卒業論文

# ハード/ソフト協調学習システムを用いた
# 割込みプロセッサの設計


氏　　名　：　PISHVA JOHN CYRUS P

学籍番号　：　2260060133-8

担当教員　：　山崎 勝弘 教授

提 出 日　：　2010 年 2 月 18 日

立命館大学　理工学部　電子情報デザイン学科

**内容概要**

本論文では、割込みの目的や原理を理解するとともに、ハード/ソフト協調学習システムを用いて割込みプロセッサを設計することで、ハードウェアとソフトウェアの両方の観点から知識を得ることを目的とし、Verilog HDL によるシングルサイクルの割込みプロセッサを設計した。設計したプロセッサは JINT（John INTerruption）と名づけた。また設計したプロセッサを HDL シミュレーションで検証、およびプロセッサデバッガと接続して FPGA ボード上で検証した。

本研究では、ハード/ソフト協調学習システムを用いて実際に割込みプロセッサを設計することで、割込みがどのように処理されるかをハードウェアとソフトウェアの両方の観点から理解し、今後の学習者が割込みの目的・原理を理解しやすくすることを目的とする。

# 目次

**図目次**

**表目次**

# 1　はじめに

近年の急速な半導体製造技術により、LSI の小型化、軽量化と高速化、そして低消費電力化が可能となった。携帯電話、自動車、カーナビゲーションシステム、炊飯器、信号機、エレベータ、自動販売機、デジタルカメラ、テレビ、ゲーム機、複写機などに挙げられる組み込み機器は、いずれもハードウェアとソフトウェアから構成されている。これら組み込み機器の普及は、これからも広がっていくことが確実視されている。そして要求される仕様は大規模かつ複雑・専用化され、実装には小型、低消費電力化が進み、製品のライフサイクルは縮小し、開発期間の短縮化が求められている。このように、高集積システム LSI 技術の進化の中、システム LSI へ求められる機能は多様化しており、ハードとソフト両方の知識に加え、プロセッサにおける命令セットとマイクロアーキテクチャの知識が必要不可欠である。

半導体製造技術の進歩によって、大規模で複雑なシステムが 1 つのチップ上に構成できるようになったこと、つまり LSI の設計と検証がシステム全体の設計と検証と等価になった。また、組み込み機器のライフサイクルが短くなり、開発期間を短くすることがますます重要となっていることにより、ハード/ソフト協調設計が開発期間短縮に大きく影響する。このような近年の LSI 開発技術はハードウェアとソフトウェアに密接な関係があり、両方の知識を習得するためにも、早期の教育が必要である。

以上の背景から、大学の教育でもハードウェアとソフトウェアの関係を意識した学習が必要である。そこで本研究室では、ハードウェアとソフトウェアを理解する上で重要な割込みの目的・役割と動作を理解し、ハード/ソフト協調学習システムを用いて実際に割込みプロセッサを設計し検証することを目的とする。ハード/ソフト協調学習システムとは、プロセッサを通じてハードとソフトの両方の学習を進めていくことを目的としたシステムである[2]。

本研究では、Verilog HDL によるシングルサイクルの割込みプロセッサの設計を行う。割込みは、外部ハードウェアの異常検知や、ソフトウェアの不正演算の防止など、ソフトウェア処理と周辺機器のハードウェアの連携を良くするとともに、性能向上・不正動作防止にもつながる。実際にプロセッサ設計を行うことにより、シングルサイクル割込みプロセッサのアーキテクチャを理解する。次に、設計したプロセッサを HDL シミュレータにより検証する。HDL シミュレータには、Xilinx 社の ModelSim を使用する。そして設計したプロセッサをプロセッサデバッガと接続し、論理合成を行い、FPGA ボード上に実装し検証する。論理合成には Xilinx 社の ISE 9.2i を使用している。以上の流れから、設計したプロセッサの評価と、さらにはハード/ソフト協調学習システムについての評価を行う。

本論文では、第 2 章でハードソフト協調学習システムについて詳細を説明する。第 3 章では割込み処理について説明し、第 4 章では割込み処理に対応した JINT プロセッサについて、第 5 章で JINT の設計と検証およびハードソフト協調学習システムの評価について述べる。

## 2　ハード/ソフト協調学習システム

### 2.1　　システム概要

ハード/ソフト協調学習システムとは、プロセッサを通じてハードウェアとソフトウェアの両方の知識を学習していくために考案されたシステムである。

ソフトウェアを学習する面では、アーキテクチャが可変な命令セットシミュレータ(MONI 仮想シミュレータ)を用いてプロセッサのアーキテクチャの仕組みを理解し、アセンブリ言語でかかれたプログラムを評価する。MONI とは、本研究室で MIPS のサブセットとして定義された教育マイクロプロセッサである。ハードウェアを学習する面では、シミュレータで理解したプロセッサの知識を基に、HDL によるプロセッサ設計を行う。そして学習者が設計したプロセッサを検証、評価することによってプロセッサ設計能力を習得する。次にハードウェア学習の際に使用するプロセッサ支援ツールについて説明する。命令セット定義ツール、汎用アセンブラ、汎用シミュレータにより、命令セットを独自に定義することができる。またプロセッサモニタとプロセッサデバッガは、学習者が設計したプロセッサを FPGA ボード上で検証する際に使用する。これらのプロセッサ設計支援ツールを使用し、ハードとソフトの両方の学習を円滑に進めていくことがこのシステムの目的である。

### 2.2　　学習体系

**図 1** にハード/ソフト協調学習システムの学習体系を示す。ソフトウェア学習の流れは、学習者自身が用意したアセンブリプログラムを MONI 仮想シミュレータ上でシミュレーションを行う。MONI 仮想シミュレータでは、アーキテクチャを単一サイクル、マルチサイクル、パイプライン、スーパースカラの 4 つが選択可能である。プログラムの命令を実行すると、その命令に対するプロセッサのデータパスが確認できるので、これにより学習者はアセンブリプログラム技術と MONI プロセッサの構造や動作の学習を行うことができる。次に、プロセッサ設計ツールの命令セット定義ツールを用いて、学習者の考えた命令セットを定義し、その出力ファイルを用いて汎用アセンブラと汎用シミュレータを使用する。また汎用アセンブラの出力ファイルは、ハードウェアの学習でのプロセッサ検証の際に使用する。学習者がこの 3 つのプロセッサ設計支援ツールを使用することにより、プロセッサにおける命令セットアーキテクチャを学習することができる。ハードウェア学習の流れは、実際に HDL を用いて MONI プロセッサの設計、またはオリジナルプロセッサの設計を行う。次に設計したプロセッサを HDL シミュレータによりシミュレーション検証を行い、それから FPGA ボード上に実装し、評価する。FPGA ボード上で検証する際、設計したプロセッサをプロセッサデバッガと接続し、プロセッサモニタを用いてデータを送受信することで検証する。

このようにしてプロセッサ設計能力の習得、またハードウェア特有の性質である遅延や設計規模を考慮したプロセッサの手法を学習することができる。以上の流れから、ソフト

ウェアとハードウェアの学習を行う。

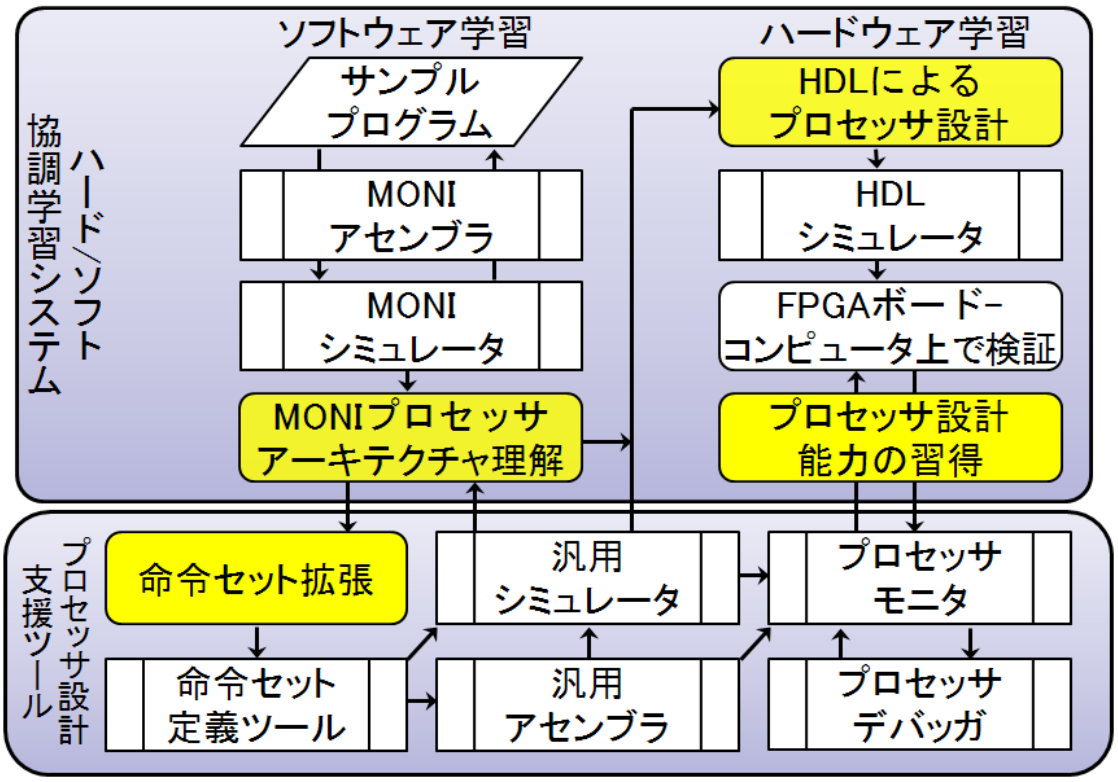


図 1　ハード/ソフト協調学習システムの学習体系

# 3　割込み処理

## 3.1　　割込み処理の目的

割込み処理とは、プロセッサが通常のソフトウェア処理中に割込み信号を受けることによって、通常処理を中断し、割込み処理に即座に切り替えることが可能となる。通常処理内で割込み信号の有無を常に確認する必要がなく、信号に対して確実に反応できるため、性能と応答性の向上にもつながるだけでなく、周辺機器などの異常を割込み信号で検知することで例外処理を容易に行うことができる。また、割込みはソフトウェアの不正演算の検知やトレース・デバッグにも用いられており、ソフトウェア開発を進める上で役に立つ。

## 3.2　　ハードウェア割込みとソフトウェア割込み

プロセッサの割込みには大きく分けて、外部で発生する「ハードウェア割込み」と、内部で発生する「ソフトウェア割込み」の 2 種類がある。一般的な割込みの種類と目的を**表 1** に示す。

**表 1　ハードウェア割込みとソフトウェア割込み**

| | 種　類 | 目　的 |
|---|---|---|
| **ハードウェア割込み** | 入出力 | 機器からの入出力の状況通知 |
| | タイマー | 時間の計測や定期処理 |
| | リセット | 意図的に処理を初期化するとき |
| | ハードウェア故障 | ハードウェア故障の検知 |
| **ソフトウェア割込み** | オーバーフロー | オーバーフローの検知・修復 |
| | ゼロ除算 | ゼロ除算の検知 |
| | メモリアクセス失敗 | 主記憶装置からデータを取込むよう指示する |
| | トレース・デバッグ | ソフトウェアのバグ検知 |

ハードウェア割込みはソフトウェア処理のどのタイミングで割込み信号が入ってくるかわからないため、一般的にマスク方式で受け付ける割込みをソフトウェアで選択する。

### 3.3　　割込みの実現条件

処理を実現させるには以下の３つの条件が重要である。

- 応答性
  割込み信号が発生したらすぐに通常処理を中断し、割込み処理に入らなければならない。割込み処理のプログラムアドレスを事前に準備しておく必要がある。
- 再開と停止
  割込み処理後に中断した通常処理を再開すること、または再開せず停止することである。再開するには割込み発生前にどこを処理していたかを保持する必要がある。
- データの退避と復旧
  通常処理で使用していたレジスタ値などの意図しない変化があってはいけないことである。割込み処理でレジスタ値を別の領域に移動することで退避し、処理後に戻すことによって退避と復旧を行う。

