1 サイクル，これらのキャッシュにミスした場合のペナルティは 8 サイクル．
- 分岐予測機構として，2 ビット飽和型カウンタを用いた分岐先バッファを備える．分岐予測ミスのペナルティは 2 サイクル．分岐先バッファはダイレクト・マップ方式で 256 エントリ．

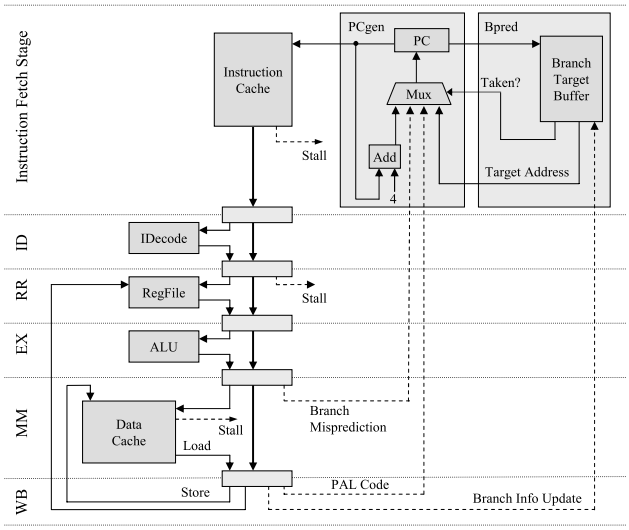

図 9: RealScalar Version 1.0 のプロセッサ構成.

以降の節では，図 9 に示したプログラムカウンタ生成ユニット PCgen，命令キャッシュユニット Icache，データキャッシュユニット Dcache の順番に説明をおこなう．その後，プロセッサの設定を変更するための方法と，プロセッサの動作タイミングの実装に関して説明をおこなう．

3.2 プログラムカウンタ生成ユニット PCgen

プログラムカウンタ生成ユニットの構成を図 10 に示す．このユニットは，様々な入力を用いて，左上に示した，フェッチする命令が格納されているアドレスを指示する pcgen_pc を出力する．

プログラムカウンタ生成ユニットのコードを図 11 に示す．基本的に，それぞれの機能ユニットの記述は，図 7 に示した Init, Step, Update という 3 つのフェーズに分割して記述される．

6 行目は Init フェーズのコードだが，その内容はない．

8 行目から 10 行目は Step フェーズのコードで，プログラムカウンタの値に 4 を加えた値を配線 pcgen_npc に設定する．

12 行目から 22 行目が Update フェーズのコードで，幾つかのフラグの内容に従ってプログラムカウンタの値を更新する．PAL 命令が実行された場合には，pcgen_pal にゼロ以外の値がセットされる．この時，pcgen_pal の値を

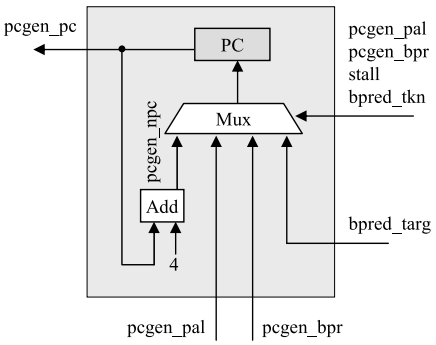

図 10: プログラムカウンタ生成ユニット PCgen.

```
 1  PCgen::PCgen(Env_s *e){ /* Constructor */
 2    ev = e;
 3    pc= e->as->pc; /* Entry PC */
 4  }
 5
 6  void PCgen::Init(chip_wire *w){}
 7
 8  void PCgen::Step(chip_wire *w){
 9    w->pcgen_npc = pc+4;
10  }
11
12  void PCgen::Update(chip_wire *w){
13    int stall =  w->data_stall | w->code_stall |
14      w->icache_stall | w->dcache_stall;
15
16    pc =
17      (w->pcgen_pal) ? w->pcgen_pal  :
18      (w->pcgen_bpr) ? w->pcgen_bpr  :
19      (stall)        ? pc            :
20      (w->bpred_tkn) ? w->bpred_targ :
21      w->pcgen_npc;
22  }
```

図 11: プログラムカウンタ生成ユニット PCgen のコード.

プログラムカウンタにセットする (17 行目)．分岐命令が実行され，分岐が成立だった場合には，pcgen_bpr にゼロ以外の値がセットされる．この時， pcgen_bpr の値をプログラムカウンタにセットする．キャッシュミスなどの場合に，プロセッサがストールしている間は，同じ値を保持するために，前のクロックの値を書き込む (19 行目)．分岐予測が分岐成立と予測した場合には，bpred_targ の値をセットする (20 行目)．

3.3 分岐予測ユニット Bpred

分岐先バッファを用いた分岐予測ユニット Bpred の構成を図 12 に示す．履歴テーブルは，セットアソシアティブ方式ではなく，ダイレクトマップの構成を採用する．

図 12 に示した Predict というフィールドには分岐予測のための 2 ビットの情報が格納されている．この情報は，図 13 に示す状態遷移に従って更新される (Bimodal 分岐予測)．

