HY-225: Οργάνωση Υπολογιστών
Άνοιξη 2019

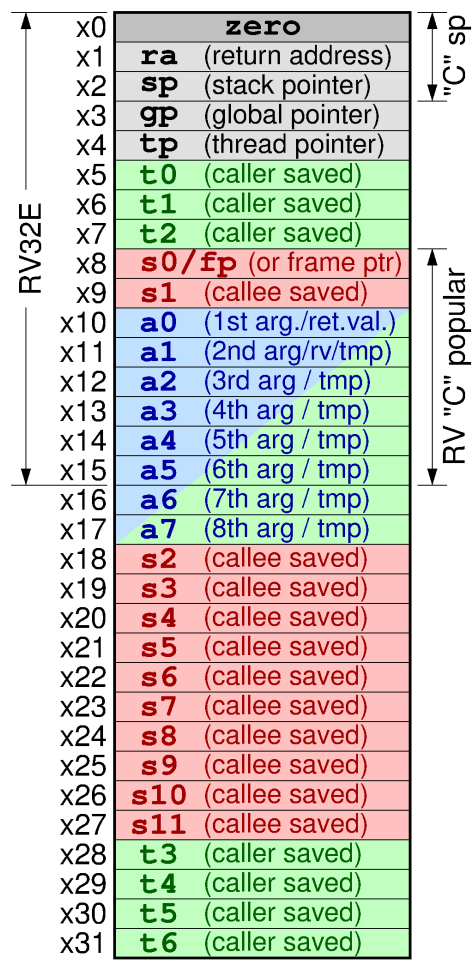
Τμ. Επ. Υπολογιστών
© Πανεπιστήμιο Κρήτης

Διαλέξεις 6:
Κάλεσμα Διαδικασιών, Χρήση και Σώσιμο Καταχωρητών

Βιβλίο: Διαβάστε την §2.8: pp. 98 - 107 στο Αγγλικό βιβλίο, ή σελίδες 152 - 163 στο Ελληνικό.

Οι διαδικασίες (procedures) αποτελούν βασική μονάδα του δομημένου, αρθρωτού (modular) προγραμματισμού, και εδώ θα εξετάσουμε τον τρόπο κλήσης και επιστροφής τους, και περάσματος ορισμάτων και επιστρεφόμενων τιμών στην C στον RISC-V. Θα ξεκινήσουμε με τη χρήση των καταχωρητών και τη διατήρηση των περιεχομένων τους, θα συνεχίσουμε με το πέρασμα ορισμάτων και επιστρεφόμενης τιμής, και θα τελειώσουμε με τα άλματα κλήσης και επιστροφής και με ένα συνολικό παράδειγμα.

6.1 Τίνος Ευθύνη είναι το Σώσιμο των Καταχωρητών



Επειδή συνήθως οι διαδικασίες έχουν λίγα ορίσματα και τομικές μεταβλητές, και επειδή συνήθως αυτά τα προσπελάζουν πολύ συχνά, οι compilers του RISC-V αυτά τα βάζουν (συνήθως) σε καταχωρητές. Όμως, προφανώς, οι μεταβλητές αυτές είναι διαφορετικές γιά τον "γονέα" (*καλούσα* διαδικασία - **caller**), και διαφορετικές γιά το "παιδί" (*καλούμενη* διαδικασία - **callee**). Επιπλέον, γονέας (ή και πρόγονοι) και παιδί (ή και απόγονοι) μπορεί να έχουν μεταφραστεί (compiled) χωριστά (από άλλα αρχεία), άρα δεν ξέρει ο ένας ποιούς καταχωρητές χρησιμοποιούν οι άλλοι, επομένως πρέπει να φροντίζουν γιά τη διατήρηση των τιμών που έχουν σε καταχωρητές και που χρειάζονται οι γονείς/πρόγονοι όταν τα παιδιά/απόγονοι, *ενδεχομένως* (αφού δεν το ξέρουμε) να χρησιμοποιούν τους ίδιους καταχωρητές γιά διαφορετικές, δικές του μεταβλητές.

Σε ένα κάλεσμα διαδικασίας, η αφελής πρακτική θα ήταν και ο γονέας, από υπερβολικό φόβο, να σώζει τις τιμές των καταχωρητών που χρησιμοποιεί στη μνήμη –στη στοίβα, όπως θα πούμε παρακάτω– και το παιδί, από υπερβολική αίσθηση ευθύνης, να προστατεύει την παλαιά τιμή του κάθε καταχωρητή, σώζοντάς την (επίσης στη μνήμη), πριν την "χαλάσει" γιά να βάλει εκεί κάτι δικό του. Προφανώς, είναι διπλός και περιττός κόπος *και* οι δύο να το κάνουν αυτό –αρκεί να το κάνει μόνον ο ένας από τους δύο –αλλά *ποιός* από τους δυό; Όπως θα δούμε, μερικές φορές συμφέρει να κάνει το σώσιμο (και επαναφορά) ο ένας, και άλλες φορές ο άλλος. Γιά να δουλέψει σωστά αυτό, και σε συνθήκες χωριστού compilation, ο RISC-V χωρίζει τους καταχωρητές του σε κατηγορίες, όπως στο σχήμα, και ορίζει **Συμβάσεις Χρήσης Καταχωρητών**, που επιγραμματικά είναι οι εξής:

- **Temporary** Registers (γιά προσωρινή χρήση - **t** - πράσινοι στο σχήμα): οιαδήποτε διαδικασία είναι ελεύθερη να τους χρησιμοποιήσει θεωρώντας τους διαθέσιμους, δηλαδή χωρίς να νοιάζεται να σώσει το παλαιό περιεχόμενό τούς. Άρα, σε ένα κάλεσμα, επειδή το παιδί (ή άλλος απόγονος) μπορεί να τους "χαλάσει", είναι **ευθύνη του καλούντος** (του γονέα), εάν έχει κάτι χρήσιμο μέσα, να το σώσει (στη στοίβα)

πριν το κάλεσμα και να το επαναφέρει από εκεί μετά την επιστροφή του καλέσματός (**caller-saved**).