これらの条件を考慮し、第 4 章では割込み処理を実現するためのプロセッサ設計を説明する。

## 4　割込みプロセッサ JINT

### 4.1　　設計思想

本研究では、割込みの目的と仕組みの理解と設計、割込みを用いたソフトウェア設計と検証、およびハード/ソフト協調学習システムがプロセッサ設計において有効であるかを評価するためにシングルサイクルの割込みプロセッサを設計した。設計したプロセッサは MONI プロセッサの命令セットを参考にした独自プロセッサで、JINT(John INTerruption)と名付けた。JINT プロセッサには以下のような特徴がある。

- 16 ビット固定命令長
- 3 オペランド命令方式
- 全 45 命令
- 5 つの命令形式
- 3 種類の割込みに対応
- ソフトウェアによる割込みソースの選択

### 4.2　　対応する割込みの種類

設計する JINT プロセッサは、代表的な 3 つの割込みに対応する。ハードウェア割込みとして、リセットとタイマー、ソフトウェア割込みとしてオーバーフローに対応した。ハードウェア割込みのリセットは、ソフトウェアの設計次第で入出力の割込みとしても利用

することができる。一般的な割込みに対応するプロセッサでは、マスク方式により同時に複数の割込みソースに対応するが､アーキテクチャが複雑になるため JINT プロセッサでは単一の割込みのみに対応する。

リセット割込みは、物理的なスイッチの入力などで信号が立ち上がるときに割込み信号が発生する。タイマー割込みは、プロセッサ内に割込み用カウントダウンモジュールを設計した。クロック毎にカウンターの値がデクリメントされ、値が 1 の時に割込み信号が発生する。オーバーフロー割込みは、特定の ALU 演算において、オーバーフローの状況を割込み信号とする。

### 4.3 アーキテクチャ

#### 4.3.1 命令セット

JINT プロセッサには、Register 形式(R 形式)、Immediate5(I5 形式)、Immediate8(I8 形式)、Jump 形式(J 形式)、Interruption 形式(INTR 形式)の 5 つの命令形式を用意した。R 形式にはレジスタ間の演算を行う命令を定義している。I5 形式にはレジスタ値と即値演算を行う命令を定義している。I8 形式には条件分岐命令とメモリ・レジスタへのデータ転送、サブルーチンジャンプ命令を定義している。J 形式命令には不条件分岐や空白命令の NOP、プログラム停止命令となる HALT、割込み時にジャンプするジャンプ先アドレスの指定、割込み処理から通常処理へ戻る割込みリターン命令を定義している。そして、INTR 形式は、割込みを制御するための命令で使用される。INTR 形式は命令によって割り込みフィールド内の組み合わせが変化する。詳しくは 4.3.2 にて述べる。**表 2** に JINT の命令形式、**表 3** に各フィールドの用途、**表 4** に JINT 命令セット一覧を示す。

**表 2 JINT の命令形式**

| bit / format | 5 | 3 | 3 | 3 | 2 |
|---|---|---|---|---|---|
| **R** | OPCODE | $R_{OUT}$ | $R_{IN0}$ | $R_{IN1}$ | FN |
| **I5** | OPCODE | $R_{OUT}$ | $R_{IN0}$ | Immediate | |
| **I8** | OPCODE | $R_{IN0}$ $R_{OUT}$ | Immediate | | |
| **J** | OPCODE | Immediate | | | |
| **INTR** | OPCODE | *割込みフィールド…* | | | |

表 3　命令フィールドの意味

| フィールド | 意 味 | bit 幅 | 用 途 |
|---|---|---|---|
| **OPCODE** | Operation Code | 5 | 命令を識別 |
| **$R_{OUT}$** | Register Out | 3 | 演算結果を格納するレジスタアドレス |
| **$R_{IN0}$** | Register In 1 | 3 | 演算元レジスタアドレス 1 |
| **$R_{IN1}$** | Register In 2 | 3 | 演算元レジスタアドレス 2 |
| **FN** | Function | 2 | R 形式・INTR 形式の命令を詳細に識別 |
| **Immediate** | Immediate | 5~11 | 即値入力 |

表 4　JINT 命令セット一覧

| 命令 | 形式 | OPCODE | FN | 命令動作 | 意味 |
|---|---|---|---|---|---|
| **ADD** | R | 00000 | 00 | ADD $R_{OUT}$ $R_{IN0}$ $R_{IN1}$ | $R_{OUT}$ = $R_{IN0}$ + $R_{IN1}$ |
| **SUB** | R | 00000 | 01 | SUB $R_{OUT}$ $R_{IN0}$ $R_{IN1}$ | $R_{OUT}$ = $R_{IN0}$ - $R_{IN1}$ |
| **AND** | R | 00000 | 10 | AND $R_{OUT}$ $R_{IN0}$ $R_{IN1}$ | $R_{OUT}$ = $R_{IN0}$ & $R_{IN1}$ |
| **OR** | R | 00000 | 11 | OR $R_{OUT}$ $R_{IN0}$ $R_{IN1}$ | $R_{OUT}$ = $R_{IN0}$ \| $R_{IN1}$ |
| **XOR** | R | 00001 | 00 | XOR $R_{OUT}$ $R_{IN0}$ $R_{IN1}$ | $R_{OUT}$ = $R_{IN0}$ ^ $R_{IN1}$ |
| **NOT** | R | 00010 | 11 | NOT $R_{OUT}$ $R_{IN0}$ | $R_{OUT}$ = ~$R_{IN0}$ |
| **SEQ** | R | 00010 | 01 | SEQ $R_{OUT}$ $R_{IN0}$ $R_{IN1}$ | $R_{OUT}$ = ($R_{IN0}$ == $R_{IN1}$) ? 1 : 0 |
| **SNE** | R | 00010 | 10 | SNE $R_{OUT}$ $R_{IN0}$ $R_{IN1}$ | $R_{OUT}$ = ($R_{IN0}$ != $R_{IN1}$) ? 1 : 0 |
| **SLT** | R | 00001 | 01 | SLT $R_{OUT}$ $R_{IN0}$ $R_{IN1}$ | $R_{OUT}$ = ($R_{IN0}$ < $R_{IN1}$) ? 1 : 0 |
| **SGT** | R | 00001 | 10 | SGT $R_{OUT}$ $R_{IN0}$ $R_{IN1}$ | $R_{OUT}$ = ($R_{IN0}$ > $R_{IN1}$) ? 1 : 0 |
| **SLE** | R | 00001 | 11 | SLE $R_{OUT}$ $R_{IN0}$ $R_{IN1}$ | $R_{OUT}$ = ($R_{IN0}$ <= $R_{IN1}$) ? 1 : 0 |
| **SGE** | R | 00010 | 00 | SGE $R_{OUT}$ $R_{IN0}$ $R_{IN1}$ | $R_{OUT}$ = ($R_{IN0}$ >= $R_{IN1}$) ? 1 : 0 |
| **SLL** | R | 00011 | 00 | SLL $R_{OUT}$ $R_{IN0}$ $R_{IN1}$ | $R_{OUT}$ = $R_{IN0}$ << $R_{IN1}$ |
| **SRL** | R | 00011 | 10 | SRL $R_{OUT}$ $R_{IN0}$ $R_{IN1}$ | $R_{OUT}$ = $R_{IN0}$ >> $R_{IN1}$ |
| **SRA** | R | 00011 | 11 | SRA $R_{OUT}$ $R_{IN0}$ $R_{IN1}$ | $R_{OUT}$ = $R_{IN0}$ >>> $R_{IN1}$ |
| **ADDI** | I5 | 00100 | - | ADDI $R_{OUT}$ $R_{IN0}$ I5 | $R_{OUT}$ = $R_{IN0}$ + I5 |
| **SUBI** | I5 | 00101 | - | SUBI $R_{OUT}$ $R_{IN0}$ I5 | $R_{OUT}$ = $R_{IN0}$ - I5 |
| **ANDI** | I5 | 00110 | - | ANDI $R_{OUT}$ $R_{IN0}$ I5 | $R_{OUT}$ = $R_{IN0}$ & I5 |
| **ORI** | I5 | 00111 | - | ORI $R_{OUT}$ $R_{IN0}$ I5 | $R_{OUT}$ = $R_{IN0}$ \| I5 |
| **XORI** | I5 | 01000 | - | XORI $R_{OUT}$ $R_{IN0}$ I5 | $R_{OUT}$ = $R_{IN0}$ ^ I5 |
| **SEQI** | I5 | 10000 | - | SEQI $R_{OUT}$ $R_{IN0}$ I5 | $R_{OUT}$ = ($R_{IN0}$ == I5) ? 1 : 0 |
| **SNEI** | I5 | 10001 | - | SNEI $R_{OUT}$ $R_{IN0}$ I5 | $R_{OUT}$ = ($R_{IN0}$ != I5) ? 1 : 0 |
| **SLTI** | I5 | 01100 | - | SLTI $R_{OUT}$ $R_{IN0}$ I5 | $R_{OUT}$ = ($R_{IN0}$ < I5) ? 1 : 0 |
| **SGTI** | I5 | 01101 | - | SGTI $R_{OUT}$ $R_{IN0}$ I5 | $R_{OUT}$ = ($R_{IN0}$ > I5) ? 1 : 0 |
| **SLLI** | I5 | 01001 | - | SLLI $R_{OUT}$ $R_{IN0}$ I5 | $R_{OUT}$ = $R_{IN0}$ << I5 |
| **SRLI** | I5 | 01010 | - | SRLI $R_{OUT}$ $R_{IN0}$ I5 | $R_{OUT}$ = $R_{IN0}$ >> I5 |
| **SRAI** | I5 | 01011 | - | SRAI $R_{OUT}$ $R_{IN0}$ I5 | $R_{OUT}$ = $R_{IN0}$ >>> I5 |
| **LDHI** | I8 | 10110 | - | LDHI $R_{OUT}$ I8 | $R_{OUT}$ = {I8, $R_{IN0}$[7:0]} |
| **LDLI** | I8 | 10111 | - | LDLI $R_{OUT}$ I8 | $R_{OUT}$ = {$R_{IN0}$[15:8], I8} |
| **BEQZ** | I8 | 10010 | - | BEQZ $R_{IN0}$ I8 | If (!$R_{IN0}$) PC=PC+1+I8 |
| **BNEZ** | I8 | 10011 | - | BNEZ $R_{IN0}$ I8 | If ($R_{IN0}$) PC=PC+1+I8 |
| **JAL** | I8 | 11010 | - | JAL $R_{OUT}$ I8 | $R_{OUT}$ =PC; PC=PC+1+I8 |
| **JR** | I8 | 11011 | - | JR $R_{IN0}$ | PC = $R_{IN0}$ |
| **JUMP** | J | 11100 | - | JUMP I11 | PC= I11 |
| **NOP** | J | 11110 | - | NOP | PC++ |
| **HALT** | J | 11111 | - | HALT | PC=PC |
| **LD** | I8 | 10100 | - | LD $R_{OUT}$ I8 | $R_{OUT}$ = MEM[I8] |
| **ST** | I8 | 10101 | - | ST $R_{IN0}$ I8 | MEM[I8] = $R_{IN0}$ |
| **IMD** | INTR | 01110 | 00 | IMD I2 | INTR_MODE = I2 |
| **IRB** | INTR | 01110 | 01 | IRB I1 | REGBANK = I1 |
| **IST** | INTR | 01110 | 10 | IST $R_{IN0}$ | TIMER = $R_{IN0}$ |
| **IJA** | INTR | 11000 | - | IJA I11 | INTR_JA = I11 |
| **IRE** | INTR | 11001 | - | IRE | PC = INTR_BA |
| **ISOF** | INTR | 00011 | 11 | ISOF $R_{IN0}$ | $R_{OF}$ = $R_{IN0}$ |
| **ISOFI** | INTR | 01111 | - | ISOFI I5 | $R_{OF}$ = I5 |