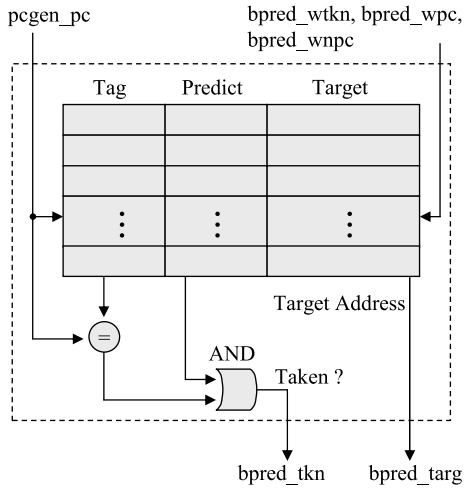

図 12: 分岐先バッファを用いた分岐予測ユニット.

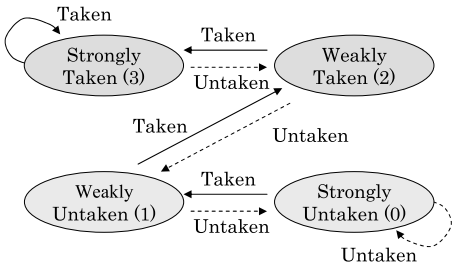

図 13: 分岐予測に利用する 2 ビット飽和型カウンタ.

図 12 の左上に示した命令フェッチアドレス pcgen_pc を用いてインデックスを生成して，分岐先バッファを参照する．参照したエントリのタグと命令フェッチアドレスが一致して，かつ，エントリの有効ビットがセットしてある場合に，BTB ヒットとなる．そうでない場合には，BTB ミスとなる．BTB ヒットの場合には，参照しているエントリから，分岐先アドレス bpred_targ と分岐予測結果 bpred_tkn を出力する．BTB ミスの場合には，当該分岐命令に関する情報が登録されていない．また，分岐が成立と予測できたとしても分岐先アドレスが利用できないので，分岐分成立と予測して，当該分岐命令の後続の命令をフェッチする．このため，タグが一致し，かつ，2 ビットカウンタが Taken と予測した場合のみ，最終的に分岐成立と予測される．

図 14 に，クラス BTB の定義とコンストラクタのコードを示す．1 行目から 6 行目のクラス btb_entry は，図 12 に示した BTB の一つのエントリに対応する．変数として，tag, target, predict を持つ．

8 行目から 16 行目が，分岐予測ユニット Bpred の定義である．18 行目から 27 行目は Bpred のコンストラクタである．各エントリの tag はゼロで初期化される (23 行目)．各エントリの target はゼロで初期化される (24 行目)．25 行目で，各エントリの predict を図 13 の右上の

```
 1  class btb_entry { /*** BTB Entry ***/
 2   public:
 3    ADDR_TYPE tag;
 4    ADDR_TYPE target;
 5    int predict;
 6  };
 7
 8  class Bpred{ /*** Branch Target Buffer ***/
 9    int size;                /* entry size */
10    btb_entry *ent;          /* table      */
11   public:
12    Bpred(Env_s*);           /* Constructor */
13    void Init(chip_wire*);   /* Reserved */
14    void Step(chip_wire*);   /* Reserved */
15    void Update(chip_wire*); /* Reserved */
16  };
17
18  #define WEAK_TKN 2  /* Initial Value */
19  Bpred::Bpred(Env_s *ev){
20    size = ev->sc2->btb_entry;
21    ent = new class btb_entry[size];
22    for(int i=0; i<size; i++){
23      ent[i].tag     = 0;
24      ent[i].target  = 0;
25      ent[i].predict = WEAK_TKN;
26    }
27  }
```

図 14: クラス BTB の定義とコンストラクタのコード.

状態の weakly taken に初期化する．

図 15 に，分岐予測ユニット Bpred の Init, Step, Update という 3 つのフェーズのコードを示す．1 行目は Init フェーズのコードだが，その内容はない．

3 行目から 11 行目が Step フェーズのコードである．5 行目でインデックスを生成する．定数 INST_OST は，命令長に依存するオフセットである．Alpha AXP アーキテクチャの命令長が 32 ビットであり，アドレスの下位 2 ビットは必ずゼロとなるので，定数 INST_OST は 2 に設定されている．6 行目で選択された履歴テーブルのエントリをポインタ変数 e に格納する．8 行目と 9 行目で分岐予測 bpred_tkn を計算する．エントリのタグとプログラムカウンタが等しく，かつ，2 ビットカウンタの値が weakly taken 以上だった場合に，分岐成立 (B_TKN) と予測する．そうでない場合には分岐不成立 (B_UKN) と予測する．ここで，タグ領域として，下位 2 ビットを除く，62 ビットを利用している点に注意する必要がある．ハードウェア量を考慮すると，予測精度を大きく低下させない範囲で，このタグのビット幅を適切に小さくする必要がある．