- **Saved** Registers (γιά μακροπρόθεσμη διατήρηση - **`s`** - κόκκινοι στο σχήμα): διατηρούν το περιεχόμενό τους διαμέσου καλεσμάτων (across calls), διότι αποτελεί ευθύνη των παιδιών/απογόνων να **μην καταστρέφουν** το περιεχόμενό τους. Επομένως, είναι **ευθύνη του καλούμενου**, εάν θέλει να χρησιμοποιήσει τέτοιον καταχωρητή, πριν τον χρησιμοποιήσει να σώσει στη στοίβα ό,τι τυχόν είχε εκεί ενδεχομένως κάποιος πρόγονός του, και να το επαναφέρει στο τέλος, πριν επιστρέψει στους προγόνους του (**callee-saved**).

Στο σχήμα, αριστερά από τον κάθε καταχωρητή, με μαύρο χρώμα, είναι γραμμένος ο αριθμός του καταχωρητή τον οποίο "ξέρει" το hardware, και το αντίστοιχο όνομα που ξεκινά με το γράμμα "x". Μέσα στο "κουτί" του κάθε καταχωρητή είναι γραμμένο ένα άλλο, συμβολικό όνομά του, το οποίο μας θυμίζει την standard χρήση του σύμφωνα με τις παρούσες συμβάσεις, και το οποίο επίσης το αναγνωρίζει ο Assembler και το μεταφράζει στον πραγματικό αριθμό του καταχωρητή μέσα στις εντολές. Στο σχήμα, σημειώνεται με ένα βέλος "RV32E" το γεγονός ότι η προαιρετική παραλλαγή "E" (Embedded) του 32-μπιτου RISC-V (RV32) έχει μόνον τους πρώτους 16 καταχωρητές, x0 - x15, με σκοπό να επιτρέπει πολύ μικρούς επεξεργαστές, με πολύ λίγα flip-flops. Επίσης, με δύο βέλη RV "C" σημειώνεται η προαιρετική παραλλαγή "C" (Compact) των 16-μπιτων εντολών. Σε αυτές: (α) υπάρχουν ειδικοί opcodes (και λιγότερα πεδία τελεστέων) όταν ένας ή περισσότεροι τελεστέοι είναι οι ειδικοί (και δημοφιλείς) καταχωρητές zero, ra, sp, και (β) τα υπόλοιπα πεδία καταχωρητών είναι των 3 bits μόνον καθένα, και αναφέρονται μόνον στους πιό δημοφιλείς καταχωρητές, x8 - x15. Έτσι εξηγείται το γεγονός ότι οι καταχωρητές τύπων **`s`** και **`t`** βρίσκονται "σπαρμένοι" μερικοί στις πρώτες και μερικοί στις τελευταίες θέσεις μέσα στο αρχείο των καταχωρητών. Ας τα δούμε τώρα με μεγαλύτερη λεπτομέρεια τους τύπους **`s`** και **`t`** καταχωρητών, ξεκινώντας με έναν ορισμό των μεταφραστών (compilers) που θα τον χρησιμοποιήσουμε κι εμείς στη συνέχεια:

> **Διάρκεια ζωής (lifetime)** μιάς μεταβλητής ή ενός καταχωρητή στον οποίο κρατάμε μιά μεταβλητή ή μιά ενδιάμεση τιμή υπολογισμού: λέγεται η χρονική περίοδος (ή οι εντολές που περιλαμβάνονται) από τη στιγμή πού στη μεταβλητή ή στον καταχωρητή εκχωρείται μία "αρχική" τιμή" (που δεν είναι συνάρτηση αυτής της ίδιας της μεταβλητής ή καταχωρητή), μέχρι την τελευταία ανάγνωση της τιμής αυτής πριν την επόμενη εκχώρηση νέας τιμής σε αυτή τη μεταβλητή ή τον καταχωρητή (που δεν είναι συνάρτηση του εαυτού της, δηλαδή που δεν την/τον διαβάζει αμέσως πριν την/τον γράψει). Μετά το τέλος μιας διάρκειας ζωής μιας μεταβλητής ή ενός καταχωρητή, και μέχρι την έναρξη της επόμενης διάρκειας ζωής της/του, η μεταβλητή ή ο καταχωρητής θεωρείται "νεκρή/νεκρός" (**dead**), διότι η τιμή της/του **δεν** μας ενδιαφέρει –κανείς δεν πρόκειται να την διαβάσει πριν την ξαναλλάξουμε– άρα είμαστε ελεύθεροι να την καταστρέψουμε.

Προσωρινοί Καταχωρητές - *temporary registers*: **`t0`** έως και **`t6`**, καθώς και όσοι από τους καταχωρητές ορισμάτων, **`a`**, δεν χρησιμοποιούνται σαν ορίσματα από την τρέχουσα διαδικασία (7 έως 14 καταχωρητές, συνολικά): Η τιμή των καταχωρητών αυτών *δεν* διατηρείται μετά από ένα κάλεσμα διαδικασίας (not preserved across call), δηλαδή η καλούμενη διαδικασία (ή άλλες που τυχόν καλούνται από αυτήν) επιτρέπεται να μεταβάλει την τιμή αυτών των καταχωρητών χωρίς προηγουμένως να σώσει την τιμή που τυχόν είχε μείνει εκεί από την καλούσα διαδικασία, επομένως και χωρίς να επαναφέρει την παλαιά εκείνη τιμή προ της επιστροφής στην καλούσα διαδικασία. Αρα, αν η καλούσα διαδικασία έχει κάτι χρήσιμο μέσα σε έναν τέτοιο καταχωρητή ενώ ετοιμάζεται να καλέσει μιαν άλλη διαδικασία, το οποίο χρήσιμο επιθυμεί να το ξαναβρεί στη θέση του μετά την επιστροφή του καλέσματος, είναι **ευθύνη της καλούσας διαδικασίας** να σώσει την χρήσιμη τιμή πριν το κάλεσμα ("**caller-saved**") (στη στοίβα στη μνήμη), και να την επαναφέρει (restore) από εκεί αμέσως μετά, δηλαδή μετά την επιστροφή του καλέσματος.