### 4.3.2　割込み命令

割込みを実現するために専用のレジスタをいくつか設計した。割込み用レジスタの意味を**表 5** に示す。

**表 5　割込み用レジスタ**

| フィールド | 使用命令 | 意味 |
|---|---|---|
| **INTR_MODE** | IMD | 選択した割込みソースを保持する |
| **REGBANK** | IRB | 使用するレジスタを切換える |
| **TIMER** | IST | 割込み用タイマーのカウント値 |
| **INTR_JA** | IJA | 割込み処理のジャンプ先アドレスを保持する |
| **INTR_BA** | IRE | 割込み処理からもどる通常処理のアドレスを保持する |
| **ROF** | ISOF,<br>ISOFI | Register OverFlowの略。オーバーフローした結果が代入されたレジスタアドレスを保持する |

また、それらのレジスタを制御するために独自の命令を定義した。それぞれの命令の目的と割込みフィールドについて述べる。

(1) IMD … Interruption MoDe

割込みソースを選択するために用いられる。選択した割込みソースは INTR_MODE レジスタで保持される。割込みソースは**表 6** の割り込みモード表から選択する。

**表 6　割込みモード表**

| モード | 番号 | 発生条件 |
|---|---|---|
| **無し** | 0 | 常に発生しない |
| **タイマー** | 1 | 割込み用カウントダウンタイマーが 1 の時に発生 |
| **オーバーフロー** | 2 | ALU 演算でオーバーフローがある場合に発生 |
| **リセット** | 3 | 外部からの信号入力で信号が HIGH の時に発生 |

IMD 命令の割込みフィールドを**表 7** に示す。0~1 の 2 ビットは FN フィールドである。**表 6** から選択した番号は 2~3 ビット目に入力しレジスタに書込まれる。

**表 7　IMD の割込みフィールド**

| bit No. | 10-4 | 3-2 | 1 | 0 |
|---|---|---|---|---|
| **filed** | - | INTR_MODE | 0 | 0 |

(2) IRB … Interruption RegisterBank

使用するレジスタを切換えるときに用いられる。命令からレジスタを参照する際、IRB で指定する REGBANK の 1 ビットと命令の 3 ビットの合計 4 ビットのアドレスでレジスタが参照される。IRB 命令の割込みフィールドを**表 8** に示す。0~1 ビット目は FN フィールドで、2 ビット目に REGBANK レジスタに代入する 1 ビットの値を入力する。

**表 8　IRB の割込みフィールド**

| bit No. | 10-3 | 2 | 1 | 0 |
|---|---|---|---|---|
| filed | - | REGBANK | 0 | 1 |

(3) IST … Interruption Set Timer

割込み用カウントダウンタイマーに値をセットするために用いられる。タイマー用レジスタ TIMER に値をセットするとクロック毎にデクリメントされ、TIMER 値が 1 のときにタイマーの割込み信号が発生する。IST 命令の割込みフィールドを**表 9** に示す。0~1 ビット目は FN フィールドで、7~10 の 3 ビットで指示されたレジスタ値を TIMER レジスタに代入する。

**表 9　IST の割込みフィールド**

| bit No. | 10-7 | 6-2 | 1 | 0 |
|---|---|---|---|---|
| filed | $R_{IN0}$ | - | 1 | 0 |

(4) IJA … Interruption Jump Address

割込み処理のジャンプ先アドレスをセットするために用いられる。割込みフィールドには絶対アドレスを 11 ビットで入力し、INTR_JA レジスタに代入される。割込みが発生した場合、PC にはこの値が代入され、割込み処理にジャンプする。

**表 10　IJA の割込みフィールド**

| bit No. | 10-0 |
|---|---|
| filed | Immediate |

(5) IRE … Interruption REturn

割込み処理から通常処理に戻る際に用いられる。IRE には引数は無く、割込みフィールドはすべて使用しない。通常処理のアドレスが保持される INTR_BA は、割込み信号に対して非同期に書込まれる。

(6) ISOF, ISOFI … Interruption Set OverFlow (Immediate)

演算のオーバーフローで発生した割込みで、オーバーフローした不正な結果に値を代入するために用いられる命令である。割込み信号が入力されると、その時の $R_{OUT}$ フィールドのレジスタ番地が $R_{OF}$ レジスタに代入される。ISOF 命令の割込みフィールドを**表 11** に示す。0~1 ビット目は FN フィールドで 5~7 ビット目で指定されたレジスタ値を $R_{OF}$ の指すレジスタに代入する。

**表 11　ISOF の割込みフィールド**

| bit No. | 10-7 | 7-5 | 4-2 | 1 | 0 |
|---|---|---|---|---|---|
| filed | - | $R_{IN0}$ | - | 1 | 1 |

そして ISOFI 命令の割込みフィールドを**表 12** に示す。0~7 ビット目で指定された I8 形式の即値を $R_{OF}$ の指すレジスタに代入する。

**表 12　ISOFI の割込みフィールド**

| bit No. | 10-8 | 7-0 |
|---|---|---|
| filed | - | Immediate |

### 4.3.3　JINT プロセッサアーキテクチャ

全体のアーキテクチャを**図 2** に示す。JINT プロセッサは赤色で示されたシングルサイクルのプロセッサモジュールとして 11 個、割込み実現のために加えた緑色で示された 9 個のモジュール、そして命令メモリとデータメモリの全部で 22 個からなる。命令メモリとデータメモリはハードソフト協調学習システムのプロセッサデバッガから提供される。なお、可読性の低下につながるので細かな制御線を省略している。

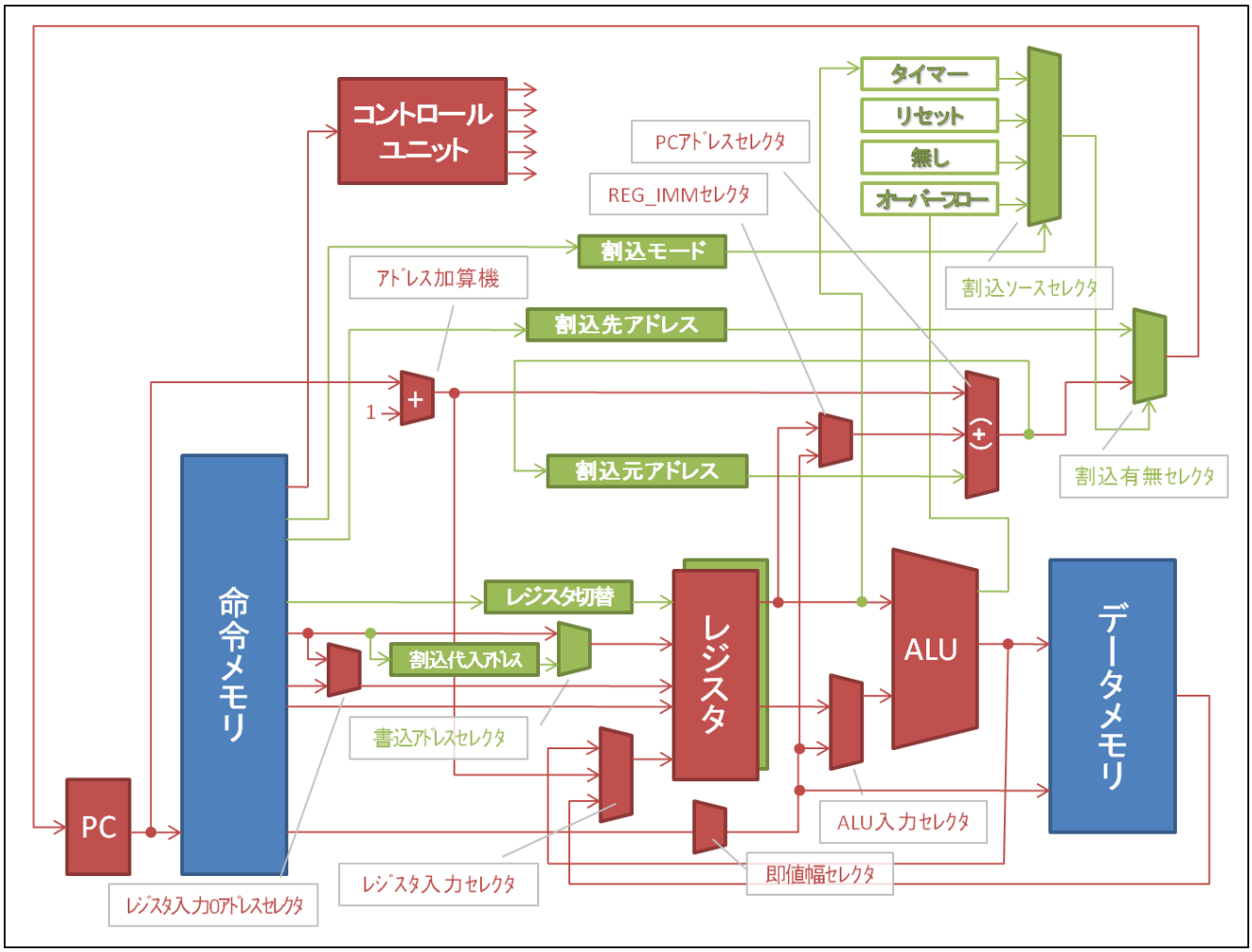


図 2　JINT プロセッサのアーキテクチャ

#### 4.3.4　割込み処理の実現のためのハードウェア設計

3.3 で述べた割込み処理の実現のために設計したハードウェアを述べる。まずは、応答性のために設計したモジュールを説明する。割込み信号に対してすぐに割込み処理に入るために、割込み処先アドレスレジスタを用意し、ジャンプ先アドレスを事前に設定するようにした。そして割込み信号の有無で次の PC の値を、通常処理か割込み処理のジャンプ先アドレスかを選択するセレクタを設計した(図 3)。これにより 1 サイクルで素早く割込み処理にジャンプすることができる。割込みソースの”無し”は常に LOW が出力され、これを選択していると割込みは発生しない。

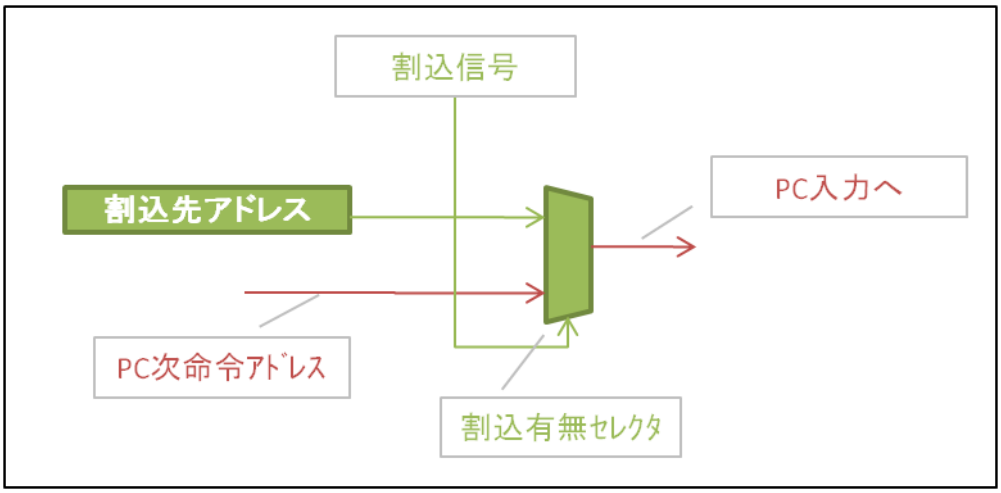


**図 3　応答性の実現**

次に再開と停止のためのハードウェアを述べる。割込み処理後に通常処理の直前まで処理していたところに復帰する必要がある。そのために、その次に処理されるべきアドレスを割込み元アドレスレジスタに割込み信号の立ち上がりで非同期に書込みを行う設計を行った（**図 4**)。なお、停止については停止命令(HALT)で行うため、特別な設計は不要である。