13 行目から 32 行目が Update フェーズのコードである．ここでは，分岐命令の実行結果を用いて，履歴テーブルの値を更新する．分岐命令のアドレスと更新すべきエントリのタグが一致し (21 行目)，分岐結果が成立の場合にはカウンタをインクリメントし (22 行目)，分岐結果が不成立の場合にはカウンタをデクリメントする (23 行目)．タグが一致しなかった場合には，tag, predict の値を初期化する．30 行目で，target のアドレスを登録する．

```
 1  void Bpred::Init(chip_wire *w){}
 2
 3  void Bpred::Step(chip_wire *w){
 4    ADDR_TYPE pc  = w->pcgen_pc;
 5    int index     = (pc>>INST_OST) % size;
 6    btb_entry *e  = &ent[index];
 7    int bc        = e->predict;
 8    w->bpred_tkn  = (e->tag==pc &&
 9                     bc>=WEAK_TKN) ? B_TKN : B_UNT;
10    w->bpred_targ = e->target;
11  }
12
13  void Bpred::Update(chip_wire *w){
14    if(w->bpred_wpc!=0){
15      int taken     = w->bpred_wtkn;
16      ADDR_TYPE pc  = w->bpred_wpc;
17      ADDR_TYPE npc = w->bpred_wnpc;
18      int index     = (pc>>INST_OST) % size;
19      btb_entry *e  = &ent[index];
20
21      if(e->tag==pc){
22        if( taken && e->predict<3) e->predict++;
23        if(!taken && e->predict>0) e->predict--;
24      }
25      else{
26        e->tag     = pc;
27        e->predict = WEAK_TKN;
28      }
29
30      e->target = (taken) ? npc : e->target;
31    }
32  }
```

図 15: 分岐予測器ユニット Bpred のコード.

3.4 命令キャッシュユニット Icache

RealScalar Version 1.0 は，ダイレクトマップ方式の命令キャッシュを持つ．命令キャッシュユニット Icache の構成を図 16 に示す．図 16 の左上に示したプログラムカウンタ pcgen_pc の値から，インデックスを生成して，命令キャッシュのエントリ (キャッシュライン) を選択する．有効ビットがセットされており，タグが一致した場合には，ラインにヒットしたことになり，キャッシュラインの適切な命令 (4 バイト) を出力する．

命令キャッシュユニット Icache の定義とコンストラクタを図 17 示す．1 行目と 2 行目がパラメータの設定である．キャッシュの容量やキャッシュラインの幅を変更する場合には，これらの値を変更する．1 行目の定数 ICLINE_SIZE は，命令キャッシュのラインサイズを定義する．標準では 64 バイトに設定されている．

4 行目から 9 行目は，命令キャッシュの一つのライン icache_line の定義である．変数として，有効ビット，タグ，データ領域を持つ．

11 行目から 24 行目は，命令キャッシュユニット Icache の定義である．20 行目の ld_line というメソッドは，指定したメモリアドレスを含むキャッシュラインをメインメモリから命令キャッシュに転送する．命令キャッシュのデータを変更することはない (リードオンリー) ので，キャッシュに存在するラインをメインメモリに書き戻すメソッ

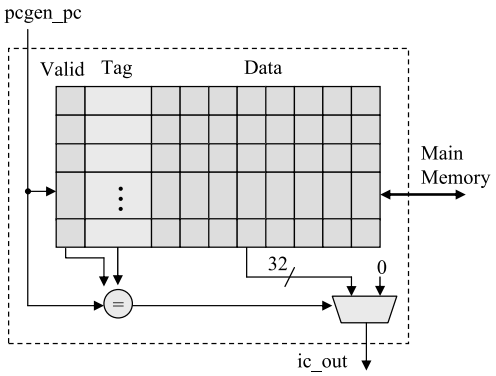

図 16: 命令キャッシュユニット Icache の構成.

```
 1  #define ICLINE_SIZE 64  /* size in byte        */
 2  #define ICLINE_SFT   6  /* 2^x = ICLINE_SIZE   */
 3
 4  class icache_line {
 5   public:
 6    int valid;
 7    ADDR_TYPE tag;
 8    INST_TYPE d[ICLINE_SIZE/ICODE_SIZE]; /* =16 */
 9  };
10
11  class Icache {
12    Env_s *ev;
13    int size;  /* the number of cache lines */
14    int nw;    /* the number of word in line */
15    icache_line *buf;
16    int stall; /* cache status */
17    ADDR_TYPE mis_pc; /* PC of the cache Miss */
18   public:
19    Icache(Env_s*);
20    void ld_line(ADDR_TYPE);
21    void Init(chip_wire*);   /* Reserved */
22    void Step(chip_wire*);   /* Reserved */
23    void Update(chip_wire*); /* Reserved */
24  };
25
26  Icache::Icache(Env_s *env){
27    stall = 0;
28    ev = env;
29    nw = ICLINE_SIZE/ICODE_SIZE;
30    size = ev->sc2->icache_entry;
31    buf  = new icache_line[size];
32    for(int i=0; i<size; i++){
33      buf[i].valid = 0;
34      buf[i].tag   = 0;
35    }
36  }
```

図 17: 命令キャッシュユニット Icache の定義とコンストラクタのコード.

