組込みソフトウェア秘匿化のための大規模命令レジスタファイルの活用

研究代表者　　藤　枝　直　輝　　豊橋技術科学大学　電気・電子情報工学系　助教

1 はじめに

近年，ソフトウェアを解析・盗用・改ざんなどから防御する能力，いわゆる耐タンパー性が，組込みシステムにおいても重要視されるようになった．特に，ソフトウェアの解析を通した企業秘密の流出は重要な問題であり，その対策に対する社会的要求は大きい．命令セット（命令と対応する機械語との関係）は通常既知であることから，ソフトウェアの命令列は容易に解析されうる．同様に，命令セットが個々のシステムで共通であることから，悪意のある機械語列が外部から挿入されれば，それは容易に実行されうる．したがって，命令セットに関して必要な情報を攻撃者から隠すことは，耐タンパー性を高める 1 つの手段である．

命令セットランダム化(ISR)[1-5]は，コストパフォーマンスに優れたソフトウェアの秘匿化手法であり，命令と対応する機械語との関係を追加・変更することで，ソフトウェアの解析や改ざんを阻害する．関係が攻撃者から隠されていれば解析は困難であるし，関係を個々のシステムごとに多様化すれば改ざんも困難である．ISR はまた，軽量な命令暗号化と捉えることもできる．すなわち，現代暗号に依拠する AEGIS [6] のようなセキュアプロセッサと比べると実装にかかるコストが小さく済む．しかしながら，命令セットランダム化を適用するにあたっては，本当に機械語列から命令列への推測が困難になっているかが問題となる．

命令レジスタファイル（IRF）[3][7] は，頻繁に使われる命令のリストである．IRF はプロセッサの命令フェッチステージの後に置かれ，搭載された命令にはそのインデックスを用いてアクセスできる．IRF は本来命令フェッチにかかる消費電力を削減するために命令を圧縮する手法であるが，インデックスと実際の命令との関係を隠すことができるため,命令セットランダム化にも利用できる.代表者らの過去の研究[3]では，命令列の推測の困難さとハードウェア使用量のコストとのバランスを取るには，IRF のエントリ数をある程度大きく（例えば 1024 個）し，同一の命令が複数の IRF エントリに割当てられることを許可すればよいことを明らかにしている．

本研究テーマでは,IRF を組込みソフトウェア秘匿化に用いるにあたっての発展的な研究調査を,特に（1）命令列の秘匿性の更なる向上と，（2）多種の命令セットへの手法適用の 2 つの点から行うことにより，手法の有用性や適用可能性を高めることを目的として研究を行った．まず本研究の背景として，ISR および IRF について 2 章で解説する．3 章では，命令列の秘匿性の更なる向上のために行った，位置レジスタとよばれる手法の改良について述べる．4 章では，多種の命令セットへの手法適用の検討として，命令長が可変である x86 アーキテクチャのプロセッサに対する IRF の実装・評価について述べる．5 章では，これらに付随する課題として行った，大規模 IRF における消費電力に関する初期検討について述べる．最後に，6 章で本報告をまとめる．

2 背景

2-1 命令セットランダム化

命令セットランダム化（ISR）は，命令と対応する機械語との関係を追加・変更する手法である．ISR は悪意のあるプログラムコードの挿入（コード挿入攻撃）からシステムを防御する手法の 1 つとして位置付けられるが，そのうちのいくつかはソフトウェアの逆アセンブルによる解析を阻害する難読化手法とも捉えられる．ISR にはエミュレーション等のソフトウェア技術で完全に実現されるもの [1][4] もあるが，組込みシステムにおいてはハードウェアの支援によるもの [2][3][5] が性能オーバーヘッドの小ささの面で有効である．ISR はまた，ランダム化された機械語への変換をあらかじめ行っておく静的な手法と，プログラムの実行時にランダム化を行う動的な手法の 2 種類に分類される．動的な方法にはソフトウェアの解析を阻害する効果はないため，本研究では静的な方法について取り扱う．

静的な ISR 手法においては，ランダム化された機械語列にアクセス可能であることから，元の命令列またはランダム化の鍵を機械語列から容易に推測させないことが重要である．しかし，暗号の安全性と実装のコストとは通常トレードオフの関係にあり，これらを同時に満たすことは挑戦的である．ISR の手法が置換暗

号に基づく場合［3][5]，機械語の使用頻度の偏りに基づいた解析（頻度分析）に対する強度が重要となる．ストリーム暗号［2］や状態機械［4］に基づく手法では，分岐命令の実行後に暗号の内部状態の一貫性を保つことが必要であり，そのための特別な命令を挿入することが性能低下を生むことが報告されている．

2-2 命令レジスタファイル

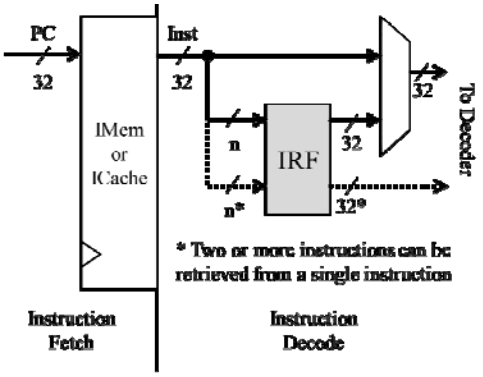

図 1 命令レジスタファイルの構成.