Προσωρινές τιμές των οποίων η διάρκεια ζωής (lifetime) δεν περιλαμβάνει καλέσματα διαδικασιών συμφέρει να τοποθετούνται σε τέτοιους καταχωρητές, **`t`**, διότι δεν χρειάζεται ούτε να σώσουμε τα παλαιά περιεχόμενα (από αυτόν που μας κάλεσε) πριν τους χρησιμοποιήσουμε (επειδή είναι τύπου t), ούτε να σώσουμε τα νέα περιεχόμενά τους πριν

τελειώσει η χρήση τους, αφού κατά τη διάρκεια ζωής τους δεν καλούμε καμία άλλη διαδικασία, άρα δεν κινδυνεύουμε από κανέναν απόγονο να μας χαλάσει το δικό μας περιεχόμενο. Εάν κατά τη διάρκεια ζωής αυτών των τιμών υπήρχαν καλέσματα θυγατρικών διαδικασιών, τότε σε κάθε τέτοιο κάλεσμα θα χρειάζονταν σώσιμο του "προσωρινού" αυτού καταχωρητή, και επαναφορά του μετά την επιστροφή, αφού η καλούμενη διαδικασία θα ήταν ελεύθερη (πιθανόν) να καταστρέψει την εκεί περιεχόμενη τιμή, οπότε και δεν θα συνέφερε η χρήση καταχωρητή τύπου **`t`**· όμως, ακριβώς επειδή δεν υπάρχουν καλέσματα θυγατρικών διαδικασιών κατά τη διάρκεια ζωής αυτής της τιμής, γι' αυτό και συμφέρει η τοποθέτησή της σε "προσωρινό" καταχωρητή.

Ένα πόρισμα είναι ότι διαδικασίες-φύλλα (**leaf procedures**), δηλαδή διαδικασίες που δεν καλούν καμία άλλη διαδικασία μέχρι να επιστρέψουν οι ίδιες (φύλλα στο δένδρο της (δυναμικής) κλήσης διαδικασιών), συμφέρει να βάζουν *όλες* τις τοπικές τους μεταβλητές και άλλες ενδιάμεσες τιμές σε καταχωρητές αυτού του τύπου, **`t`**. Παρατηρήστε ότι εάν το δένδρο της (δυναμικής) κλήσης διαδικασιών έχει μέσο fan-out μεγαλύτερο του 2, τότε οι διαδικασίες-φύλλα αποτελούν την πλειοψηφία των (δυναμικά) ενεργοποιούμενων διαδικασιών.

Διατηρούμενοι Καταχωρητές - *saved registers*: **`s0`** έως και **`s11`** (12 καταχωρητές, συνολικά): Η τιμή των καταχωρητών αυτών διατηρείται μετά από ένα κάλεσμα διαδικασίας (preserved across call), δηλαδή είναι **ευθύνη της καλούμενης διαδικασίας** να σώσει την παλαιά τιμή κάθε τέτοιου καταχωρητή πριν την χαλάσει ("**callee-saved**"), και να την επαναφέρει στη θέση της πριν επιστρέψει στην καλούσα διαδικασία. Αρα, η καλούσα διαδικασία μπορεί να αφήνει χρήσιμες τιμές σε αυτούς τους καταχωρητές, πριν καλέσει άλλες διαδικασίες, και να τις ξαναβρίσκει μετά την επιστροφή από αυτές, χωρίς να χρειάζεται –η καλούσα διαδικασία– να κάνει κάτι ιδιαίτερο γι' αυτό.

Μεταβλητές και τιμές των οποίων η διάρκεια ζωής (lifetime) περιλαμβάνει δύο ή περισσότερα καλέσματα διαδικασιών, με επανειλημμένη χρήση της παλαιάς τιμής τους μεταξύ των καλεσμάτων, συμφέρει να τοποθετούνται σε τέτοιους καταχωρητές, **`s`**: Γιά κάθε καταχωρητή **`s`** που χρησιμοποιεί η τρέχουσα διαδικασία, αρκεί *ένα* σώσιμο της τιμής που (πιθανόν) έχει αφήσει εκεί κάποιος πρόγονος, και μία επαναφορά της τιμής του προγόνου. Το σώσιμο του "προγονικού" περιεχομένου πρέπει να γίνει μετά την έναρξη εκτέλεσης της τρέχουσας διαδικασίας και πριν χρησιμοποιηθεί γιά πρώτη φορά ο καταχωρητής, η δε επαναφορά του προγονικού αυτού περιεχομένου πρέπει να γίνει μετά το τέλος χρήσης του καταχωρητή από την τρέχουσα διαδικασία και πριν αυτή επιστρέψει στον γονέα της. Αντίθετα, δεν απαιτείται σώσιμο και επαναφορά κάθε φορά που η τρέχουσα διαδικασία καλεί παιδιά της –πράγμα που θα χρειάζονταν εάν η μεταβλητή αυτή είχε τοποθετηθεί σε καταχωρητή τύπου **`t`**, και γι' αυτό δεν θα συνέφερε εδώ η τοποθέτηση σε **`t`** αφού τα καλέσματα παιδιών στη διάρκεια ζωής είναι δύο ή περισσότερα.

Τιμές των οποίων η διάρκεια ζωής (lifetime) περιλαμβάνει ακριβώς ένα κάλεσμα θυγατρικής διαδικασίας κοστίζουν το ίδιο από πλευράς σωσίματος-επαναφοράς καταχωρητών, είτε τοποθετηθούν σε καταχωρητή τύπου **`t`**, είτε σε καταχωρητή **`s`**.

6.2 Ορίσματα, Επιστρεφόμενες Τιμές, και άλλοι Καταχωρητές

- **Ορίσματα Διαδικασίας** - *Procedure Arguments* - καταχωρητές **`a0`** έως και **`a7`**: περιέχουν τα πρώτα 8 ορίσματα (arguments) της διαδικασίας. Αν υπάρχουν περισσότερα από 8 ορίσματα, τα υπόλοιπα περνιούνται στη στοίβα.
  – Αν υπάρχουν λιγότερα από 8 ορίσματα, τότε οι υπόλοιποι (μη χρησιμοποιούμενοι) καταχωρητές **`a`** χρησιμοποιούνται σαν τους προσωρινούς καταχωρητές, **`t`**. Π.χ., αν η παρούσα διαδικασία έχει δύο ορίσματα μόνο, τότε είναι ελεύθερη να χρησιμοποιήσει τους a2, ..., a7 σαν προσωρινούς καταχωρητές χωρίς να σώσει τις τιμές τους, αρκεί να μην ζητά να διατηρούνται αυτές οι τιμές όταν καλεί παιδιά της: αν τα παιδιά της έχουν πολλά ορίσματα τότε πρέπει να τους χρησιμοποιήσει γιά να βάλει εκεί ορίσματα των παιδιών, κι αν τα παιδιά έχουν λίγα ορίσματα τότε επιτρέπεται τα παιδιά αυτά να χαλάσουν τις τιμές των a2, ..., a7.
  – Κατ' αναλογία, και οι a0, a1,... περιέχουν τα ορίσματα της παρούσας διαδικασίας μόνο μέχρι το κάλεσμα του πρώτου παιδιού της –γιά το κάλεσμα αυτό, οι

καταχωρητές αυτοί πρέπει να χρησιμοποιηθούν γιά τα ορίσματα των παιδιών (ή σαν προσωρινοί των παιδιών, αν τα ορίσματά τους είναι λιγότερα).