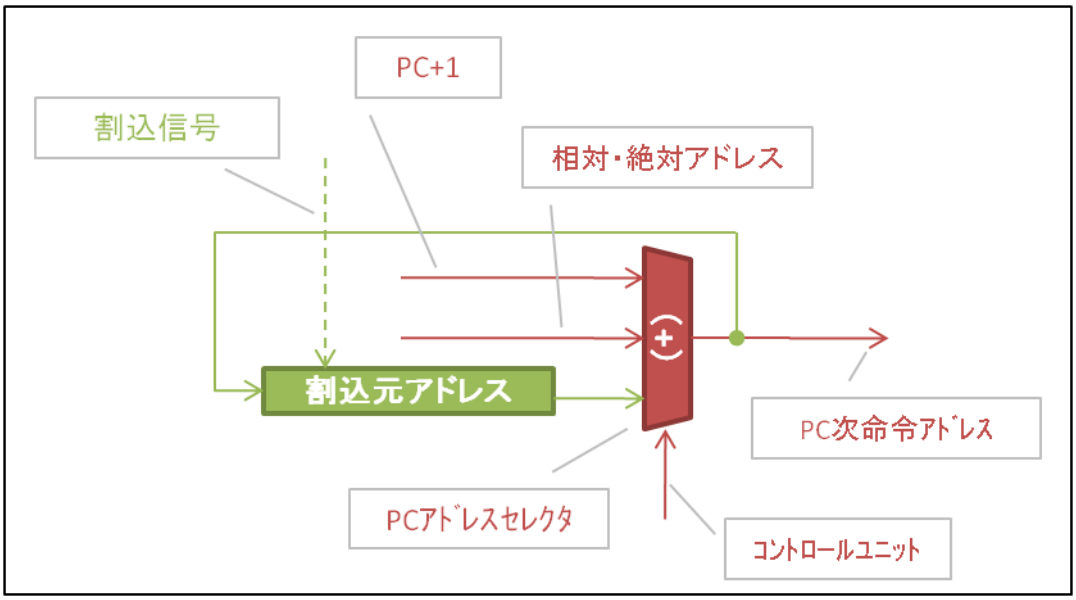


**図 4　再開と停止の実現**

次に、データの退避と復旧のためのハードウェアを述べる。データの退避と復旧の方法としてレジスタ容量を大きくした。レジスタのアドレッシングには命令の 3 ビット+レジスタ切換の 1 ビットの合計 4 ビットで行われる。通常処理ではレジスタ切換を OFF の状態で使用し割込み処理でレジスタ切換を ON にし、割込み処理後に再び OFF に戻すことで、あたかもレジスタが 2 つあり、切換えることで退避と復旧を行うことができる(図 5)。この方法で 1 命令実行することでレジスタの切換えを行うことができるため、スタックを用いたデータの退避と復旧に比べ高速に行うことができる。

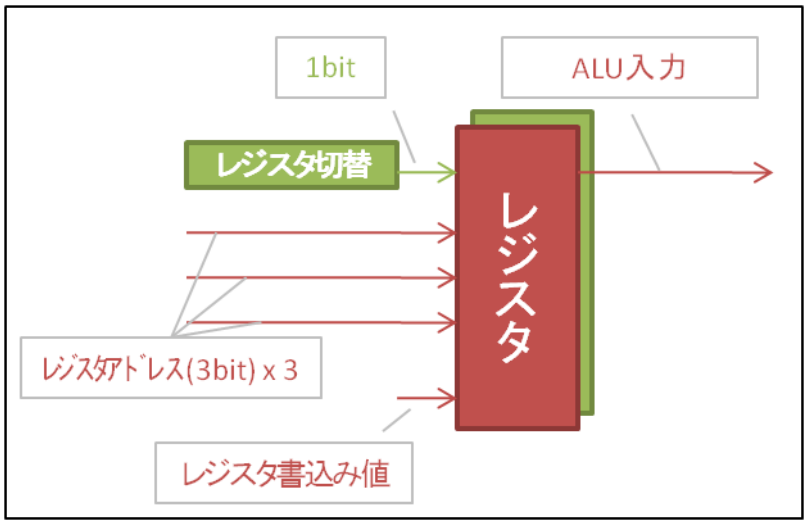


図 5　データの退避と復旧の実現

### 4.3.5　データパス

実行される命令形式、割込み制御命令、および割込み発生時と通常処理への復帰のデータパスを述べる。すべての命令は、PC の値の指すアドレスから取り出した命令から、コントロールユニットに OPCODE フィールドと FN フィールドを渡す。コントロールユニットはそれぞれのフィールドからデータパスを選択することでデータパスが変化する。

(1) R 形式

R 形式の命令のデータパスを**図 6** に示す。R 形式は、ADD や SUB など、レジスタ値＋レジスタ値の演算を行い、結果をレジスタに代入する命令形式である。まず、命令から $R_{IN0}$、$R_{IN1}$ の 2 つのフィールドのアドレス値からそれぞれの指すレジスタ値を取り出す。次に FN フィールドで ALU の動作を選択し、ALU の演算結果を $R_{OUT}$ の指すレジスタに代入する。PC はアドレス加算機によって 1 加算され、次のクロックで次の命令が実行される。

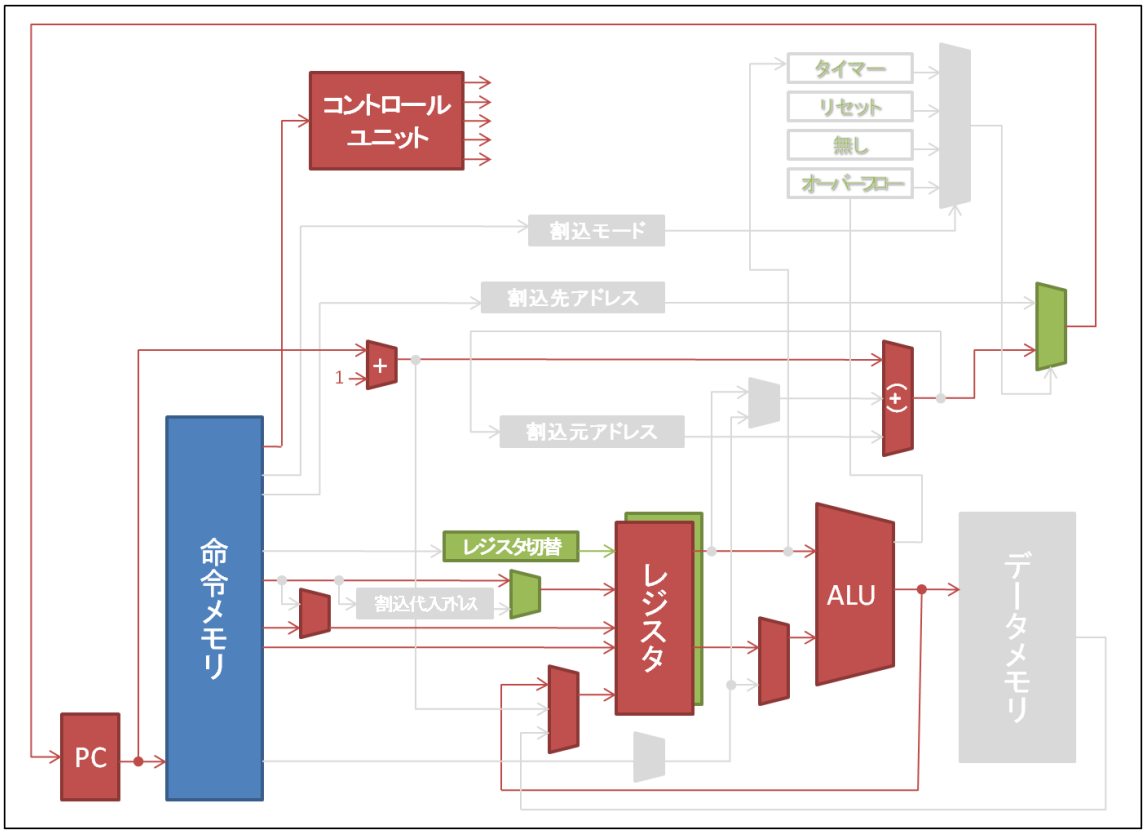


**図 6　R 形式のデータパス**

(2) I5 形式

I5 形式の命令が実行される過程を図 7 に示す。I5 形式は、ADDI や SUBI など、レジスタ値と即値 5 ビットの演算を行い、結果をレジスタに代入する命令形式である。まず、$R_{IN0}$ の指すレジスタ値と Immediate フィールドから即値幅セレクタによって取り出した下位 5 ビットの値を用いて ALU で演算を行う。ALU は OPCODE フィールドのみで動作を選択する。そして、演算結果を $R_{OUT}$ の指すレジスタに代入する。PC は R 形式と同様である。

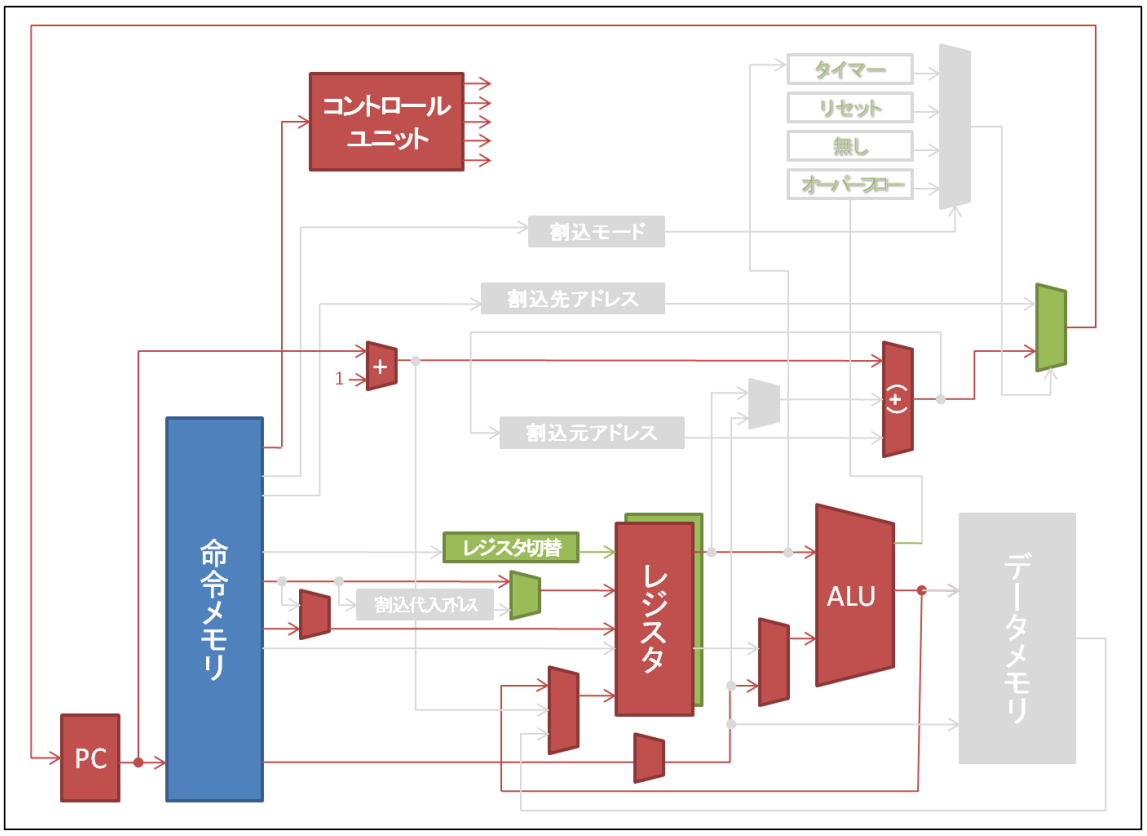


図 7　I5 形式のデータパス

(3) I8 形式

I8 形式の命令が実行される過程を述べる。I8 形式の命令には 8 ビット即値演算と条件分岐、サブルーチンジャンプ、データメモリのロード・ストアの 4 種類にデータパスが変化する。