命令レジスタファイル（IRF）[3][7] の構成を図 1 に示す．なお，ここでは MIPS のような一般的な 32 ビットの RISC 型プロセッサを前提に，命令およびアドレス空間を 32 ビットと仮定している．IRF は頻繁に使用される命令のリストであり，搭載された命令にはインデックスを用いてアクセスできる．インデックスを n ビットとすれば，IRF に搭載できる命令の数は 2^n となる．IRF に搭載された命令は，コンパイル時にインデックスを含む特別な命令に変換される．こうした命令が命令メモリからフェッチされると，命令中のインデックスが IRF に渡され，実際の命令が取り出されてデコーダに渡される．通常の命令は直接デコーダに渡される．複数のインデックスを 1 つの命令にパックすることもでき，IRF の当初の提案［7］ではこれにより機械語列の長さを短くすることで消費電力の削減を達成していた．

代表者らの過去の研究［3］では，こうした IRF による命令の変換を ISR の一種と捉え，IRF を耐タンパー化のために利用する方法について検討した．IRF に搭載された命令のリストが攻撃者から隠されていれば，実際の命令を特別な命令から復元することは難しい.すなわち特別な命令は一種の置換暗号としてはたらく．ただし，あくまで IRF は一部のよく使われる命令しか持たず，全ての命令を秘匿することはできないことに留意する必要がある．IRF による命令の変換の例として，仮に `addiu $v0, $v0, 1` なる命令が IRF の 10 番目のエントリに収録されているとすれば，この命令は `IRF #10` などといった命令に変換される．IRF の内容はシステムごとにシャッフルできるので，別のシステムではこの命令は IRF の 20 番目のエントリにあり，この命令は `IRF #20` に変換されるかもしれない．このような多様性をもつことで，あるシステムのために作られた悪意のあるプログラムコードを別のシステムで使用することが難しくなる．なお代表者らは，その実現に必要となる，IRF に搭載された命令を元々の機械語で実行できなくする追加の措置［8］についても提案している．

IRF の当初の提案では，IRF のエントリ数を 32 としていたため，インデックス（図 1の n）は 5 ビット幅であった［7］．これは，機械語列の圧縮効率を高めるため，IRF に収録される命令のカバー率と 1 つの特別な命令にパックできるインデックスの数とのバランスとを考慮して決定されたものであった．一方，耐タンパー化のために IRF を用いる場合,ハードウェア量のコストと IRF の内容が推測できるリスクとを考慮して，IRF のエントリ数を 1,024 程度とある程度大きくすることが推奨される［3].

IRF による耐タンパー化の定量的な指標を定義する際には，なるべく多くの命令を IRF により秘匿化することと，IRF に収録された命令が頻度分析によって推測しにくいこととを考慮している．前者は実行される命令のうち IRF に収録された命令の割合によって定量化でき，これを γ(IRF)とおく．後者は IRF のインデックスの出現頻度の平坦さによって定量化でき，これを E(IRF)とおく．この 2 つの指標から，耐タンパー化指標については，命令 I の動的な実行頻度を P_D(I)，静的な出現頻度を P_S(I)，IRF の i 番目のエントリに対応する命令を IRF_i とおき，耐タンパー化指標 S(IRF)を，

$$\gamma(\mathrm{IRF}) = \sum_{i=0}^{N-1} P_D(\mathrm{IRF}_i),$$

$$E(\mathrm{IRF}) = \frac{-\sum_{i=0}^{N-1} p_s(\mathrm{IRF}_i)\log p_s(\mathrm{IRF}_i)}{\log N},$$

$$S(\mathrm{IRF}) = \gamma(\mathrm{IRF}) \times E(\mathrm{IRF})$$

と定義する．ただし，IRF のエントリ数を N，p_s を $p_s(\mathrm{IRF}_i) = P_s(\mathrm{IRF}_i)/\sum_{j=0}^{N-1} P_s(\mathrm{IRF}_j)$，すなわち相対的なインデックスの出現率とする．これまでの評価の結果，複数の IRF エントリに同一の命令を収録することを許容することにより，わずかな γ(IRF)の減少で E(IRF)を大きく高められることがわかっている [3].