- **Επιστρεφόμενες Τιμές Διαδικασίας** - *Procedure Return Values* - καταχωρητής **`a0`** (ίσως και **`a1`**): η τιμή που η κάθε διαδικασία επιστρέφει στο γονέα της επιστρέφεται στον καταχωρητή **`a0`** (σε περίπτωση επιστροφής δύο τιμών (σε γλώσσες άλλες από την C?), τότε η δεύτερη επιστρέφεται στον καταχωρητή **`a1`**). Προφανώς, μέσα σε μία διαδικασία, η τιμή που μας επέστρεψε ένα παιδί μας ισχύει μόνο μέχρι το επόμενο κάλεσμα παιδιού (αφού, γιά εκείνο, θα χρειαστεί να βάλουμε νέο όρισμα στον **`a0`**), ή μόνο μέχρι λίγο πριν επιστρέψουμε εμείς οι ίδιοι (αφού, γιά την επιστροφή μας, θα χρειαστεί να βάλουμε τη δική μας επιστρεφόμενη τιμή στον **`a0`**).

- **Διεύθυνση Επιστροφής** - *Return Address* - καταχωρητής **`ra`** (`x1`): Κάθε κάλεσμα διαδικασίας πρέπει να καταγράψει κάπου τη διεύθυνση της επόμενης εντολής του καλούντος (caller), στην οποία και πρέπει να επιστρέψει (άλμα - jump) ο καλούμενος (callee) μετά το πέρας της εργασίας του –δηλαδή καταγράφουν το ποιός είναι ο caller, που φυσικά δεν είναι πάντα ένας και ο αυτός. Η κατάλληλη προς τούτο δομή δεδομένων είναι η στοίβα, όπως ξέρουμε από άλλα μαθήματα και θα υπενθυμίσουμε και στην επόμενη παράγραφο. Οι επεξεργαστές CISC (με πολύπλοκες εντολές) συνήθως καταγράφουν τη διεύθυση επιστροφής στη στοίβα, στη μνήμη. Αντίθετα, οι επεξεργαστές RISC συνήθως την καταγράφουν σε καταχωρητή, στον x1 (`ra`) σύμφωνα με τη σύμβαση του RISC-V. Αυτό έχει το πλεονέκτημα ότι γλυτώνουμε μία μεταφορά προς και μία από τη μνήμη (στοίβα) γιά το κάθε κάλεσμα διαδικασίας-**φύλλου**, όπως θα πούμε τώρα. Όπως λέγαμε, εάν το δένδρο της (δυναμικής) κλήσης διαδικασιών έχει μέσο fan-out μεγαλύτερο του 2, τότε οι διαδικασίες-φύλλα αποτελούν την πλειοψηφία των (δυναμικά) ενεργοποιούμενων διαδικασιών.
  – Κάθε κάλεσμα διαδικασίας καταστρέφει το προηγούμενο περιεχόμενο του καταχωρητή `ra`, άρα κάθε διαδικασία που καλεί άλλες διαδικασίες (δηλ. κάθε διαδικασία που δεν είναι φύλλο στο δέντρο των καλεσμάτων) πρέπει να σώσει τη διεύθυνση επιστροφής της στην αρχή της εκτέλεσής της (πριν το πρώτο κάλεσμα), και να την επαναφέρει αφού επιστρέψει το τελευταίο παιδί της, πριν να επιστρέψει η ίδια. (Το σημείο αυτό είναι ασαφές στο σχήμα 2.11 του βιβλίου (σελ. 159 Ελληνικής έκδοσης, p. 104 Αγγλικής): η διεύθυνση επιστροφής στην οποία αναφέρεται εκείνο το σχήμα είναι η διεύθυνση του παρόντος καλέσματος, και όχι η διεύθυνση επιστροφής της διαδικασίας μέσα από την οποία γίνεται το κάλεσμα).

- **Δείκτης Στοίβας** - *Stack Pointer* - καταχωρητής **`sp`** (`x2`): Όπως θα πούμε στην επόμενη παράγραφο, περιέχει πάντα τη διεύθυνση του τελευταίου έγκυρου (εν χρήσει) Byte της στοίβας (runtime procedure activation frame stack), πέραν του οποίου (μικρότερες διευθύνσεις) υπάρχουν σκουπίδια και μόνον. Η κάθε διαδικασία τον αλλάζει (εν δυνάμει) την ώρα που τρέχει, αλλά πριν επιστρέψει στο γονέα της πρέπει να τον επαναφέρει εκεί που αυτός ήταν όταν ο γονέας την κάλεσε.

- **Δείκτης Πλαισίου** - *Frame Pointer* - **`fp`** - *προαιρετικός* - όταν χρησιμοποιείται είναι ο `x8` (ίδιος με τον `s0`): Τον χρειαζόμαστε μόνον γιά διαδικασίες των οποίων το activation frame περιλαμβάνει δεδομένα μεταβλητού μεγέθους (μεγέθους που ποικίλει από ενεργοποίηση σε ενεργοποίηση, π.χ. local arrays μεγέθους που είναι συνάρτηση των ορισμάτων της διαδικασίας). Σε αυτές τις περιπτώσεις, ο μεν sp δείχνει την κορυφή του activation frame, ο δε fp δείχνει τη βάση του activation frame. Επειδή το activation frame έχει μεταβλητό μέγεθος, θα ήταν δύσκολο να κάνουμε index σε σχέση με τον sp τις προσπελάσεις τοπικών μεταβλητών και ποσοτήτων που βρίσκονται στη βάση του activation frame, άρα τις κάνουμε index σε σχέση με τον fp.