ドは存在しない．

26 行目から 36 行目は，命令キャッシュユニット Icache のコンストラクタである．全ての有効ビットとタグをリセットする．エントリを利用する前に，必ずメインメモリからデータをロードするので，データ領域の値を初期化する必要はない．

```
 1  void Icache::Step(chip_wire *w){
 2    ADDR_TYPE pc     = w->pcgen_pc;
 3    ADDR_TYPE tag    = (pc>>ICLINE_SFT);
 4    int       index  = (pc>>ICLINE_SFT) % size;
 5    int       offset = (pc>>INST_OST) % nw;
 6
 7    icache_line *e = &buf[index];
 8
 9    if(stall){ /** Waiting for the data **/
10      w->icache_stall = stall;
11    }
12    else{
13      /**** I-Cache Hit ****/
14      if(e->valid && e->tag==tag){
15        stall           = 0;
16        w->icache_stall = 0;
17        ev->e2->icache_access++;
18      }
19      /**** I-Cache Miss ****/
20      else{
21        stall           = ev->sc2->icache_penalty;
22        w->icache_stall = ev->sc2->icache_penalty;
23        mis_pc = pc;
24        ev->e2->icache_mis++;
25      }
26    }
27    w->ic_out = (stall) ? 0 : e->d[offset];
28  }
29
30  void Icache::Update(chip_wire *w){
31    if(stall>0) stall--;
32    if(stall==1) ld_line(mis_pc);
33  }
34
35  void Dcache::Init(chip_wire *w){}
```

図 18: 命令キャッシュユニット Icache のコード．

図 18 に，命令キャッシュユニット Icache のコードを示す．1 行目から，28 行目までが Step フェーズのコードである．命令キャッシュユニットは，メインメモリの参照の状態を表す stall という変数を持つ．この変数がゼロ以外の値を持つ場合は，キャッシュミスが発生し，データ待ちの状態を意味するので，キャッシュの出力をゼロとする (27 行目)．そうでない場合には，キャッシュラインのオフセットが示すデータ d[offset] を出力する．

ストールが発生していない場合で，有効ビットがセットされタグが一致した場合 (14 行目) には，キャッシュのヒットとなる．この時，キャッシュの状態を適切に設定 (15 行目と 16 行目) し，シミュレーション終了後の統計データの出力のためにヒットの回数をインクリメント (17 行目) する．

キャッシュミスの場合 (20 行目から 25 行目) には，メインメモリからデータが到着するまでのサイクル数を表す変数 stall に icache_penalty の値を代入する (21 行目)．また，シミュレーション終了後の統計データの出力のためにミスの回数をインクリメント (24 行目) する．

Update フェーズのコードが 30 行目から 33 行目である．変数 stall がゼロより大きい場合 (31 行目) には，この値をデクリメントする．また，この変数の値が 1 の時に，メソッド ld_line を呼び出すことで (32 行目)，当該キャッシュラインをメインメモリからロードする．これにより，次のサイクルでは，変数 stall がゼロになるとともに，必要とするキャッシュラインがデータキャッシュユニットに格納され利用できるようになる．

3.5 データキャッシュユニット Dcache

データキャッシュユニット Dcache の多くの部分は，先の節で述べた Icache と同様に動作する．ただし，データキャッシュに存在するデータは更新されることがある．このために，幾つかの点で構造が複雑になっている．RealScalar Version 1.0 は，ダイレクトマップ方式，ライトノーアロケイト，ライトスルーのデータキャッシュを持つ．

図 19 に，データキャッシュユニット Dcache の定義とコンストラクタのコードを示す．6 行目から 12 行目が，データキャッシュの一つのライン dcache_line の定義である．ラインがロードされた後に，データが変更されているかを識別するための変数 dirty(9 行目) が追加されている．

14 行目から 43 行目がクラス Dcache の定義である．命令キャッシュと比較して，キャッシュラインをメインメモリにストアする st_line というメソッド (24 行目) と，データキャッシュの内容をフラッシュする flush(25 行目) というメソッドが追加されている．

図 20 に，データキャッシュユニット Dcache の Step フェーズのコードを示す．

4 行目の条件文で読み出すべきデータの有無を判定する．もし dc_radr がゼロととなり，読み出すデータが無い場合には，当該メソッドの処理を終了する．キャッシュミスが発生してデータの到着を待っている場合にも，当該メソッドの処理を終了 (9 行目) する．

これらの場合を除いて，12 行目以降のコードが実行される場合には，プロセッサ内でロード命令が処理され，キャッシュに参照があったということを意味する．この場合，有効ビットとタグが一致するかを判定 (20 行目) し，条件が真の場合にはキャッシュヒットとして，18 行目から 38 行目までのブロックを実行する．そうでなければ，キャッシュミスとして，39 行目から 46 行目までのブロックを実行する．