(a) 8 ビット即値演算

8 ビット即値演算のデータパスを図 8 に示す。これは LDHI と LDLI の 2 つの命令で使用される。これは I5 形式によく似たデータパスになっているが、$R_{IN0}$ と $R_{OUT}$ が同じフィールドから参照されていることに注意してほしい。即値は即値幅セレクタにより下位 8 ビットが取り出される。そして ALU で演算を行い $R_{OUT}$ のアドレスの指すレジスタに結果が代入される。

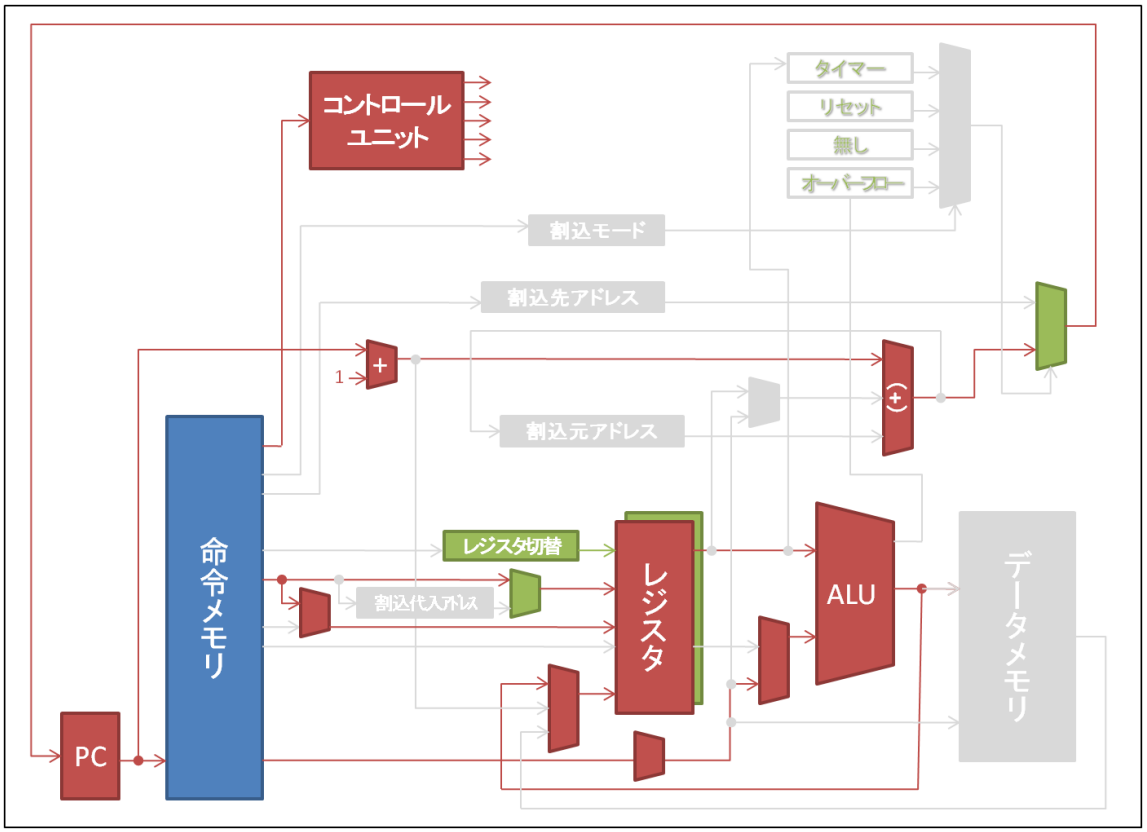


図 8　I8 形式、8 ビット即値演算のデータパス

(b) 条件分岐

条件分岐のデータパスを図 9 に示す。条件分岐は BEQZ と BNEZ の 2 つの命令で使用される。$R_{IN0}$ の指すレジスタ値について ALU で評価し、相対ジャンプを行うかを PC アドレスセレクタに入力する。PC アドレスセレクタには PC+1 の入力と即値 8 ビットの相対アドレスが入力され、判定結果判定結果によって、PC+1 のアドレスか、PC+1+相対アドレスを選択することで条件分岐を行う。

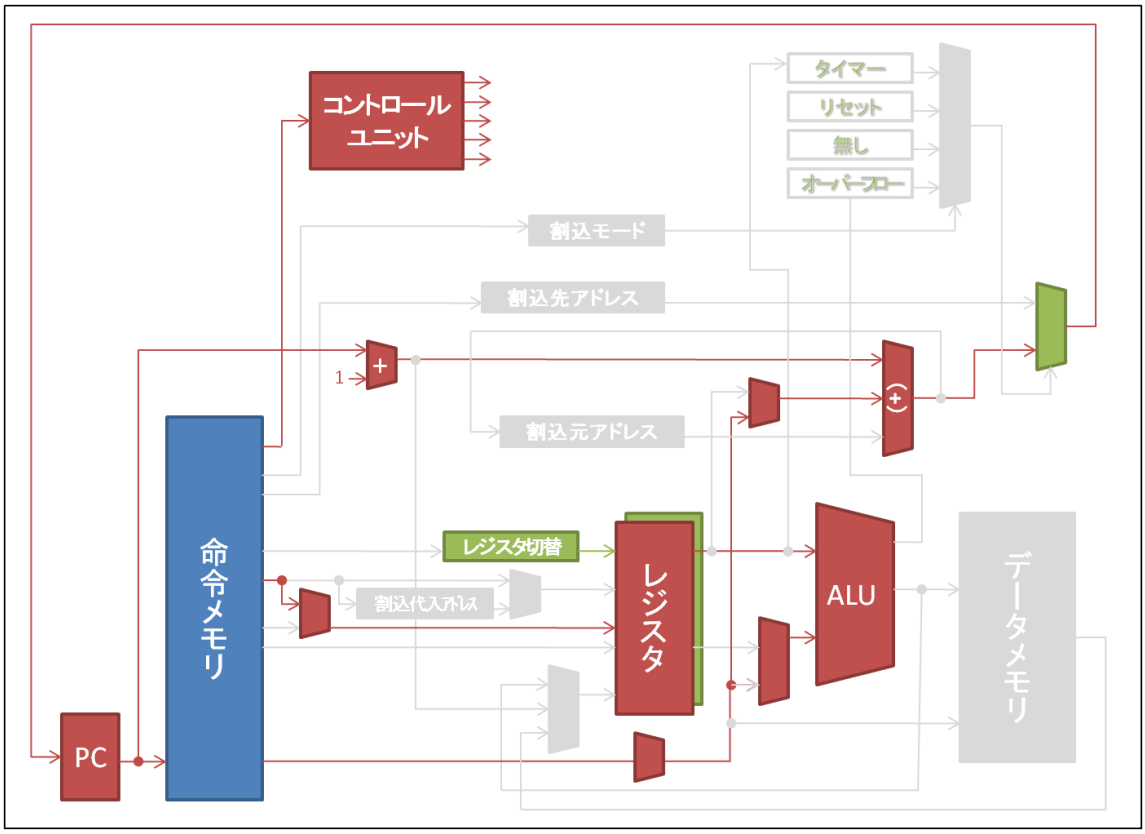


図 9　I8 形式、条件分岐のデータパス

(c) サブルーチンジャンプ

サブルーチンジャンプのデータパスを図 10 に示す。サブルーチンジャンプは JAL と JR の 2 つの命令で使用される。JAL 命令では $R_{OUT}$ の指すレジスタに PC+1 の絶対アドレスを代入し、即値 8 ビットの指す相対アドレスにジャンプする。JR 命令では $R_{IN0}$ の指すレジスタ値に絶対ジャンプを行う。これにより、JAL 命令～JR 命令でサブルーチンジャンプを行うことが可能である。

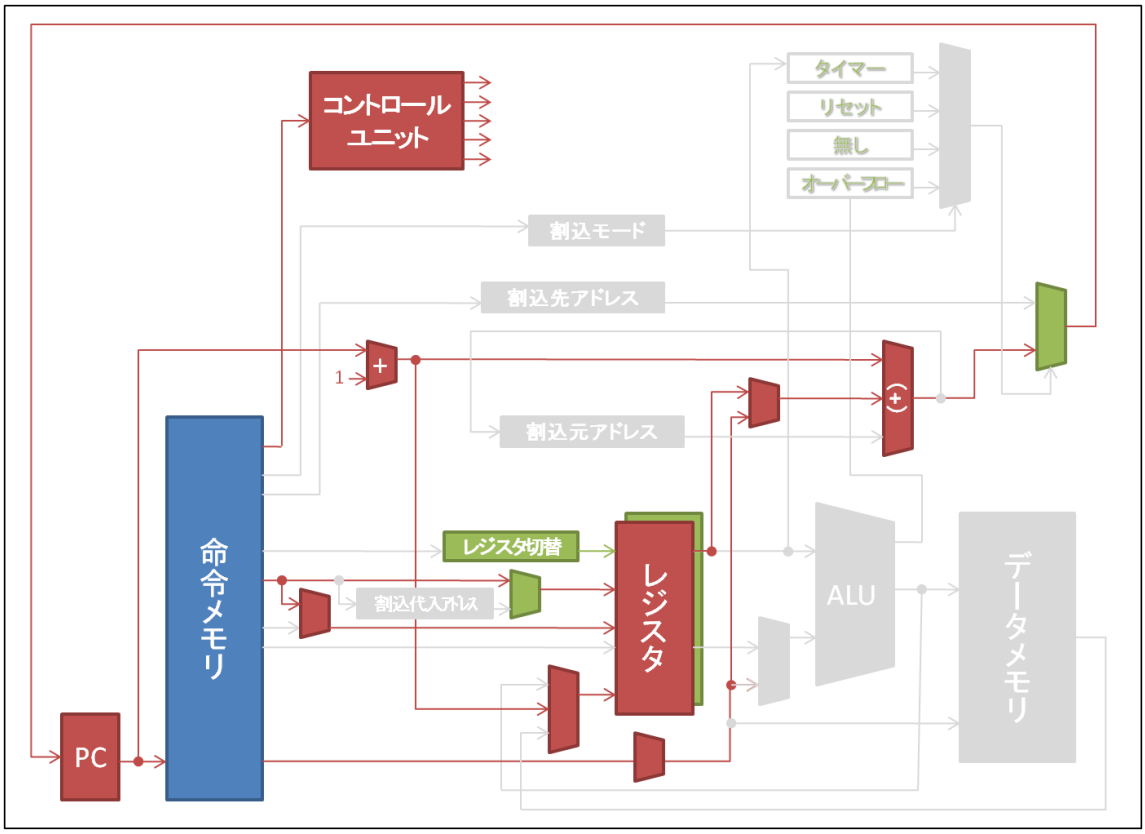


図 10　I8 形式、サブルーチンジャンプのデータパス

(d) データメモリのロード・ストア

データメモリのロード・ストアのデータパスを図 11 に示す。これは LD と ST の 2 つ命令で使用される。即値 8 ビットは読み書きするデータメモリのアドレス指定に使用される。LD 命令でデータメモリから値が読出され、$R_{IN0}$ の指すレジスタに代入される。このとき ALU は使用されない。ST 命令で $R_{OUT}$ の指すレジスタの値がデータメモリに書込まれる。