- **Δείκτης Περιοχής Καθολικών Δεδομένων** - *Global Pointer* - καταχωρητής **`gp`** (`x3`): Δείχνει στη μέση μιάς περιοχής μνήμης μεγέθους 4 KBytes τα περιεχόμενα της οποίας μπορούμε να προσπελάσουμε με μία μόνο εντολή load ή store χρησιμοποιώντας αυτόν τον καταχωρητή και το offset της ίδιας της εντολής που άρα χωράει σε 12 bits αφού η περιοχή είναι 4 KBytes. Επομένως, στην περιοχή αυτή συμφέρει να τοποθετούνται οι καθολικές (global) βαθμωτές (scalar) μεταβλητές του προγράμματος. Δεν τον αλλάζει (ούτε τον σώζει) καμία διαδικασία.

- **Δείκτης Νήματος** - *Thread Pointer* - καταχωρητής **`tp`** (`x4`): χρησιμοποιείται μάλλον σε πολυ-νηματικά (multi-threaded) προγράμματα, δηλαδή στον παράλληλο προγραμματισμό, μάλλον σαν δείκτης στο thread-local storage, αλλά δεν έχω περισσότερες πληροφορίες.

- **zero** - καταχωρητής **`x0`**: περιέχει την σταθερά (hardwired) μηδέν. Επιτρέπονται εγγραφές σε αυτόν, αλλά **δεν** αλλάζουν το μηδενικό (hardwired) περιεχόμενο του.

6.3 Η Στοίβα γιά Τοπικές Μεταβλητές και Σώσιμο Καταχωρητών

Αν οι διαδικασίες δεν ήταν αναδρομικές, δηλαδή αν απαγορεύονταν να καλέσει μιά διαδικασία τον εαυτό της –είτε άμεσα είτε έμμεσα μέσω άλλων που αυτή καλεί– τότε θα αρκούσε μιά συγκεκριμένη περιοχή στη μνήμη γιά την κάθε διαδικασία, όπου αυτή να φυλάει όσες τοπικές της μεταβλητές δεν χωράνε σε καταχωρητές, καθώς και τις τιμές των καταχωρητών που πρέπει να σώσει και αργότερα να επαναφέρει. Ομως αυτό, (α) ούτε θα συνέφερε, διότι θα κρατούσε χώρο γιά *κάθε* διαδικασία (που είναι πολλές), αντί μόνο γιά τις ενεργοποιημένες (που είναι λιγότερες), και επίσης (β) οι σημερινές γλώσσες προγραμματισμού επιτρέπουν την αναδρομή, κι έτσι μιά τέτοια λύση δεν θα δούλευε. Δεδομένου ότι τα καλέσματα και οι επιστροφές διαδικασιών λειτουργούν με τρόπο "last in first out" (LIFO), δηλαδή η τελευταία που καλέστηκε είναι η πρώτη που θα επιστρέψει, η φυσική δομή δεδομένων γιά δυναμική παραχώρηση και απελευθέρωση μνήμης στις διαδικασίες και από τις διαδικασίες είναι η **στοίβα** (stack).

Η "Στοίβα των Πλαισίων Ενεργοποίησης Διαδικασιών" (Procedure Activation Frame Stack, ή Runtime Stack) –κατά το επίσημο όνομά της– συνήθως (ίσως πάντα) ξεκινάει από τις μεγαλύτερες διευθύνσεις που έχει στη διάθεσή της η διεργασία του χρήστη (user process) (σε αντιδιαστολή με τον πυρήνα του Λειτουργικού Συστήματος - O.S. kernel), και μεγαλώνει (όποτε καλείται νέα διαδικασία) προς τις μικρότερες διευθύνσεις. (Αυτό βοηθά στο να αργήσει να συναντηθεί (ελπίζουμε ποτέ να μην συναντηθεί) με τον "σωρό" (heap), όπου δίνει χώρο η malloc(), και ο οπίος μεγαλώνει από τις μικρές προς τις μεγάλες διευθύνσεις). Στους 32-μπιτους επεξεργαστές, συχνά η στοίβα ξεκινάει από τη διεύθυνση 7F.FF.FF.FF (δεκαεξαδικό), δηλαδή από τη μέση της μνήμης (όπου ο υπόλοιπος μισός χώρος διευθύνσεων συχνά είναι γιά το OS kernel), και μεγαλώνει προς τις μικρότερες διευθύνσεις (προς τη διεύθυνση 0). Ο Stack Pointer (καταχωρητής `sp`) δείχνει στην τελευταία χρησιμοποιούμενη λέξη της στοίβας. (Στον κανονικό RISC-V, ο sp κρατιέται πάντα ευθυγραμμισμένος σε ακέραια πολλαπλάσια των 16 Bytes (quad words), αλλά εμείς εδώ δεν θα το επιβάλουμε αυτό). Πριν αποθηκεύσουμε N νέες 32-μπιτες λέξεις στη στοίβα πρέπει να ελαττώσουμε τον sp κατά 4N (το μέγεθος N λέξεων των 4 Bytes καθεμία) (ή κατά 8N αν οι λέξεις είναι 64-μπιτες), πράγμα που ισοδυναμεί με δήλωση από πλευράς του προγράμματός μας (προς το λειτουργικό σύστημα, σε περίπτωση διακοπής (interrupt - page fault)) ότι τώρα η στοίβα μας είναι τώρα μεγαλύτερη κατά N λέξεις. Η αντίστροφη πράξη (αύξηση του $sp κατά 4N (ή 8N) –απελευθέρωση μνήμης) πρέπει να γίνει αφού πάρουμε τις αποθηκευμένες λέξεις και δεν τις χρειαζόμαστε άλλο πιά στη στοίβα. Οι τιμές που βρίσκονται αποθηκευμένες στη στοίβα προσπελαύνονται συνήθως μέσω εντολών load και store με διευθυνσιοδότηση σχετικά με τον sp. Καθώς ο sp αυξομειώνεται λόγω παραχώρησης/απελευθέρωσης μνήμης, η απόσταση των αποθηκευμένων τιμών στη στοίβα από τον sp αλλάζει, αλλά παραμένει πάντοτε γνωστή στον compiler/προγραμματιστή (εκτός των περιπτώσεων δυναμικής παραχώρησης μνήμης στη στοίβα –πράγμα σπάνιο στις γλώσσες προγραμματισμού– οπότε και απαιτείται η χρήση του fp).