キャッシュがヒットした場合には，まず，当該データを変数 tmp に格納する (22 行目)．その後，ストア命令とロード命令のフォワーディングの検出をおこなう．先行するストア命令と，当該ロード命令が同一のアドレスを参照する場合には，キャッシュにヒットしたとしても，キャッシュ内のデータではなく，ストア命令が書き込んだデータを供給する必要がある．このことを 25 行目で判定し，一致した場合には，26 行目から 33 行目でデータを準備して，変数 tmp を上書きする．

```
 1  #define DCLINE_SIZE 64  /* size in byte, 8 word */
 2  #define  DDATA_SIZE  8  /* data size in byte    */
 3  #define DCLINE_SFT   6  /* 2^x = DCLINE_SIZE    */
 4  #define  DDATA_SFT   3  /* 2^x = DDATA_SIZE     */
 5
 6  class dcache_line {
 7  public:
 8    int valid;
 9    int dirty;
10    ADDR_TYPE tag;
11    data_t d[DCLINE_SIZE/DDATA_SIZE];
12  };
13
14  class Dcache {
15    Env_s *ev;
16    int size;   /* the number of cache lines */
17    int nw;     /* the number of word in line */
18    dcache_line *buf;
19    int stall; /* cache status */
20    ADDR_TYPE mis_adr; /* Adr of the cache Miss */
21   public:
22    Dcache(Env_s*);
23    void ld_line(ADDR_TYPE);
24    void st_line(ADDR_TYPE);
25    void flush();
26    void Init(chip_wire*);   /* Reserved */
27    void Step(chip_wire*);   /* Reserved */
28    void Update(chip_wire*); /* Reserved */
29  };
30
31  Dcache::Dcache(Env_s *env){
32    ev = env;
33    nw = DCLINE_SIZE/DDATA_SIZE;
34    size = ev->sc2->dcache_entry;
35    buf  = new dcache_line[size];
36
37    stall=0;
38    for(int i=0; i<size; i++){
39      buf[i].valid = 0;
40      buf[i].dirty = 0;
41      buf[i].tag   = 0;
42    }
43  }
```

図 19: データキャッシュユニット Dcache の定義とコンストラクタのコード.

```
 1  void Dcache::Step(chip_wire *w){
 2    w->dcache_stall = stall; /* default */
 3
 4    if(w->dc_radr==0) return;
 5
 6    if(stall){ /***** Waiting for the data *****/
 7      w->dcache_stall = stall;
 8      return;
 9    }
10
11    /** Load Instruction **/
12    ADDR_TYPE adr = w->dc_radr;
13    ADDR_TYPE tag = (adr>>DCLINE_SFT);
14    int     index = (adr>>DCLINE_SFT) % size;
15    int    offset = (adr>>DDATA_SFT ) % nw;
16    dcache_line *cl = &buf[index];
17
18    if(cl->valid && cl->tag==tag){ /*** DC Hit ***/
19      stall = 0;
20      w->dcache_stall = 0;
21      ev->e2->dcache_access++;
22      data_t tmp = cl->d[offset];
23
24      /*** Store -> Load data forwarding ***/
25      if((w->dc_radr>>3)==(w->dc_wadr>>3)){
26        data_t dat;
27        DATA_TYPE mask;
28        int ci = get_code(w->dc_wir);
29        set_st_data(ci, st_bytes(ci),
30                    &w->dc_wadr,
31                    &w->dc_wdat, &dat, &mask);
32        tmp= ((cl->d[offset] & mask) | dat);
33      }
34
35      int ci = get_code(w->dc_rir);
36      set_ld_data(ci, ld_bytes(ci),  &w->dc_radr,
37                  &tmp, &w->dc_rdat);
38    }
39    else{  /*** D-Cache Miss ***/
40      stall = ev->sc2->dcache_penalty;
41      mis_adr = adr;
42      w->dcache_stall = stall;
43      st_line(adr);
44      ld_line(adr);
45      ev->e2->dcache_mis++;
46    }
47  }
```

図 20: データキャッシュユニット Dcache の Step フェーズ.

```
 1  void Dcache::Update(chip_wire *w){
 2    if(stall>0) stall--;
 3    if(w->dc_wadr==0) return;
 4
 5    data_t dat;
 6    DATA_TYPE mask;
 7    int ci=get_code(w->stg[ev->sc2->P_WRITEBK]->ir);
 8    set_st_data(ci, st_bytes(ci), &w->dc_wadr,
 9                &w->dc_wdat, &dat, &mask);
10
11    /***** Store to the cache *****/
12    ADDR_TYPE adr = w->dc_wadr;
13    ADDR_TYPE tag = (adr>>DCLINE_SFT);
14    int     index = (adr>>DCLINE_SFT) % size;
15    int    offset = (adr>>DDATA_SFT ) %
16      (DCLINE_SIZE/DDATA_SIZE);
17    dcache_line *cl = &buf[index];
18
19    /**** D-Cache Hit ****/
20    if(cl->valid && cl->tag==tag){
21      cl->d[offset]=((cl->d[offset] & mask) | dat);
22      cl->dirty=1;
23    }
24    /**** D-Cache Miss ****/
25    else{  /** Store to the memory **/
26      ev->mem->st_8byte(&w->dc_wadr, &dat, mask);
27    }
28  }
```

図 21: データキャッシュユニット Dcache の Update フェーズ.