3 位置レジスタの改良

3-1 位置レジスタの原理

表 1 位置レジスタの表現の例.

Line	Original	Positional
1	`add $t0, $s0, $s1`	`add $t0, $s0, $s1`
2	`lw  $t0, 0($t0)`	`lw  dst[0], 0(dst[0])`
3	`add $t1, $s0, $t0`	`add $t1, src[1], dst[0]`
4	`lw  $t1, 0($t1)`	`lw  dst[0], 0(dst[0])`
5	`add $t2, $s0, $t1`	`add $t2, src[1], dst[0]`
6	`lw  $t2, 0($t2)`	`lw  dst[0], 0(dst[0])`

位置レジスタとは，最近実行された命令で使用されたレジスタ番号を記録するレジスタファイルのことである [7]．位置レジスタを用いると，ある命令が先行する命令と同一のレジスタを利用していた場合，そのレジスタを位置レジスタによっても指定できるようになる．これによって同種のよく似た命令をグルーピングし，同一命令とみなすことができれば，これらを IRF の 1 つのエントリに収録することができ，IRF の利用効率を高められる．位置レジスタを使用する命令は，通常のレジスタ番号のかわりに，位置レジスタのインデックスを格納する．また，位置レジスタを使用することを示す指示子ビットに 1 を指定する．こうした命令を実行する際には，位置レジスタを参照し，命令本来のレジスタ番号を復元してから実行する．

位置レジスタによる命令の抽象化の例を表 1に示す．Original が本来の命令表現，Positional が位置レジスタを用いた命令表現を表す．2 行目の命令は，レジスタ`$t0` から読み出し，書き込みを行う命令であり，このレジスタは 1 行目の命令の書き込み先（デスティネーション）と同一である．したがって，このレジスタ番号は 1 つ前のデスティネーション（表中では `dst[0]`と表記）を表す位置レジスタのインデックスで表現できる．同様に，4 行目，6 行目の命令もそれぞれ 3 行目，5 行目のデスティネーションを参照しているので，`dst[0]`を使って表現できる．すると，これら 3 つの命令は位置レジスタを用いた表現ではいずれも `lw dst[0], 0(dst[0])`となる．したがって，これらは元々の命令が異なるにもかかわらず，IRF の 1 つのエントリに収録できるようになる．

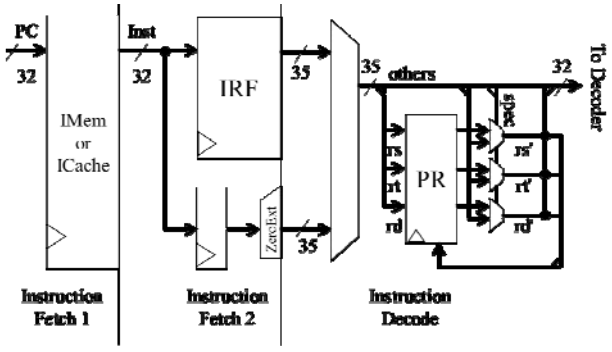

図 2 位置レジスタを付加した IRF の実装.

MIPS プロセッサにおける位置レジスタを付加した IRF の実装例を図 2に示す．位置レジスタに求められる機能は，実行された命令が参照したレジスタ番号を記録することと，位置レジスタを使用する命令の元のレジスタ番号を復元することである．MIPS プロセッサにはレジスタ番号を格納するフィールドが 3 個 (rs, rt, rd) あるので，指示子ビットが 3 ビット付加された 35 ビットの命令が IRF から出力される．各フィールドに

ついて対応する指示子ビットを参照し，指示子が 1 ならば位置レジスタ（PR）を参照して得られたレジスタ番号を，0 ならば元々のレジスタ番号を，そのフィールドのレジスタ番号とすることで，元の命令を復元する．最後に，得られたレジスタ番号を位置レジスタに記録する．

3-2 記録するレジスタ番号

位置レジスタを用いる際には，レジスタ番号をどのように記録するかによって命令のグルーピングの効率が変わりうる．IRF の当初の提案［7］では，読み出し元（ソース）のレジスタ番号とデスティネーションのレジスタ番号をそれぞれ別々に記録していた.以後この方式を PR-Orig と表記する(Orig は Original の意)．この方式の問題点は，命令によってはソースのレジスタ番号が 2 つある（rs または rt）場合があり，その扱いのために位置レジスタのハードウェアが複雑化することである．この問題は，ソースのレジスタ番号が 2 つある命令に対して，その片方（rt）だけを記録するように簡単化することで解決できる．これにより，1 つの命令につき記録されるレジスタ番号は必ず 2 つになる．この方式を PR-SD と表記する（SD は Source, Destination の意）．