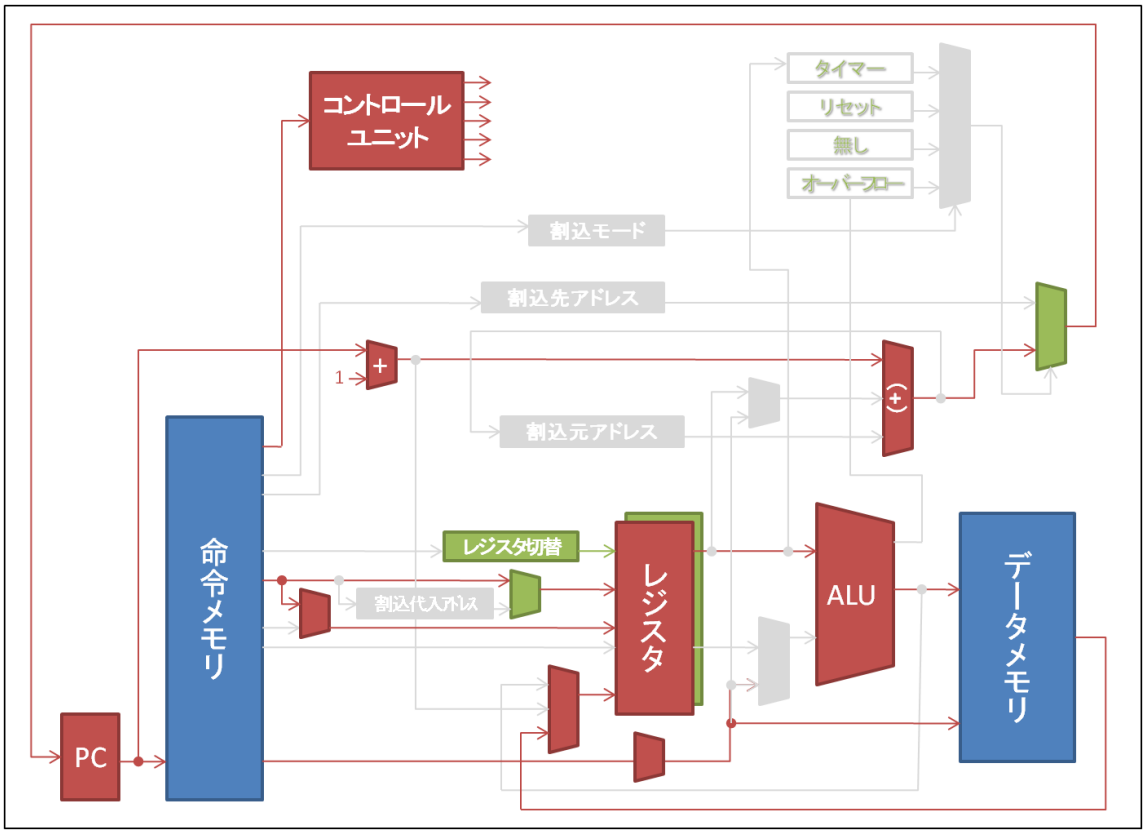


図 11　I8 形式、データメモリのロード・ストアのデータパス

(4) J 形式

J 形式のデータパスを**図 12** に示す。これは JUMP、HALT、NOP の 3 命令で使用される。JUMP 命令では即値 11 ビットを絶対アドレスとし、PC にそのまま代入される。NOP 命令では即値フィールドを無視してそのまま PC+1 が代入される。HALT 命令は、PC の値を更新せずに、常に HALT 命令を実行することでプロセッサが停止する。HALT 命令が実行されても割込みが発生すると割込み処理にジャンプすることが可能である。

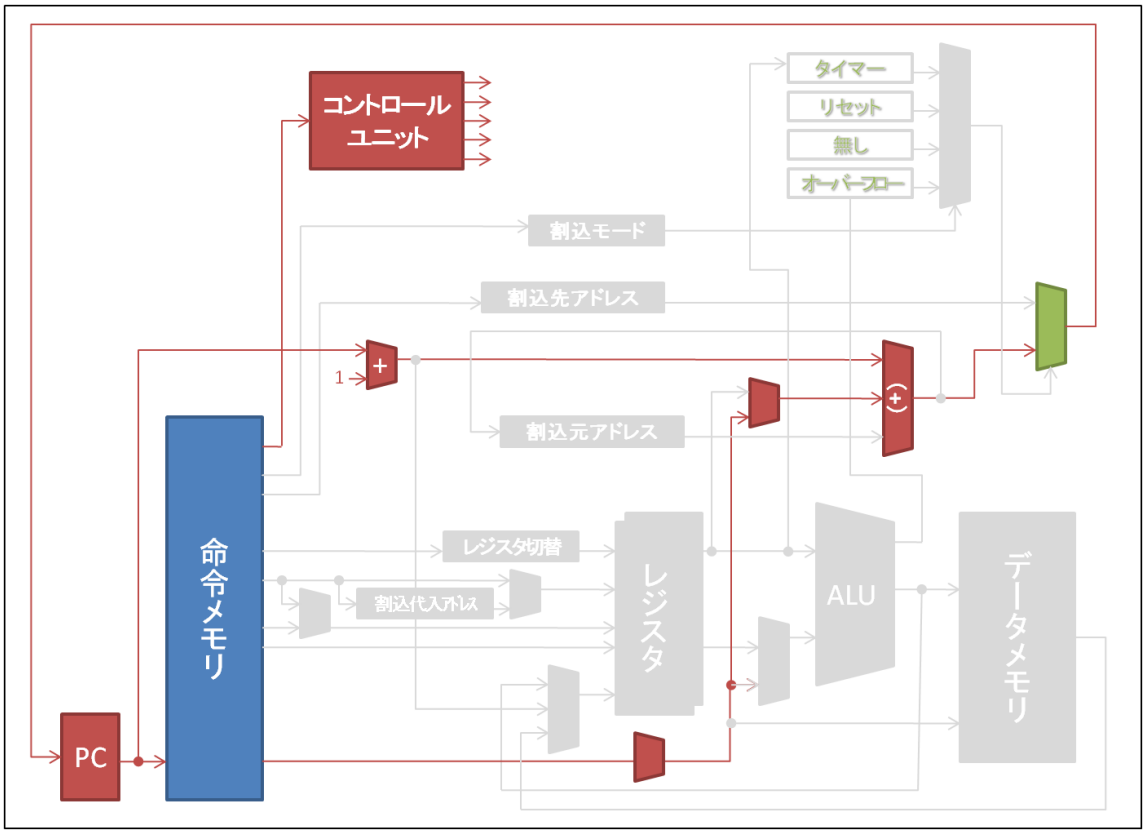


**図 12　J 形式のデータパス**

(5) 割込み発生時

割込み発生時のデータパスを**図 13** に示す。割込モードによって選択された割込みソースから割込み信号が入力されると、PC には IJA 命令で指定した割込先アドレス(INTR_JA レジスタ)が代入される。そして割込元アドレス(INTR_RE レジスタ)には次に実行されるはずだったアドレスが代入される。また割込代入アドレス($R_{OF}$)に命令の $R_{OUT}$ フィールドが代入される。なお、この図は割込みで処理をジャンプするところのデータパスだけをあらわし、灰色にしている部分は発生時に実行していた命令のデータパスになる。

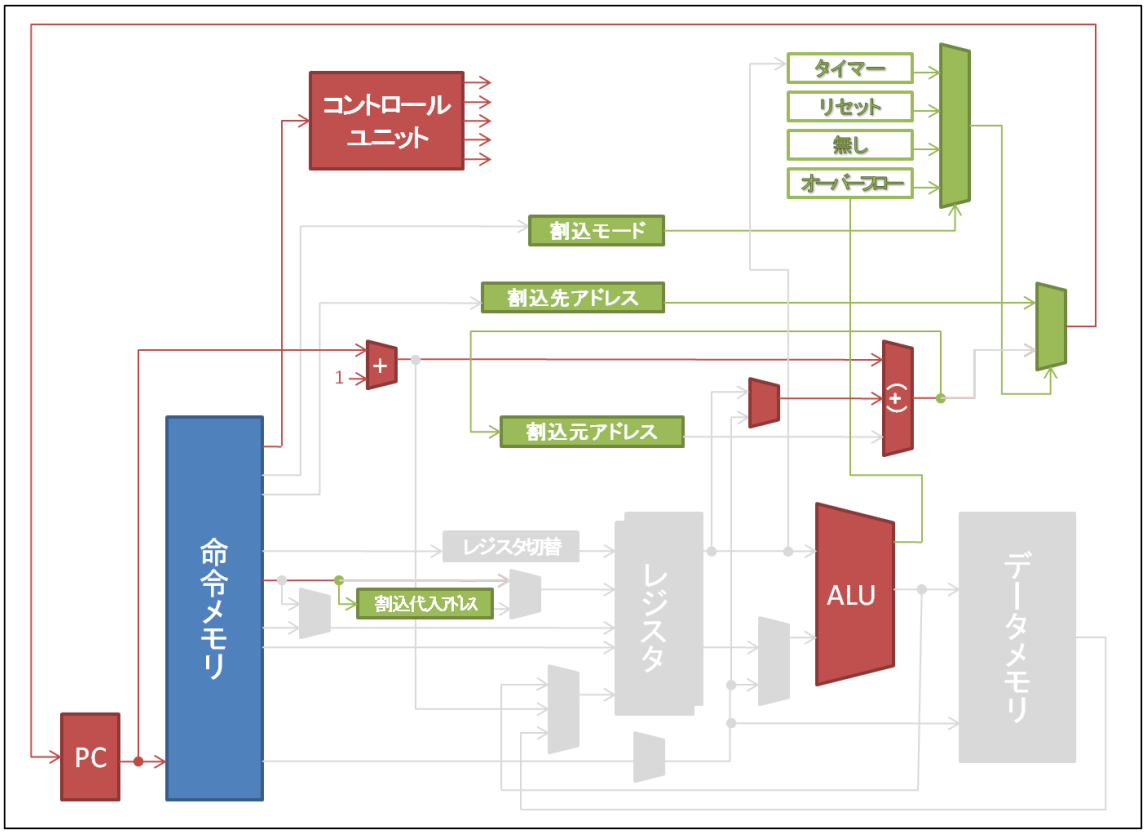


**図 13　割込み発生時のデータパス**

割込み処理にジャンプすると、ユーザーが記述したソフトウェアによってその後の動作が決まる。タイマー割込みの場合では一般的に復帰できることが多いが、オーバーフロー割込みやリセット割込みでは、停止することが多い。割込みの発生でプロセッサを停止する場合、HALT 命令を割込み処理内に記述することでプロセッサの処理を停止できる。

(6) 通常処理への復帰

割込みの種類によっては割込み処理から復帰が可能の場合がある。割込み処理から通常処理へ復帰する時のデータパスを**図 14** に示す。割込みの復帰には割込み発生時に次実行されるべき PC アドレスが割込元アドレス(INTR_RE レジスタ)に保持されている。通常処理へ復帰する IRE 命令によって、PC に割込み元アドレスが代入され通常処理に復帰することが可能である。