一部のロード命令では，ロードした値を符号拡張するといった付加的な操作が必要となる．35 行目から 37 行目では，この操作をおこなっている．

39 行目から 46 行目は，キャッシュにミスした場合の処理である．この場合には，参照するデータを命令キャッシュにロードする必要がある．これに先だって，キャッシュラインをメインメモリに書き戻し (43 行目)，44 行目の ld_line により，必要とするラインをロードする．

図 21 に，データキャッシュユニット Dcache の Update フェーズのコードを示す．ここでは，主に，ストア命令が完了した場合のデータキャッシュの更新処理をおこなう．

ロード命令と同様に，一部のストア命令では，ストアすべき値の符号拡張するといった付加的な操作が必要となる．5 行目から 9 行目では，この操作をおこなっている．

20 行目では，書き込むべきアドレスのラインがキャッシュに存在するかを判定し，そうであれば，キャッシュヒットとして，21 行目でデータの書き込みをおこなう．この時，キャッシュラインのデータが変更されたという意味で変数 dirty をセットする (22 行目).

キャッシュがミスの場合には，ライトノーアロケート方式なので，不在のラインをキャッシュに持ってくることはせずに，データをメインメモリにストアする (26 行目).

3.6 プロセッサの設定

RealScalar Version 1.0 は，様々なプロセッサのパラメータを変更できるように工夫されている．

個々の命令の演算レイテンシを設定するメソッド latency のコードを図 22 に示す．標準では，整数の乗算が 4 サイクル，浮動小数点演算は 5 サイクルに，それ以外は 1 サイクルとしている．演算レイテンシを変更するためには，適切な命令クラスや個々の命令毎に関数の返値を設定しればよい．

```
1  int scalar_chip::latency(char Op, int CI){
2    if(Op==OP_INTM) return 4;
3    if(Op==OP_FLTI) return 5;
4    return 1; /* default instruction latency */
5  }
```

図 22: 命令のレイテンシを設定するメソッド latency.

例えば AND 命令の命令レイテンシを 10 サイクルに変更するには，図 23 のように，2 行目のコードを追加すればよい．

```
1  int scalar_chip::latency(char Op, int CI){
2    if(CI==_AND___) return 10;
3    if(Op==OP_INTM) return 4;
4    if(Op==OP_FLTI) return 5;
5    return 1; /* default instruction latency */
6  }
```

図 23: AND 命令のみのレイテンシを 10 サイクルに設定.

図 24 に，クラス system_config_s の定義を示す．主なプロセッサのパラメータは，ここに示した system_config_s の変数に保存される．

```
 1  class system_config_s{
 2   public:
 3    int btb_entry;
 4    int icache_entry;
 5    int icache_penalty;
 6    int dcache_entry;
 7    int dcache_penalty;
 8    int branch_miss_penalty;
 9    int P_INFETCH; /* Stage:instruction fetch  */
10    int P_IDECODE; /* Stage:instruction decode */
11    int P_REGREAD; /* Stage:register read      */
12    int P_EXECUTE; /* Stage:exexute stage      */
13    int P_MEM_ACC; /* Stage:memory access      */
14    int P_WRITEBK; /* Stage:write back         */
15    int P_DEPTH;   /* pipeline depth */
16    system_config_s(char**);
17  };
```

図 24: クラス system_config_s の定義とコンストラクタ.

3.7 プロセッサの動作記述

これまでに，図 8 に示したメインループから呼び出される多くの関数に関して説明を加えてきた．メインループから呼び出される関数で，解説をおこなっていないもの

は Init, Step, IDecode, RegRead, Execute, WriteBack, Update, house_keeping の 8 個である．house_keeping を除く 7 個の関数を順番に見ていくことにする．

```
 1  void scalar_chip::Init(){
 2    w->pcgen_pc = pc_gen->pc;
 3
 4    w->stg[sc2->P_INFETCH] = Inull;
 5    for(int i=1; i<sc2->P_DEPTH; i++)
 6      w->stg[i]=stg[i-1];
 7
 8    instruction_s *p1 = w->stg[sc2->P_WRITEBK];
 9    w->dc_wir  = (p1->ST) ? p1->ir  : 0;
10    w->dc_wadr = (p1->ST) ? p1->Adr : 0;
11    w->dc_wdat = (p1->ST) ? p1->Rav : 0;
12
13    instruction_s *p2 = w->stg[sc2->P_MEM_ACC];
14    w->dc_rir  = (p2->LD) ? p2->ir  : 0;
15    w->dc_radr = (p2->LD) ? p2->Adr : 0;
16    w->dc_rdat = 0;
17  }
```

図 25: Init のコード.

図 25 に，Init のコードを示す．データキャッシュの入力として利用される幾つかの配線と，パイプラインレジスタからの出力を設定する．

```
 1  void scalar_chip::Step(){
 2    /***** data deps stall calculation *****/
 3    if(w->stg[sc2->P_REGREAD]==Inull ||
 4       w->stg[sc2->P_EXECUTE]==Inull){
 5      w->data_stall = 0;
 6    }
 7    else{
 8      instruction_s *pr = w->stg[sc2->P_REGREAD];
 9      instruction_s *pw = w->stg[sc2->P_EXECUTE];
10
11      /** dependency: RegRead after Load **/
12      int w_ld  = pw->LD && pw->WB!=31 &&
13        (pr->Ra == pw->Rc || pr->Rb == pw->Rc);
14
15      /** depencency: CMOV    after Load **/
16      int w_cm = pr->CM && pw->LD && pw->WB!=31 &&
17        pr->Rm == pw->Rc;
18
19      w->data_stall = w_ld | w_cm;
20    }
21  }
```

図 26: Step のコード.