本研究では，命令の類似性の更なる抽出を目的として，もう 1 つのレジスタ番号の記録方法を提案する．これはソースとデスティネーションのレジスタ番号に加え，デスティネーションのレジスタ番号に+1, -1 した番号を記録するものであり，この方式を PR-SDPN と表記する（SDPN は Source, Destination, Previous, Next の意）．表 1の例では，3 行目の命令のデスティネーションである`$t1` は過去に使われていないが，2 行目の命令のデスティネーション（`$t0`）の次の番号のレジスタである．同様に，5 行目のデスティネーションである`$t2` は 4 行目のデスティネーション（`$t1`）の次の番号のレジスタである．結果的に，これら 2 つの命令は，いずれも `add dst[0]+1, src[1], dst[0]` といった形に抽象化できる．このように，レジスタ番号の利用に関する空間的な局所性を抽出することが，この方法の狙いである．

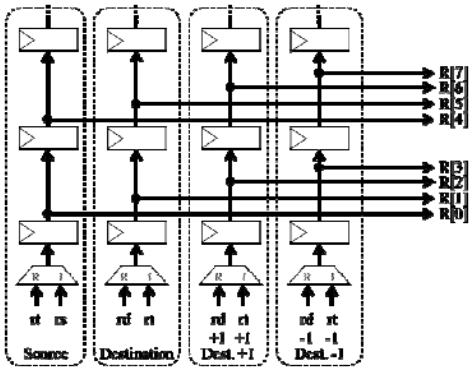

図 3　PR-SDPN 方式の位置レジスタの内部構造.

図 3に，PR-SDPN 方式を適用した場合の位置レジスタの回路の構造を示す．位置レジスタは，レジスタ番号の履歴を記録するシフトレジスタの組で表現される．PR-SDPN 方式の場合，ソース，デスティネーション，その前後のレジスタ番号を記録する 4 組のシフトレジスタが,それぞれ 8 命令分のレジスタ番号を記録する．

3-3 評価方法

本研究では，レジスタ番号の記録方式による耐タンパー化指標やハードウェア実装への影響を評価した．評価にあたっては，代表者らの過去の研究［3］と同様に，組込みプロセッサ向けのベンチマークである MiBench［9］から命令のプロファイルを作成した．ベンチマークは MIPS32 命令セット向けに gcc 4.7.3 でコンパイルされたものを用いる．ただし，位置レジスタを用いる場合は，同一の命令でも先行する命令によっては異なる機械語表現を持ちうる．これを考慮した場合の命令の種類は，過去の研究が 74,589 通りであったのに対して，今回は 375,478 通りである．IRF のエントリ数は 1,024 とする．2-2 節で定義した S(IRF)を耐タンパー化の評価指標とする．3-2 節で述べた 3 つの記録方式（PR-Orig, PR-SD, PR-SDPN）による位置レジスタを用いた場合とともに，評価のベースラインとして位置レジスタを用いない（IRF only）場合についても評価した．

一方，ハードウェア実装の評価には，オープンソースの MIPS I 命令準拠のプロセッサである Plasma［10］に IRF を実装したものを用いた．これに対して Xilinx 社 Spartan-3E XC3S500E FPGA 向けに ISE 14.7 で論理合成・配置配線を行い，報告された論理スライス数（Slice）と最大動作周波数（Fmax）を評価した．

3-4 評価結果

表 2 位置レジスタを備えた IRF の評価結果.

Setting	Tamper Resistance			Hardware	
	γ(IRF)	E(IRF)	S(IRF)	Slice	Fmax[MHz]
IRF only	0.6708	0.9167	0.6149	2,194	31.98
PR-Orig	0.7129	0.9238	0.6586	2,564	28.32
PR-SD	0.7133	0.9241	0.6592	2,255	28.71
PR-SDPN	0.7185	0.9258	0.6652	2,271	28.40

表 2 に，位置レジスタを備えた IRF の耐タンパー化指標およびハードウェア実装の評価結果を示す．耐タンパー化指標については従来の記録方式による位置レジスタの付加により 7.1%，PR-SDPN 方式による位置レジスタの付加により 8.2% 上昇する結果となった．8.2% の耐タンパー化指標の増加は，位置レジスタを備えない IRF においてエントリ数を約 32% 増加させることに相当する．指標の増加の大部分はγ(IRF)の増加，すなわち，複数のよく似た命令を IRF の 1 つのエントリにまとめたことにより，より多くの命令を IRF から実行できたことに由来している．