Τοπικές (Local) και Καθολικές (Global) Μεταβλητές:
Οι "καθολικές" και οι "τοπικές" μεταβλητές είναι έννοιες των γλωσσών προγραμματισμού υψηλού επιπέδου (HLL - π.χ. C, κλπ). Γιά τον Assembler δεν υπάρχουν αυτές οι έννοιες, ενώ η υλοποίηση των εννοιών αυτών σε επίπεδο γλώσσας Assembly επαφίεται στον προγραμματιστή (ή στον compiler). Στις HLL, μια καθολική μεταβλητή αντιστοιχεί πάντα σε μια δεδομένη, σταθερή θέση μνήμης (ή καταχωρητή, αν και σπανίως αυτές τοποθετούνται σε καταχωρητές), ανεξαρτήτως του ποιά διαδικασία εκτελείται κάθε στιγμή· η τιμή μιας καθολικής μεταβλητής ούτε χρειάζεται να αποθηκευτεί ποτέ στη στοίβα, ούτε επανατίθεται ποτέ σε παλαιές τιμές της από τη στοίβα. Αντίθετα, μιά τοπική μεταβλητή έχει νόημα μόνο όσο είναι ενεργή η διαδικασία στην οποία ανήκει, και είναι ανύπαρκτη πριν την εκκίνηση

της διαδικασίας αυτής ή μετά τον τερματισμό (επιστροφή) της διαδικασίας· εάν η διαδικασία επιστρέψει και ξανακαλεστεί, η παλαιά τιμή της τοπικής μεταβλητής έχει χαθεί. Επίσης, τοπικές μεταβλητές με το ίδιο όνομα αλλά σε διαφορετική διαδικασία (ή σε διαφορετική ενεργοποίηση της ίδιας (αναδρομικής) διαδικασίας!) είναι διαφορετικές και άσχετες μεταξύ τους!

Συνήθως, οι μεταφραστές (compilers) τοποθετούν την κάθε καθολική μεταβλητή σε μία σταθερή διεύθυνση μνήμης –όχι στη στοίβα· η διεύθυνση αυτή δεν αλλάζει από διαδικασία σε διαδικασία. Αντίθετα, οι τοπικές μεταβλητές αντιστοιχούν σε μιά θέση στη στοίβα, σε δεδομένη απόσταση **σχετικά** με τον stack-pointer, η οποία θέση υπάρχει μόνον όση ώρα είναι ενεργοποιημένη η αντίστοιχη διαδικασία, και η οποία θέση –σαν απόλυτη διεύθυνση μνήμης– πολύ πιθανόν να αλλάζει από ενεργοποίηση σε ενεργοποίηση της διαδικασίας. Εάν ένα αντίτυπο της μεταβλητής κρατηθεί σε καταχωρητή (ή, όπως η συνηθισμένη βελτιστοποίηση, κρατηθεί μόνο το "αντίτυπο", χωρίς το πρωτότυπο), τότε η διαδικασία αυτή (σε συνεργασία με τις άλλες) έχει την ευθύνη να σώζει το αντίτυπο στη στοίβα και να το επαναφέρει όποτε υπάρχει κίνδυνος μια άλλη διαδικασία να χρησιμοποιήσει τον καταχωρητή αυτό διαφορετικά, όπως αναλύσαμε παραπάνω.

Παρατηρήστε ότι οι παραπάνω έννοιες δεν είναι ίδιες με τις "καθολικές ετικέτες" (global labels) που γιά τον Assembler ορίζονται με την οδηγία "`.globl`". Οι ετικέτες του Assembler είναι απλά διευθύνσεις μνήμης –είτε δεδομένων, είτε εντολών μέσα σε πρόγραμμα, είτε οτιδήποτε. Καθολική ετικέτα είναι απλώς μια διεύθυνση την οποία ζητάμε από τον Assembler να την τοποθετήσει στον Πίνακα Συμβόλων (Symbol Table), μαζί με το συμβολικό της όνομα, ο οποίος πίνακας περιλαμβάνεται στο αρχείο δυαδικού κώδικα (binary/object code file). Τότε, ο Linker, όταν ξαναβρεί το ίδιο όνομα σε άλλο αρχείο, θεωρεί ότι πρόκειται γιά την ίδια διεύθυνση, και "συνενώνει" τις δύο αναφορές. Με άλλα λόγια, μιά καθολική ετικέτα (σύμβολο) είναι η ίδια και καθολικά ορατή ανάμεσα σε όλα τα συνενούμενα (linked) αρχεία ενός προγράμματος.

6.4 jal, jalr: Κάλεσμα Διαδικασιών, Επιστροφή, άλλες Χρήσεις

Όπως έχουμε ξαναπεί, το κάλεσμα διαδικασίας μοιάζει με άλμα, αλλά επιπλέον πρέπει να κρατήσουμε πληροφορία γιά το πού να επιστρέψει η καλούμενη διαδικασία, αφού αυτή μπορεί να καλείται από πολλαπλά, διαφορετικά σημεία και πρέπει να ξέρει σε ποιό από αυτά να επιστρέψει. Στις αρχιτεκτονικές CISC, η εντολή καλέσματος, που συνήθως λέγεται "call", αποθηκεύει αυτή τη *διεύθυνση επιστροφής* στη στοίβα (στη μνήμη)· σε μερικές από αυτές, η ίδια αυτή εντολή call μπορούσε επιπλέον και να αποθηκεύει και καταχωρητές στη στοίβα. Στις αρχιτεκτονικές RISC, η εντολή καλέσματος αποθηκεύει τη διεύθυνση επιστροφής σε *καταχωρητή*, και επειδή μόνον οι εντολές store γράφουν στη μνήμη, και επειδή έτσι μπορούμε να γλυτώσουμε αυτή την εγγραφή και ανάγνωση από τη στοίβα γιά όλες τις κλήσεις σε διαδικασίες-φύλλα (§6.2). Προκειμένου να τονιστεί η διαφορά από την παραδοσιακή εντολή call, αυτές οι εντολές καλέσματος με την απλούστερη συμπεριφορά ονομάστηκαν **Jump and Link** (άλμα και συνένωση). Στον RISC-V, η εντολή αυτή γράφεται **`jal`**, ακολουθεί το J-format (§4.3), και κάνει τα εξής:

`jal rd, Imm20` ⇒ `rd ← PC+4; PC ← PC + 2×Imm20`*(signed)*

Με άλλα λόγια, η jump-and-link στον RISC-V, *(α)* γράφει στον καταχωρητή προορισμού τη διεύθυνση της "από κάτω" της εντολής (PC+4), δηλαδή τη διεύθυνση όπου πρέπει να επιστρέψει η καλούμενη διαδικασία, και *(β)* αλλάζει τον PC (εκτελεί άλμα) ούτως ώστε η επόμενη εντολή που θα εκτελεστεί (δηλ. η πρώτη εντολή της καλούμενης διαδικασίας) να είναι αυτή που βρίσκεται σε (προσημασμένη) απόσταση (offset) από την παρούσα εντολή όση ορίζει η 20-μπιτη σταθερά Imm20, και η οποία μετρά σε μονάδες half-words (ζευγάρια Bytes - δηλαδή διπλασιάζεται γιά να δώσει Byte Address), δηλαδή PC-relative και σε διευθύνσεις ακέραια πολλαπλάσια του 2, γιά τους ίδιους λόγους που έτσι μετρούσαν και οι εντολές διακλάδωσης του RISC-V. Ο καταχωρητής προορισμού, όπου γράφεται η διεύθυνση επιστροφής, είναι συνήθως ο ra (δηλαδή ο x1), όπως είπαμε στην §6.2, οπότε ο RISC-V Assembler δεέχεται και την ψευδοεντολή "jal label" και την μεταφράζει σε "jal x1, label". Μπορεί όμως ο rd να είναι και οιοσδήποτε άλλος καταχωρητής: σε μερικά περιβάλλοντα χρησιμοποιείται και ο t0 (δηλ. ο x5) ως "alternate link register" (π.χ. στο κάλεσμα "millicode

routines", όπως γιά σώσιμο και επαναφορά καταχωρητών σε compressed code), και επίσης μπορεί να είναι ο x0 στην πολύ συνηθισμένη ψευδοεντολή **jump** που την έχουμε αναφέρει ήδη από την §1.3, και ξανά στην §5.7:

Ψευδοεντολή Jump: **`j label ≡ jal x0, label`** ⇒ `PC ← PC + 2×Imm20`*(signed)*

Η άλλη εντολή καλέσματος –αλλά που επίσης χρησιμοποιείται σε εξειδικεύσεις της και γιά άλλους σκοπούς– είναι η έμμεση (indirect / register-indexed) jump-and-link. Αυτή ακολουθεί το I-format (§4.3), λέγεται **Jump-and-Link-Register (`jalr`)**, και η διευθυνσιοδότησή της είναι η ίδια όπως των εντολών load (και store), δηλαδή το Offset είναι 12-μπιτο (πάντοτε signed) και *δεν* πολλαπλασιάζεται επί 2, παρ' ότι εδώ μιλάμε γιά τον PC:

`jalr rd, Offset(rs1)` ⇒ $\texttt{rd}_{\texttt{new}} \leftarrow \texttt{PC+4; PC} \leftarrow \texttt{rs1}_{\texttt{old}} + \texttt{Offset}$*(signed)*

Όπως και η απλή jump-and-link, έτσι και αυτή γράφει τη διεύθυνση της επόμενής της εντολής, PC+4, στον καταχωρητή προορισμού –που στην περίπτωση καλέσματος θα είναι συνήθως ο ra (= x1). Όμως η διαδικασία που καλείται (όταν προκειται πραγματικά γιά κάλεσμα) *δεν* είναι πάντα η ίδια, στατικά καθορισμένη όπως ήταν γιά την jal όπου η διεύθυνση άλματος προέκυπτε σαν μιά σταθερή απόσταση σε σχέση με την τρέχουσα εντολή (PC). Εδώ, αντιθέτως, η διεύθυνση προορισμού προκύπτει μέσω του (αυθαίρετου) καταχωρητή rs1, ο οποίος μπορεί να περιέχει έναν οιοδήποτε αυθαίρετο pointer, που μπορεί κάλιστα να μεταβάλεται από κάλεσμα σε κάλεσμα: πρόκειται γιά **κάλεσμα μέσω pointer**. (Επιτρέπεται, αν και δεν ξέρω εάν χρησιμεύει σε κάτι, ο καταχωρητής rd να είναι ο ίδιος με τον rs1, στην οποία περίπτωση η παλαιά τιμή του καταχωρητή χρησιμοποιείται γιά τον υπολογισμό της νέας τιμής του PC, ενώ αντίστροφα η παλαιά τιμή του PC, συν 4, γράφεται σαν νεά τιμή του καταχωρητή).

Το Offset (12 bits) μπορεί πολλές φορές να είναι άχρηστο, οπότε το βάζουμε 0, αλλά μερικές φορές μπορεί και να χρησιμοποιείται, π.χ., εάν η αμέσως προηγούμενη εντολή φορτώνει τα 20 bits της σταθεράς της Imm20 στο αριστερό μέρος του rs1, τότε το 12-μπιτο Offset προστιθέμενο σε αυτά δημιουργεί έναν αυθαίρετο 32-μπιτο αριθμό από τις δύο σταθερές, κι έτσι μας δίνει έναν τρόπο άλματος στην οιαδήποτε αυθαίρετη διεύθυνση μνήμης. Το 12-μπιτο Offset *δεν* διπλασιάζεται εδώ, παρ' ότι χρησιμοποιείται σε πράξη που το αποτέλεσμά της προορίζεται γιά τον PC, επειδή η εντολή αυτή έχει το ίδιο format (και χρησιμοποιεί τα ίδια κυκλώματα) με τις εντολές load (και παρόμοια με τις store), οι οποίες δεν διπλασιάζουν το Offset τους (και δεν χρειάζεται διπλασιασμός όταν τα 12 bits του Offset προστίθενται στα 20 bits αριστερά από ένα Imm20 γιά να δημιουργήσουν μιά αυθαίρετη 32-μπιτη σταθερά (διεύθυνση)). Παρ' όλα αυτά, στο τελικό αποτέλεσμα της πρόσθεσης rs1+Offset, το hardware του RISC-V μηδενίζει το (ένα) δεξιότερο (LS) bit του αποτελέσματος πριν το γράψει στον PC (ή αλλιώς: ο PC έχει στην πραγματικότητα ένα λιγότερο flip-flop δεξιά, και υποτίθεται ότι συμπληρώνεται πάντα με ένα μηδενικό δεξιά όταν είναι να πάει στη μνήμη (η οπία μνήμη, όταν είναι 16-μπιτη ή φαρδύτερη, αγνοεί ούτως ή άλλως το δεξιότερο (τουλάχιστο) bit)).