図 26 に，Step のコードを示す．ここでは，データ依存関係によるストールと，メモリを介したストール，条件付き移動命令に関するストールの計算をおこなっている．

図 27 に，命令デコード IDecode のコードを示す．命令デコード IDecode では，SimCore/Alpha Functional Simulator の Slot(4 行目), Issue(5 行目) という関数を呼び出す．また，データ依存関係の確認を容易におこなうために，変数 Ra, Rb, Rc, Rm の値を計算する．

```
 1  void scalar_chip::IDecode(){
 2    instruction_s *pt = w->stg[sc2->P_IDECODE];
 3
 4    pt->Slot();
 5    pt->Issue();
 6
 7    /*** a part of decode for dependency check ***/
 8    pt->Ra =(pt->Ar << 5) |  pt->RA;
 9    pt->Rb =(pt->Br << 5) |  pt->RB;
10    pt->Rc =(pt->WB==31) ? 0xff:
11      (WR(pt->Op)<< 5) | pt->WB;
12    pt->Rm = (!pt->CM) ? 0xf0 :
13      (WR(pt->Op)<< 5) | pt->WB;
14  }
```

図 27: IDecode のコード.

```
 1  void scalar_chip::RegRead(){
 2    w->stg[sc2->P_REGREAD]->RegisterRead();
 3  }
```

図 28: RegRead のコード.

図 28 に，レジスタリード RegRead のコードを示す．レジスタリード RegRead では，SimCore/Alpha Functional Simulator の RegisterRead という関数を呼び出す．

図 29 に，実行ステージ Execute のコードを示す．実行ステージ Execute では，マルチサイクルの命令に対応するために，変数 busy を用いて演算器の状態を管理する．10 行目で演算器が動作中でないと判定された場合には，新しい命令の実行を開始する (11 行目から 19 行目)．そうでなければ，マルチサイクルの命令を実行しているので，命令レイテンシで指定されたサイクルを費やして処理を継続 (21 行目から 30 行目) する．

図 30 に，書き戻しステージ WriteBack のコードを示す．完了した命令が分岐命令だった場合 (4 行目) にこの情報を用いて分岐予測ユニット Bpred の情報を更新する．この時，分岐予測ユニットの入力の配線を 5 行目から 8 行目で設定する．完了した命令が PAL 命令だった場合には，データキャッシュの内容をフラッシュ(11 行目) し，幾つかの配線を設定 (18 行目から 21 行目) する．

図 31 に，Update のコードを示す．命令キャッシュがストールしていない時 (3 行目) には，適切な命令がフェッチされているので，フェッチした命令の情報を適切なレジスタに格納する (4 行目から 10 行目)．

メモリ参照のステージでロード命令が実行されている場合 (14 行目) には，データキャッシュの出力をレジスタ Rcv に格納する (15 行目)．

18 行目から 29 行目では，命令間のデータ依存関係を解析し，必要に応じてデータをフォワーディングする．

31 行目から 39 行目は，幾つかのフラグを参照して，パイプラインレジスタの更新の処理をおこなう．

```
 1  void scalar_chip::Execute(){
 2    static int busy=0; /* ALU is busy ? */
 3    w->code_stall = 0;
 4    w->flush_bpr  = 0;
 5    w->pcgen_bpr  = 0;
 6
 7    instruction_s *pt = w->stg[sc2->P_EXECUTE];
 8    if(pt==Inull) return;
 9
10    if(!busy){ /** fire new instruction  **/
11      insn_latency = latency(pt->Op, pt->CI);
12      if(insn_latency==1){
13        execute_one(pt, w);
14        busy=0;
15      }
16      else{
17        busy=1;
18        w->code_stall=1;
19      }
20    }
21    else{ /** executing multi-cycle inst. **/
22      if(insn_latency>1) insn_latency--;
23      if(insn_latency==1){
24        execute_one(pt, w);
25        busy=0;
26      }
27      else{
28        w->code_stall=1;
29      }
30    }
31  }
```

図 29: 実行ステージ Execute のコード.

```
 1  void scalar_chip::WriteBack(){
 2    instruction_s *p = w->stg[sc2->P_WRITEBK];
 3
 4    int br = (p->Op==OP_JSR || p->BR);
 5    w->bpred_wtkn = (br) &
 6      (p->Cpc+4 != p->Npc);
 7    w->bpred_wpc  = (br) ? p->Cpc : 0;
 8    w->bpred_wnpc = (br) ? p->Npc : 0;
 9
10    /***** Data Cache Flush by PAL*****/
11    if(p!=Inull && p->Op==OP_PAL) dcache->flush();
12
13    if(p!=Inull){
14      ev->as->pc = p->Npc;
15      p->WriteBack();  /** Write Back **/
16    }
17
18    int pal = (p!=Inull && p->Op==OP_PAL);
19    /* sigreturn updates as->ps */
20    w->pcgen_pal = (pal) ? ev->as->pc : 0;
21    w->flush_pal = (pal) ? 1 : 0;
22  }
```