ハードウェア実装の評価では，提案する記録方式の位置レジスタを追加したことによるスライス数の増加は多くても 77 で，これは Plasma プロセッサの 3.5% に相当する．一方で従来の記録方式では著しくスライス数が増加する結果となった．これは，Xilinx 社の FPGA にはシフトレジスタを効率よく実装する仕組みが備えられているが，従来の記録方式ではその適用のための条件を満たせなかったためであると考えられる．最大動作周波数は最大で 11.4% の低下が見られた．これは Plasma プロセッサがデコードステージの処理に最も長い時間を要していることに由来しており，位置レジスタの処理をフェッチステージに回すようにプロセッサのパイプライン設計を最適化することで解消可能であることを確認している．

4 x86 アーキテクチャへの IRF の適用

4-1 適用方法

これまでの IRF の提案 [3][7] の適用対象は RISC 型のプロセッサである MIPS アーキテクチャや ARM アーキテクチャであった．これらの命令長は 32 ビットと比較的長いが，16 ビットや 8 ビットのより短い命令長を持つアーキテクチャも存在する．もしこうしたアーキテクチャにおける命令のバリエーションが少なければ，命令レジスタファイルによって秘匿化できる命令の増加が予想される．また，可変長の命令をもつ CISC 型のプロセッサである x86 アーキテクチャは，汎用のプロセッサでは広く用いられており，こうしたプロセッサにも IRF が適用可能であることを示すことは実用的な意義が大きい．本研究ではその一検討として，x86 アーキテクチャのプロセッサに IRF を追加し，ハードウェア使用量と命令長の削減について評価した．本章ではその適用方法と，FPGA への実装・評価について述べる．

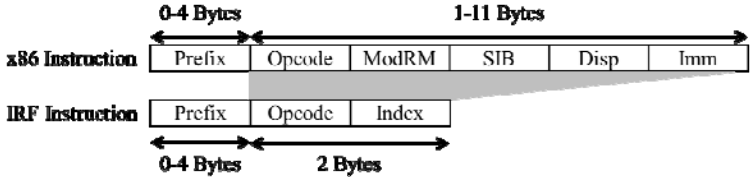

図 4 x86 命令セットにおける IRF 参照命令.

本章における IRF の適用対象は，オープンソースの 80486SX 命令準拠のプロセッサである ao486 [11] とした．本研究では，x86 アーキテクチャでの IRF を参照する命令（IRF 参照命令）を図 4 のように定義した．x86 の命令は，0〜4 個のプリフィクス（Prefix）に続き，命令の種類を表すオペコード（Opcode）や各種のオペランドが並ぶ構成をもつ．ao486 で命令をデコードするにあたっては，まず先頭のバイトがプリフィクスかどうかを判定する．もしプリフィクスであればその内容をデコーダで記憶してからそのバイトを読み飛ばし，プリフィクスでない（オペコードである）バイトに達するまでそれを繰り返す．その後，オペコードに従い命令をデコードし，記憶していたプリフィクスの情報とともに後続のステージに渡す．こうした ao486 の動作を考慮して，本研究での IRF 参照命令は，プリフィクスの後に IRF 参照命令を示すオペコードと，IRF

のインデックスが続くものとした．このとき，命令長は（命令のオペランド部分の長さ － 1）バイト短縮される．IRF のエントリ数は，1 バイトのインデックスで表現可能な 256 個とし，命令長削減の効果を評価するため，IRF には短縮の期待値，すなわち（短縮される命令長×実行頻度）が大きい命令を順に格納するものとする．

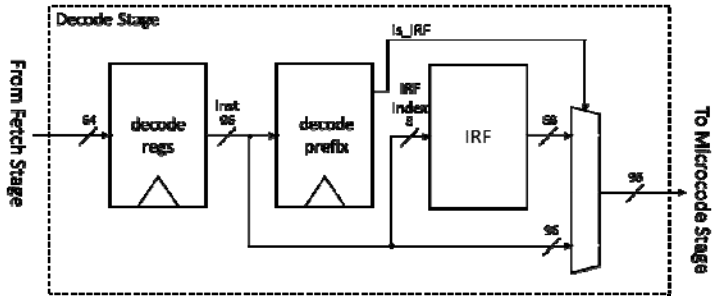

図 5 ao486 における IRF の実装.