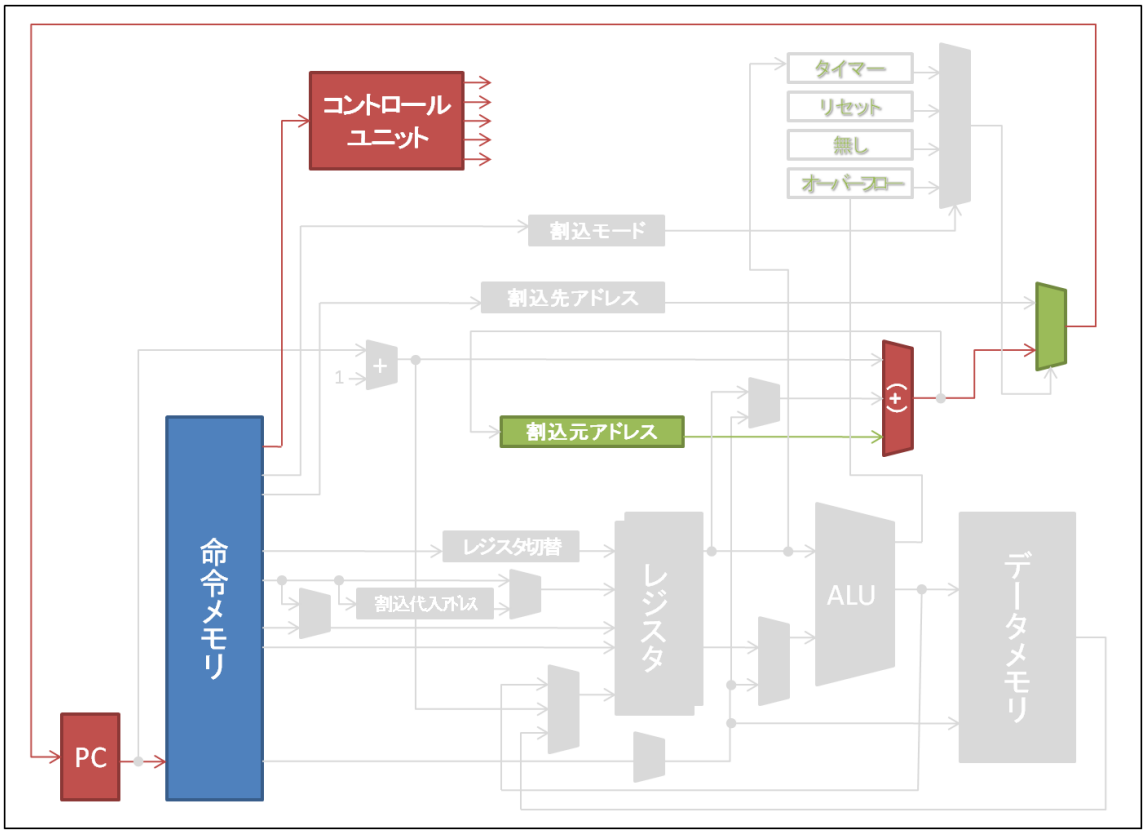


**図 14　通常処理への復帰のデータパス**

# 5　JINT の設計と検証

## 5.1　　Verilog HDL による設計

JINT プロセッサの設計には Verilog HDL を用いた。設計したモジュールと**図 2** に示されている名前の関係、およびそれらの目的を**表 13** に示す。

**表 13　モジュール一覧**

| 名　前 | モジュール名 | 目　的 |
|---|---|---|
| (ヘッダファイル) | _header | ビット幅や命令などの定数を定義する |
| ALU | ALU | 命令で指定された演算を行う |
| 即値幅セレクタ | ALU_IMM_MUX | 取り出す即値の幅を選択する |
| ALU 入力セレクタ | ALU_MUX | 演算に使用する値を選択する |
| コントロールユニット | CU | 全モジュールを制御する |
| PC | PC | 実行する命令アドレスを指定する |
| PC アドレス加算機 | PC_INCR | PC に 1 加算したものを出力 |
| PC アドレスセレクタ | PC_MUX | 次の PC のアドレスを選択する |
| REG_IMM セレクタ | PC_REGIMM_MUX | 相対ジャンプを行うアドレスを選択する |
| レジスタ | REG | レジスタファイル(16bit × 16word) |
| レジスタ入力 0 アドレスセレクタ | REG_IN0_MUX | 入力 0 アドレスを選択する |
| レジスタ入力セレクタ | REG_IN_MUX | レジスタに入力する値を選択する |
| レジスタ切換 | REGBANK | レジスタアドレスの上位 1 ビットを指定する |
| 割込元アドレス | INTR_BA | 通常処理に戻る時のアドレスを保持する |
| 割込代入アドレス | INTR_ISOF | オーバーフローで代入されるレジスタアドレスを保持する |
| 書込アドレスセレクタ | INTR_ISOF_MUX | 書込むアドレスを選択する |
| 割込先アドレス | INTR_JA | 割込み処理の開始アドレスを保持する |
| 割込モード | INTR_MODE | 選択した割込みソースを保持する |
| 割込ソースセレクタ | INTR_MODE_MUX | 割込みソースを選択する |
| 割込有無セレクタ | INTR_PC_MUX | 割込み信号があれば割込み処理にジャンプする |
| タイマー | INTR_TIMER | 割込み用カウントダウンタイマー |
| リセット | (無し) | リセット割込み用、外部入力 |
| 無し | (無し) | 常に LOW が出力される |
| オーバーフロー | (無し) | ALU の演算結果のオーバーフロー |
| (JINT プロセッサトップモジュール) | JINT | シミュレーション用トップモジュール |
| データメモリ | DM | シミュレーション用データメモリ |
| 命令メモリ | IM | シミュレーション用命令メモリ |
| (デバッガ接続用トップモジュール) | TOP | デバッガに接続するためのトップモジュール |

シミュレーション用のモジュールは 5.2 で説明する HDL シミュレータで検証する際に使用する。これらは FPGA へ実装を行うことはできない。以下に割込み制御に関係するモジュールの外部仕様と詳細について説明する。

(1) REG

命令メモリから値を出力するレジスタ番地 REG_OUTADDR0、REG_OUTADDR1 の 2 つと値を書込むレジスタ番地 REG_INADDR を 1 つ入力する。実際に入出力される値は、レジスタ切換 REG_BANKADDR から入力する 1 ビットと各番地の 3 ビット、合計 4 ビットで指定される番地の値である。

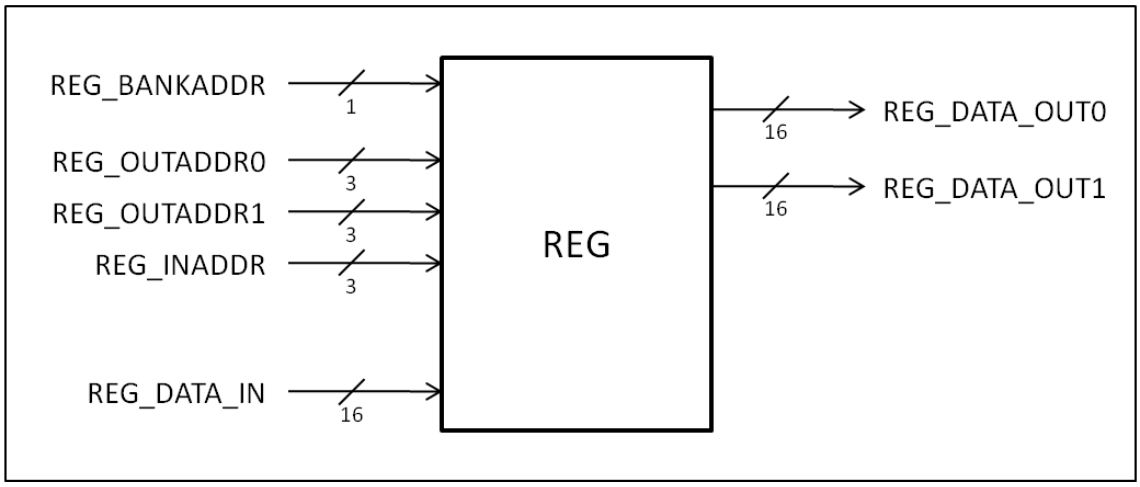


**図 15　REG の入出力仕様**

(2) REGBANK

レジスタの入出力の際の番地の上位 1 ビットを保持する。これの値を切換えることであたかもレジスタが 2 つように扱え、割込み処理時の値の退避を簡単に行うことができる。実行命令が IRB のとき、RB_IN には実行命令の 3 ビット目が入力され、コントロールユニットによって RB_WRITE_ENABLE_IN が HIGH になり、内部のレジスタに書込みが行われる。

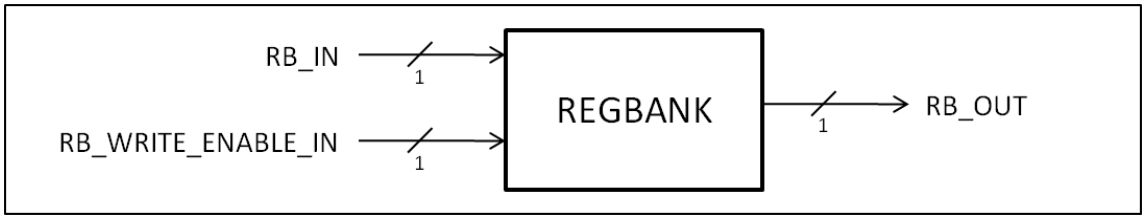


**図 16　REGBANK の入出力仕様**

(3) INTR_ISOF

オーバーフロー代入 ISOF、ISOFI の命令のために設計したモジュールである。INTR_ISOF_IN には割込み信号が入力され、INTR_ISOF_REGADDR_IN には、レジスタに書込む番地(**図 15** の REG_INADDR)が接続される。割込み信号の立ち上がりでオーバーフローの結果が代入されたレジスタ番地が INTR_ISOF 内のレジスタに書込まれる。

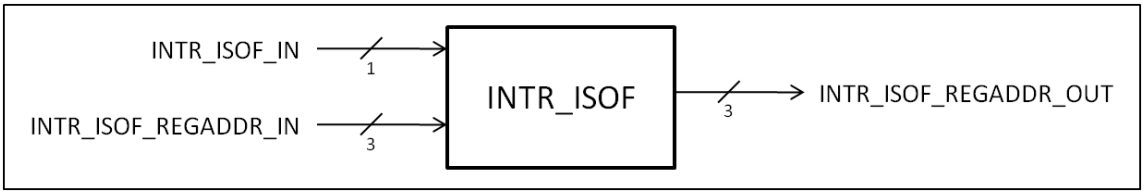


**図 17　INTR_ISOF の入出力仕様**

(4) INTR_MODE

ソフトウェアで選択した割込みモードを保持する。割込みモードの選択には IMD 命令で**表 6** の番号を入力する必要がある。その番号は実行命令の 3~4 ビット目の 2 ビットで指定され、INTR_MODE_IN から入力される。同時にコントロールユニットによって、INTR_MODE_WRITE_ENABLE_IN の書込み信号が HIGH になり、内部のレジスタに保持される。INTR_MODE_OUT は保持している値を出力し、割込みセレクタに接続することで割込みモードの選択を行うことができる。

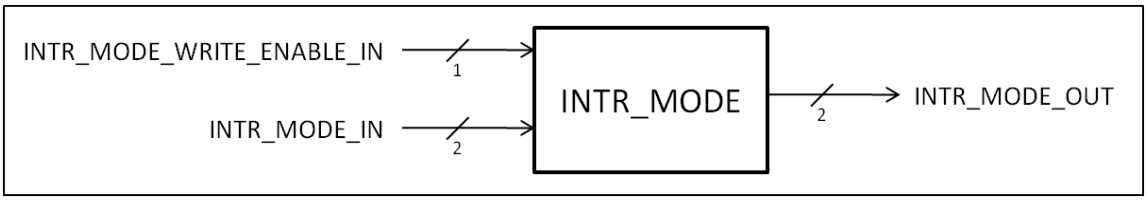


**図 18　INTR_MODE の入出力仕様**

(5) INTR_MODE_MUX

INTR_MODE モジュールで保持されている値が、INTR_MODE_IN に入力される。この値によって 3 つの割込みソースと、常に LOW が出力される 0 から選択される。割込みソースの入力はそれぞれの割込み信号を入力し、選択された割込み信号が INTR_MODE_MUX_OUT に出力される。