図 30: 書き戻しステージ WriteBack のコード.

```
 1  void scalar_chip::Update(){
 2    /***** Set fetched code info. *****/
 3    if(!w->icache_stall){
 4      instruction_s *p = inst_buf->alloc();
 5      p->ir  = w->ic_out;
 6      p->Cpc = w->pcgen_pc;
 7      p->Npc = w->pcgen_npc;
 8      p->Ppc = (w->bpred_tkn) ? w->bpred_targ :
 9        w->pcgen_npc;
10      w->stg[sc2->P_INFETCH] = p;
11    }
12
13    /***** Set the Rcv from the data cache *****/
14    if(w->stg[sc2->P_MEM_ACC]->LD)
15      w->stg[sc2->P_MEM_ACC]->Rcv=w->dc_rdat;
16
17    /**** Data forwarding to Rav & Rbv & Adr ****/
18    if(w->stg[sc2->P_REGREAD]!=Inull){
19      instruction_s *pr = w->stg[sc2->P_REGREAD];
20      for(int i=sc2->P_WRITEBK;
21          i>=sc2->P_EXECUTE; i--){
22        instruction_s *pw = w->stg[i];
23        if(pw!=Inull){
24          if(pr->Ra==pw->Rc) pr->Rav=pw->Rcv;
25          if(pr->Rb==pw->Rc) pr->Rbv=pw->Rcv;
26          if(pr->Rm==pw->Rc) pr->Adr=pw->Rcv;
27        }
28      }
29    }
30
31    int hazard = w->flush_pal | w->dcache_stall |
32      w->flush_bpr | w->code_stall | w->data_stall;
33    if(!hazard){
34      for(int i=0; i<sc2->P_DEPTH; i++){
35        stg[i] = w->stg[i];
36      }
37    }
38    else {
39      UpdatePipeReg();
40    }
41  }
```

図 31: Update のコード.

4 おわりに

プロセッサアーキテクチャ研究やプロセッサ教育のためのツールとして様々なプロセッサシミュレータが利用されている．我々は，可読性が高く扱いやすいコードであることを第一の条件としてプロセッサシミュレータの構築をおこなっている．

本稿では，クロックレベルのプロセッサシミュレータである SimCore/Alpha RealScalar Simulator Version 1.0 (RealScalar Version 1.0) の設計と実装をまとめた．

RealScalar Version 1.0 が模倣するプロセッサは，6 段の命令パイプラインを持つシンプルなスカラープロセッサである．ダイレク・トマップ方式の命令キャッシュ，ダイレクト・マップ方式のデータキャッシュ，分岐予測機構を持つ．

RealScalar Version 1.0 の主な特徴は次の通り．(1) クロックレベルのスカラプロセッサシミュレータとして豊富な機能を提供する．(2)SimCore/Alpha Functional Simulator に追加する形で実装され，追加されるコード量は，C++で 1,215 行と少ない．

RealScalar Version 1.0 のソースコードは次の URL からダウンロードできる．

`http://www.yuba.is.uec.ac.jp/~kis/SimCore/`

参考文献

[1] Bob Cmelik and David Keppel. Shade: a fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pp. 128–137, 1994.

[2] Doug Burger and Todd M. Austin. The Simplescalar Tool Set, Version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin-Madison, June 1997.

[3] Kenji Kise, Hiroki Honda, and Toshitsugu Yuba. SimAlpha Version 1.0: Simple and Readable Alpha Processor Simulator. *Lecture Note in Computer Science (LNCS)*, Vol. 2823, pp. 122–136, September 2003.

[4] S. McFarling. Combining Branch Predictors. Technical report, WRL Technical Note TN-36, Digital Equipment Corporation, 1993.

[5] David A. Patterson and John L. Hennessy. コンピュータの構成と設計. 日経 BP 社, 1996.

[6] Eric Schnarr and James R. Larus. Fast out-of-order processor simulation using memoization. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pp. 283–294, 1998.

[7] Shubhendu S. Mukherjee, Sarita V. Adve, Todd Austin, Joel Emer, and Peter S. Magnusson. Performance Simulation Tools. *IEEE Computer*, Vol. 35, No. 2, pp. 38–39, February 2002.

[8] 吉瀬謙二, 片桐孝洋, 本多弘樹, 弓場敏嗣. Simcore/alpha functional simulator の設計と評価. 情報処理学会研究報告 2004-ARC-156, pp. 31–36, February 2004.

[9] 吉瀬謙二, 本多弘樹, 弓場敏嗣. SimAlpha: シンプルで理解しやすいコード記述を目指した Alpha プロセッサシミュレータ. Technical Report UEC-IS-2002-2, 電気通信大学 大学院情報システム学研究科, July 2002.

[10] 吉瀬謙二, 本多弘樹, 弓場敏嗣. SimAlpha: C++で記述したもうひとつの Alpha プロセッサシミュレータ. 情報処理学会研究報告 2002-ARC-149, pp. 163–168, August 2002.

[11] 中田尚, 大野和彦, 中島浩. 高性能マイクロプロセッサの高速シミュレーション. 先進的計算基盤システムシンポジウム SACSIS2003 論文集, pp. 89–96, May 2003.