図 5 に ao486 における IRF のデコードステージの実装の概略を示す．図中の decode_regs はフェッチされた命令列を記憶する一種の FIFO であり，前述したプリフィクスの判断およびその内容の記憶は decode_prefix モジュールで行われる．本研究での実装では，プリフィクスの判断と同時に IRF 参照命令かどうかの判断を行う．もし IRF 参照命令であれば IRF から元の命令が出力され，命令デコーダや後続のマイクロコードステージへ渡る信号が IRF 参照後の命令に切り替わる．そうでなければ，decode_regs 内の命令がそのままデコードされ，後続ステージに渡される．

4-2 評価方法

IRF に格納する命令の最大長が命令長の削減やハードウェア使用量に与える影響を評価するため，（プリフィクスを除く）命令の最大長を 3 バイトから 8 バイトまで変化させ，それぞれの場合で IRF を FPGA 上に実装し，評価した．命令長の削減の評価にあたっては，486SX 向けにコンパイルされた MiBench [9] をもとに命令プロファイルを作成し，命令が最大長以下であり短縮の期待値が大きい命令を IRF に格納した．評価対象のプログラムは Dhrystone ベンチマークおよび MiBench の bitcount，dijkstra，stringsearch とし，先頭から 10 万命令をシミュレートした際の，フェッチステージから供給される命令の合計サイズ（以下フェッチされた命令長とよぶ）および decode_regs から取り出される命令の合計サイズ（以下デコードされた命令長とよぶ）を評価した．

ハードウェア使用量の評価にあたっては，ao486 のプロセッサ部分に対して Xilinx 社 Zynq XC7Z020 FPGA SoC 向けに Vivado 2017.1 で論理合成・配置配線を行い，報告された論理スライス数（Slice）を評価した．動作周波数は 50 MHz に設定した．

4-3 評価結果

表 3 ao486 における IRF 追加による命令長とハードウェア使用量の増減.

Length	Fetched Instruction				Decoded Instruction				HW
[Byte]	Dh	Bi	Di	St	Dh	Bi	Di	St	Slice
3	-2.0	-4.6	-0.3	+2.6	-4.1	-11.1	-0.3	-0.2	-2.5
4	-3.6	-5.0	-0.5	-0.1	-6.5	-11.7	-1.1	-0.2	+1.3
5	-3.6	-5.0	-0.5	-0.1	-6.4	-11.7	-1.1	-0.2	-1.8
6	-3.7	-6.0	-0.5	-0.1	-6.5	-14.1	-1.1	-0.2	-1.4
7	-3.0	-5.0	-0.5	+2.6	-5.7	-11.7	-1.1	-0.2	-1.9
8	-3.0	-5.0	-0.5	+2.6	-5.7	-11.7	-1.1	-0.2	-1.4

表 3 に，ao486 に IRF を追加した際の命令長の削減およびハードウェア使用量の評価結果を示す．Length は IRF に格納する命令の最大長を表し，Dh,Bi,Di,St がそれぞれ Dhrystone,bitcount,dijkstra,stringsearch の結果を示す．結果の数値はいずれも IRF を追加する前との相対値（単位は%）を表す．

デコードされた命令長は，bitcount を最大長 6 バイトの IRF を用いて実行した場合に削減幅が最も大きくなり，このとき削減幅は 14.1%であった．最大長 7 バイト以上の時にむしろ削減幅が小さくなるのは，短縮の期待値が大きい命令として実行頻度は低いが命令長が長い命令が選ばれるようになったものの，こうした

命令が実際には使われなかったためであると考えられる．一方，フェッチされた命令長の削減は最大で 6.0%にとどまった．これは，フェッチされて decode_regs に溜められていた命令が，分岐予測ミスによりフラッシュされることに起因する．ao486 は分岐が成立しないことを前提に命令フェッチを続ける設計を取っているため，分岐予測ミスが頻繁に発生し，フラッシュされる命令が多い．その結果削減率の計算における分母が大きくなるため，削減率も低くなる結果となった．なお，ハードウェア使用量の増減は CAD による最適化で吸収される範囲（2.5%減～1.3%増）であった．

以上の評価結果から，x86 アーキテクチャに対しても IRF を実装することで一定の効果が見込めることがわかったが，こと消費電力の削減を達成する場合には，命令長の削減だけでなく分岐予測などの要因も考慮すべきであることが示唆された．また，耐タンパ性向上に向けたより詳細な検討も今後の課題として残された．

5 大規模 IRF における消費電力に関する初期検討

5-1 2 つの IRF 参照命令のパッキング

2-2 節でも述べたように，また 4 章でも評価した通り，IRF は元は命令長の削減による命令フェッチの消費電力を削減するための手法であった [7]．これまでの代表者らの研究 [3] では大規模 IRF による秘匿性向上のみにフォーカスし，複数の命令をパッキングすることによる消費電力削減については考慮外としてきた．本研究では電力の削減効果を評価するための前段階として，パッキングを考慮に入れた命令レジスタファイルの実装・評価を行った．