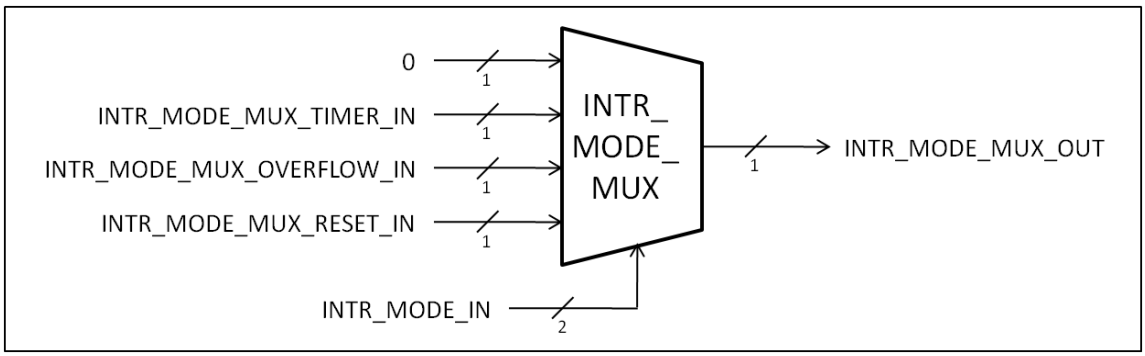


**図 19　INTR_MODE_MUX の入出力仕様**

(6) INTR_TIMER

割込み用カウントダウンタイマーモジュールで、内部のレジスタがゼロになるまでクロックが入るたびに 1 ずつデクリメントを行う。このレジスタの値が 1 のときのみ INTR_TIMER_OUT には HIGH が出力される。REG モジュールの REG_DATA_OUT0 が INTR_TIMER_IN に接続されており、IST 命令によって値の書込みが行われる。

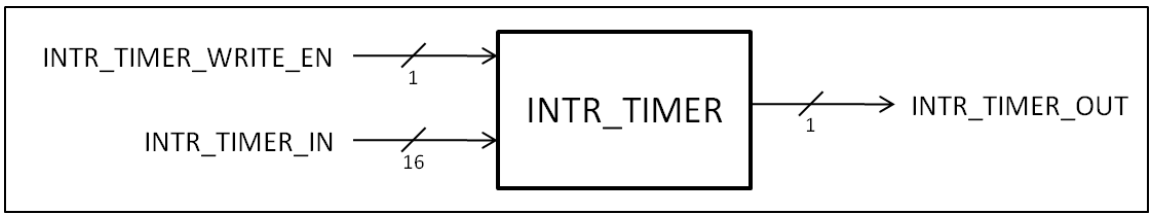


**図 20　INTR_TIMER の入出力仕様**

### 5.2 HDL シミュレータでの検証

設計した JINT プロセッサを Xilinx 社提供の ModelSimXE Ⅲ6.4b を用いて HDL シミュレーションを行った。シミュレーションではシミュレーション用モジュール JINT トップモジュール、命令メモリとデータメモリとしてそれぞれシミュレーション用 IM、DM モジュールを接続した。シミュレーションに用いたプログラムは、通常プログラムとして N までの和、除算、割込みを用いたプログラムとして、10 秒のカウント、ソフトウェアのハングアップを想定したリセット処理、オーバーフローした値の飽和処理である。

#### 5.2.1 通常アセンブリプログラムの検証

N までの和と除算のプログラムの検証について述べる。データメモリには 0 番地に $000A_{16}$、1 番地に $0003_{16}$ の値を入れてそれぞれのプログラムを実行した。N までの和では、データメモリの 0 番地を N の値とし、総和を求めてデータメモリ 1 番地に結果を返す。実行結果として $0037_{16} = 55_{10}$ の正しい結果が得られた。除算プログラムでは A ÷ B = X ... Y とした場合、データメモリの 0 番地 1 番地を A, B を入力とし、2 番地 3 番地に商 X、余 Y を返す。データメモリの値の入出値と処理にかかったクロックサイクル数を**表 14** に示す。

**表 14 通常プログラムの検証結果**

| データメモリ番地＼プログラム | N までの和 | 除算 |
|---|---|---|
| 0 | $000A_{16} = 10_{10}$ | $000A_{16} = 10_{10}$ |
| 1 | $0037_{16} = 55_{10}$ | $0003_{16} = 3_{10}$ |
| 2 | - | $0003_{16} = 3_{10}$ |
| 3 | - | $0001_{16} = 1_{10}$ |
| クロックサイクル数 | 34 | 18 |

```
        LD      $1      0
        SUB     $2      $2      $2
LOOP:   ADD     $2      $2      $1
        SUBI    $1      $1      #1
        BNEZ    $1      LOOP
        ST      $2      1
        HALT
```

**図 21 N までの総和のアセンブリプログラム**

```
        LD      1,0
        LD      2,1
        XOR     3,3,3
LOOP:   SUB     1,1,2
        ADDI    3,3,1
        SLT     4,1,2
        BEQZ    4,LOOP
FIN:    ST      3,2
        ST      1,3
        HALT
```

図 22　除算のアセンブリプログラム

### 5.2.2　割込みを用いたアセンブリプログラムの検証

タイマー割込みを用いて 10 秒のカウントプログラムを検証した。10 秒という時間は 5.3 で述べる FPGA 上でプロセッサが動作する際、動作周波数が 5MHz になるのでそれに合わせて10秒になるように計算を行った。5MHzの動作周波数で10秒を数えるには、$50 \times 10^6$クロックを数える必要がある。タイマーはクロック毎に 1 デクリメントされる。TIMER レジスタに代入できる最大値は 65,535 なので、これを 763 回繰り返すことで、ほぼ 10 秒をカウントすることができる。シミュレーションで検証した結果、処理を終えた時点で 50,005,503 クロックサイクルかかり、動作周波数 5MHz の環境でほぼ 10 秒カウントできることが検証できた。

```
//$1 = TIMER( 0xFFFF = 65535)
//$2 = COUNT( 0x02FB = 763)
        XOR     0,0,0
        LDHI    1,-1
        LDLI    1,-1
        LDHI    2,2
        LDLI    2,-5
        ADD     3,0,2
        IJA     INTR
        IMD     1
        IST     1
LOOP:   JUMP    LOOP
INTR:   SUBI    3,3,1
        BEQZ    3,EXIT
        IST     1
        JUMP    LOOP
EXIT:   HALT
```

図 23　10 秒カウントのアセンブリプログラム

次にソフトウェアのハングアップを想定したリセット割込みの検証を行った。何らかの原因でソフトウェア内の無限ループが発生した場合、内部から処理を抜け出すことは困難である。そこでリセット割込みを用いることでループを抜け出しリセット処理を行う。今回の検証では、意図的に無限ループを発生させリセット割込みによってループから抜け出すことを確認する。シミュレーションでは 100 クロックサイクル経過してからリセット割込みを入力した。シミュレーションの結果、101 クロックサイクルまで無限ループを繰り返し、102 サイクルから割込み処理に入って無限ループを抜け出せたことが確認できた。

```
        IJA     INTR
        IMD     3
        XOR     0,0,0
        XOR     1,1,1
        XOR     2,2,2
LOOP:
        ADDI    1,1,1
        JUMP    LOOP
INTR:
        IMD     0
        LDLI    2,-1
        HALT
```

**図 24　リセット割込みのアセンブリプログラム**

次にオーバーフロー割込みを用いて、オーバーフローした値を飽和し終了するアセンブリプログラムの検証を行う。オーバーフロー割込みは加減算命令などで演算結果が許容の 16 ビットを超えた場合に発生する。今回は、無限ループ内で回数を数えながら、汎用レジスタに $999_{10}$ という値をオーバーフローするまで加算し続け、割込み処理で最大値を代入し終了するアセンブリプログラムで検証を行った。その結果、66 回目の加算でオーバーフロー割込みが発生した。このときオーバーフローしたレジスタには $398_{10}$ という値になっていたが、その後の飽和処理によって最大値 65,535 が代入され処理を終了した。

```
        IJA     INTR
        IMD     2
        XOR     1,1,1
        XOR     2,2,2
        LDHI    3,3
        LDLI    3,-25
LOOP:   ADD     1,1,3
        ADDI    2,2,1
        JUMP    LOOP
INTR:   LDHI    6,-1
        LDLI    6,-1
        ISOF    6
        HALT
```

図 25　オーバーフロー割込みのアセンブリプログラム

### 5.3　　FPGA ボード上での検証

FPGA ボード上に実装するにあたり、開発環境として Xilinx 社の ISE9.2i を使用した。また FPGA ボードには、同じく Xilinx 社の Spartan 3 Starter Kit Board を使用した。FPGA ボード上に実装する際に、設計したプロセッサのデバッガ接続用トップモジュールとプロセッサデバッガを接続した。このとき、命令メモリとデータメモリはプロセッサデバッガから提供さえるものを使用する。

表 15　割込みプロセッサ JINT の設計規模と最大遅延

| モジュール | スライス数 | LUT 数 | フリップフロップ数 | 最大遅延(ns) |
|---|---|---|---|---|
| TOP | 605 | 1160 | 314 | 27.61 |

JINT プロセッサの FPGA 使用率はスライス数が 31%、LUT 数が 30%、フリップフロップ数が 8%となった。最高動作周波数は 36.2MHz であった。

5.2 で検証したプログラム、及び JINT の全命令を使用したアセンブリコードも FPGA ボード上で検証した。すべてにおいて正しい結果が得られ、10 秒カウントのコードも 10 秒で処理を終えたことが確認できた。

## 6　おわりに

本論文では、本研究室で開発を進めているハード/ソフト協調学習システムを用いて、シングルサイクルの割込みプロセッサを設計した。設計したプロセッサを、HDL シミュレータで検証、さらには FPGA ボード上に実装し、検証を行った。本研究を通して、割込み処理の目的と原理・仕組みをハードウェアとソフトウェアの両方の観点から学習を行えた。

今後の課題としては、ハードウェア面ではより多くの割込みの種類や同時に複数に対応すること、またマルチサイクルやパイプラインに対応した割込みプロセッサの設計を行うことがあげられ、ソフトウェア面では仮想シミュレータ環境があるとより効果的に割込み原理の理解度を高めることができるため、仮想シミュレータ環境の開発を進めることがあげられる。

**謝辞**

本研究の機会を与えてくださり、ご指導を頂きました山崎勝弘教授に深く感謝いたします。また、本研究に関して様々な相談に乗って頂き、貴重な助言を頂いた井出純一氏をはじめ、様々な面で貴重な助言や励ましをくださった研究室の皆様に深く感謝いたします。

## 参考文献


[1] 志水 建太 : プロセッサ設計教育のための命令セット・スーパースカラシミュレータの試作と評価、立命館大学理工学研究科修士論文、2009

[2] 難波 翔一朗 : プロセッサ設計支援ツールの実装とハード/ソフト協調学習システムの評価、立命館大学理工学研究科修士論文、2007

[3] 井手 純一 : ハード/ソフト協調学習システム を用いたプロセッサ設計と評価、立命館大学理工学部情報学科卒業論文、2008

[4] 志水建太 : ハード/ソフト協調学習システム上でのプロセッサ設計とプロセッサデバッガによる検証、立命館大学理工学部情報学科卒業論文、2007.

[5] 志水建太 : 命令セット定義ツール・シミュレータ・アセンブラ設計仕様書・使用方法説明書、2009

[6] 難波翔一郎、志水建太、山崎勝弘、小柳滋 : プロセッサ設計支援ツールの設計・実装とハード/ソフト協調学習システムの評価、FIT2007, LC002, 2007.

[7] 池田修久・中村浩一郎・Tran So Cong・難波翔一朗 : RC100 を用いた FPGA ボードコンピュータ設計仕様書、立命館大学理工学研究科　高性能計算研究室・コンピュータシステム研究室、2004

[8] 大八木睦 : ハード/ソフト・コラーニングシステム上でのアーキテクチャ可変なプロセッサシミュレータの設計と試作、立命館大学理工学研究科修士論文、2004

[9] 池田修久 : ハード/ソフト・コラーニングシステム上での FPGA ボードコンピュータの設計と実装、立命館大学理工学研究科修士論文、2004

[10] David A.Patterson/John L.Hennessy 著／成田光彰 訳 : コンピュータの構成と設計（上）（下）、日経 BP 社、2006.

[11] 堀桂太郎 : 図解 Verilog HDL 実習、森北出版、2006.

[12] 小林 優 : 初めてでも使える Verilog HDL 文法ガイド― 記述スタイル編、http://www.kumikomi.net/archives/2009/07/verilog_hdl.php

[13] booran : ためになるページ・Verilog、http://www.booran.com/menu/verilog/