Στον RISC-V, όπως η (ψεύδο)εντολή jump είναι στην πραγματικότητα ειδική περίπτωση της εντολής jump-and-link, έτσι και η (ψεύδο)εντολή **Jump-Register**, που είδαμε στην §5.7, συντίθεται σαν ειδική περίπτωση της jump-and-link-register, ως εξής (και η επιστροφή από διαδικασία είναι συνήθως η ακόμα πιό ειδική περίπτωση "jr ra", δηλαδή "jalr x0, 0(x1)"):

Ψευδοεντολή Jump-Register: **`jr rs1 ≡ jalr x0, 0(rs1)`** ⇒ `PC ← rs1`

6.5 Παράδειγμα Αναδρομικής Διαδικασίας: factorial

Θα δώσουμε εδώ, σε ελαφρά παραλλαγή, το παράδειγμα διαδικασίας, που καλεί και η ίδια μιά διαδικασία-παιδί, της §2.8 του βιβλίου· μάλιστα, πρόκειται γιά *αναδρομική* διαδικασία, δηλαδή το παιδί που καλεί είναι μιά νέα "μετενσάρκωση" (ενεργοποίηση, "κλονοποίηση") του ίδιου του εαυτού της. Πρόκειται γιά μιά απλή διαδικασία υπολογισμού του *n!* –του *n* παραγοντικού (factorial)– παρά το γεγονός ότι, προφανώς, το *n!* μπορεί να υπολογιστεί πολύ απλούστερα και γρηγορότερα με έναν απλό βρόχο πολλαπλασιασμών. Το παράδειγμα είναι διατυπωμένο γιά 64-μπιτο RISC-V:

```
long long int fact( long long int n )
{ if ( n<2 ) { return(1); } else { return( n * fact(n-1) ); } }
```

Επειδή η διαδικασία fact() καλεί (τουλάχιστο μία) άλλη διαδικασία, θα χρειαστεί να σώσει στη στοίβα τουλάχιστο τον καταχωρητή ra της, και ενδεχομένως και άλλον ή άλλους καταχωρητές, και να τον(τους) επαναφέρει μετά την επιστροφή του (των) παιδιού(ών) της. Στο βιβλίο, το σώσιμο αυτό γίνεται ευθύς με την είσοδο στην fact() ("με το καλημέρα σας"). Εμείς εδώ, όμως, θα κάνουμε μιά βελτιστοποίηση: εάν το if() "στρίψει" προς το "then", τότε δεν υπάρχει κάλεσμα παιδιού, άρα δεν χρειάζεται ούτε σώσιμο-επαναφορά· μόνον εάν το if() στρίψει προς το else, μόνον τότε υπάρχει κάλεσμα άρα μόνον τότε χρειάζεται σώσιμο και επαναφορά.

Εντός του else{}, απαιτείται ο πολλαπλασιασμός του *n* (πρώτο όρισμα της δικής μας διαδικασίας, άρα στον καταχωρητή a0) επί την τιμή που επιστρέφει το κάλεσμα του παιδιού. Ο πολλαπλασιασμός αυτός, επομένως, μπορεί να γίνει μόνον μετά την επιστροφή από το κάλεσμα του παιδιού. Όμως, το κάλεσμα ενός παιδιού καταστέφει (εν δυνάμει) τα ορίσματα (και τις τοπικές μεταβλητές) της δικής μας διαδικασίας, άρα θα καταστρέψει (εδώ σίγουρα) (και) τον καταχωρητή a0 (δηλαδή το όρισμά μας *n*). Άρα πριν το κάλεσμα πρέπει να σώσουμε στη στοίβα, εκτός από τον ra, *και* τον καταχωρητή a0. Αφού υποθέτουμε 64-μπιτο RISC-V, ο καθένας καταχωρητής πιάνει 8 Bytes στη στοίβα, άρα σύνολο 16 Bytes εδώ, άρα πριν το σώσιμο θα πρέπει να μεγαλώσουμε τη στοίβα κατά 16 Bytes (sp ← sp-16), και μετά την επαναφορά των καταχωρητών από τη στοίβα θα πρέπει να την μειώσουμε κατά τα ίδια 16 Bytes (sp ← sp+16).

Πριν καλέσουμε το παιδί μας, πρέπει να του δώσουμε σαν (πρώτο και μοναδικό) όρισμα το n-1, και αυτό πρέπει να του το δώσουμε, σύμφωνα με τη σύμβαση, στον καταχωρητή a0. Όταν θα επιστρέψει σε εμάς το παιδί μας, θα βρούμε την επιστρεφόμενή του τιμή, fact(n-1), σύμφωνα με τη σύμβαση, πάλι στον καταχωρητή a0. Αντίστοιχα και εμείς οι ίδιοι, πριν επιστρέψουμε στο γονέα μας, πρέπει να βάλουμε στον καταχωρητή a0 την τιμή που επιστρέφουμε (1 στην περίπτωση then, αλλιώς n*fact(n-1) στην περίπτωση else). Θα χρειαστούμε και έναν προσωρινό καταχωρητή –ας χρησιμοποιήσουμε τον t0 (υπενθύμιση: αφού είναι "tmp", *δεν* χρειάζεται σώσιμο-επαναφορά των τυχόν περιεχομένων του από τους προγόνους). Με αυτές τις σκέψεις, η διαδικασία fact() μεταφράζεται ως εξής σε Assembly του RISC-V:

```
fact:  addi  t0, zero, 2   # immediate 2 needed for "if(n<2)"
       bge   a0, t0, elseF # if n<2 false, i.e. if n≥2 goto ELSE
       addi  a0, zero, 1   # THEN: create return-value 1, place in reg. a0
       jr    ra            # return --this is the end of the "then" clause
elseF: addi  sp, sp, -16   # PUSH1: allocate 16 Bytes on the stack
       sd    ra, 8(sp)     # PUSH2: save ra into first allocated word
       sd    a0, 0(sp)     # PUSH3: save my argument (n) into second word
       addi  a0, a0, -1    # create argument (n-1) into a0 for my child
       jal   ra, fact      # call my child procedure
       add   t0, a0, zero  # copy return value from my child into t0
                           # (because I need to restore my own argument into a0)
       ld    ra, 8(sp)     # POP1: restore ra from stack
       ld    a0, 0(sp)     # POP2: restore a0 from stack
       addi  sp, sp, 16    # POP3: dealloc the 16 B that I had allocated
       mul   a0, a0, t0    # multiply my own arg a0==n times the return
                           # value from my child that I had copied into t0, and
                           # place the result into a0, as my own return value
       jr    ra            # return
```