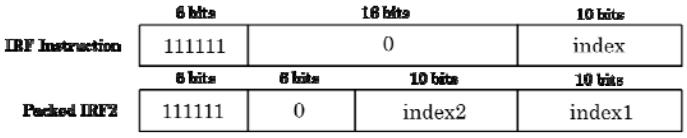

図 6 パックド IRF 命令のフォーマット．

図 6 に今回検討した，2 つの IRF 参照命令をパッキングする命令（パックド IRF 命令）のフォーマットを示す．従来の IRF 参照命令のフォーマット [3] をベースに，予約されていた 16 ビットのうち 10 ビットを第 2 の命令のインデックスとして使用する．IRF の 0 番目のエントリには必ず何もしない命令（nop 命令）を格納することで，従来の IRF 参照命令は（index2 を 0 とした）パックド IRF2 命令の一部として扱う．

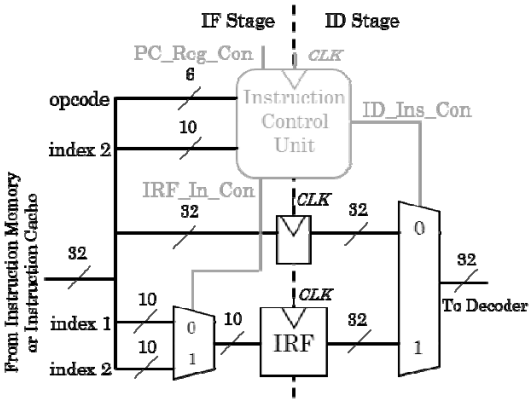

図 7 パックド IRF 命令を付加した IRF のデータパスと制御．

本章における IRF，およびパックド IRF 命令の適用先は，オープンソースの MIPS32 Release 1 命令準拠のプロセッサである OpenCores の mips32r1 [12] とした．このプロセッサは 1 クロックサイクルに 1 命令を実行するインオーダ型のプロセッサである．パックド IRF 命令を付加した IRF 回路を図 7 に示す．パックド IRF 命令がフェッチされると，インデックスを 2 つ持つかどうか（index2 が 0 でないか）を判別する．もしインデックスを 2 つ持つならば，デコーダに 2 つの命令を順番に供給するために以下の動作を行う．(1) 最初のクロックサイクルでは index1 に対応する命令を供給するとともに,一時的にプログラムカウンタの更新を停止する．(2) 次のクロックサイクルで index2 に対応する命令を供給する．インデックスを 1 つだけ持つ場合は，単に index1 に対応する命令だけを供給する．

5-2 評価方法

本章の評価では，分岐命令とその後続の命令（遅延スロット）を IRF への参照に置換しないものとして評価を行った．命令長の削減の評価においては， MIPS32 Release 1 向けにコンパイルされた MiBench [9] をもとに命令プロファイルを作成し，過去の研究 [3] と同様の方法で IRF に格納する命令を選出した．評価対象のプログラムは Dhrystone ベンチマークおよび MiBench の bitcount，dijkstra，stringsearch とし，プログラム全体をシミュレートした際の，命令メモリから命令がフェッチされた回数を測定した．

ハードウェア使用量の評価にあたっては，mips32r1 のプロセッサ部分に対して Xilinx 社 Artix-7 XC7A35T FPGA 向けに Vivado 2017.3 で論理合成・配置配線を行い，報告された論理スライス数（Slice）を評価した．動作周波数は 109 MHz に設定した．

5-3 評価結果

表 4 mips32r1 におけるパックド IRF 命令追加によるフェッチ回数とハードウェア使用量の増減.

	# of Fetch [$\times 10^3$]				HW
	Dh	Bi	Di	St	Slice
Baseline	52,700	37,112	32,173	1,750	1,342
Baseline + IRF	52,700	37,112	32,173	1,750	1,387
Baseline + Packed IRF	46,700 (−11.4%)	31,997 (−13.8%)	32,158 (−0.1%)	1,727 (−1.3%)	1,365 (+1.7%)

表 4 に，mips32r1 に IRF，およびパックド IRF 命令を追加した際のフェッチ回数およびハードウェア使用量の評価結果を示す．Baseline + IRF が従来の IRF 参照命令を実装した場合，Baseline + Packed IRF がパックド IRF 命令を実装した場合を示す．# of Fetch は命令がフェッチされた回数を示し，HW がスライス数を示す．Baseline + Packed IRF については Baseline からの相対的な増減を括弧書きで併記している．

Baseline + IRF におけるフェッチ回数は，Baseline と完全に一致した．これは（従来の）IRF 参照命令が元の命令と 1 対 1 対応していることから妥当である．Baseline + Packed IRF におけるフェッチ回数の減少はプログラムにより異なり，最大で 13.8%（bitcount）であった一方，0.1%（dijkstra）とほとんど減少しなかったプログラムもあった．これらのプログラムの特徴を分析する限りでは，元々多くの命令が IRF への参照で置き換えられており，かつ分岐命令の少ないプログラムほどフェッチ回数の削減が大きいことがうかがえた．なお，ハードウェア使用量の増加は Baseline 比で 1.7%の増加であったが，従来の IRF 参照命令を実装した場合よりも小さくなった．したがって，この増加は CAD による最適化で吸収される範囲であると考えられる．

以上の評価結果から，大規模 IRF を使用した場合でも命令のパッキングが行えて命令フェッチの回数が削減できること，それによる追加のハードウェアの増加はわずかであることが確認できた．今後の課題としては，詳細な消費電力に関する検討はもちろんのこと，より多くの命令がパッキングできるように命令の順序を入れ替えるといった効率化の検討，更にはそれらを耐タンパ性の向上と両立させることが挙げられる．

6 おわりに

本研究テーマでは，IRF を組込みソフトウェア秘匿化に用いるにあたっての発展的な研究調査を行った．命令列の秘匿性の更なる向上の点からは位置レジスタとよばれる手法の適用とその改良を，多種の命令セットへの手法適用の点からは x86 アーキテクチャへの IRF の実装と評価を行い，付随する課題として大規模 IRF における消費電力の削減効果に関する初期評価を行った．

x86 アーキテクチャへの実装やパックド IRF 命令の実装を通じて，IRF への参照に変換する際に命令長が変化する場合，秘匿性を保ったまま変換を行うためには，特に分岐命令の扱いについていくつか考慮すべき点があることもわかった．その解決は今後の課題としたい．

【参考文献】

[1] E.G. Barrantes, D.H. Ackley, S. Forrest, and D. Stefanović, “Randomized instruction set emulation,” ACM Trans. Inf. Syst. Secur., vol.8, no.1, pp.3–40, 2005.

[2] J.L. Danger, S. Guilley, and F. Praden, “Hardware-enforced Protection against Software Reverse-Engineering based on an Instruction Set Encoding,” Proc. 3rd ACM SIGPLAN Program Protection and Reverse Engineering Workshop, San Diego, CA, pp.5:1–5:11, 2014.
[3] N. Fujieda, T. Tanaka, and S. Ichikawa, “Design and Implementation of Instruction Indirection for Embedded Software Obfuscation,” Microproc. Microsy., vol.45, no.A, pp.115–128, 2016.
[4] A. Monden, A. Monsifrot, and C. Thomborson, “Tamper Resistant Software System Based on a Finite State Machine,” IEICE Trans. Fundam., vol.E88-A, no.1, pp.112–122, 2005.
[5] A. Papadogiannakis, L. Loutsis, V. Papaefstathiou, and S. Ioannidis, “ASIST: Architectural Support for Instruction Set Randomization,” Proc. 20th ACM Conference on Computer and Communications Security, Berlin, Germany, pp.981–992, 2013.
[6] G.E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, “AEGIS: architecture for tamper-evident and tamperresistant processing,” Proc. 17th annual international conference on Supercomputing, San Francisco, CA, pp.160–171, 2003.
[7] S. Hines, J. Green, G. Tyson, and D. Whalley, “Improving program efficiency by packing instructions into registers,” Proc. 32nd annual international symposium on Computer Architecture, Madison, WI, pp.260–271, 2005.
[8] N. Fujieda, K. Sato, and S. Ichikawa, “A Complement to Enhanced Instruction Register File against Embedded Software Falsification,” Proc. 5th Program Protection and Reverse Engineering Workshop, Los Angeles, CA, pp.3:1–3:7, 2015.
[9] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown, “MiBench: A free, commercially representative embedded benchmark suite,” Proc. 2001 IEEE International Workshop on Workload Characterization, Austin, TX, pp.3–14, 2001.
[10] S. Rhoads, “Plasma – most MIPS I(TM) opcodes.” https://opencores.org/project,plasma
[11] A. Osman, “ ao486 :: Overview. ” https://opencores.org/project,ao486
[12] G. Ayers, “ MIPS32 Release 1 :: Overview. ” https://opencores.org/project,mips32r1

〈発　表　資　料〉

題　　名	掲載誌・学会名等	発表年月
Evaluation of Register Number Abstraction for Enhanced Instruction Register Files	IEICE Transactions on Information and Systems	2018 年 6 月（採録決定）
CISC プロセッサ ao486 を対象とした命令レジスタファイルの実装	電子情報通信学会 2018 年総合大会	2018 年 3 月
MIPS プロセッサに対するパックド IRF 命令の実装と評価	電子情報通信学会 2018 年総合大会	2018 年 3 月